

Test Set Augmentation Technique for Deep Learning Image Classifiers

Qiang Chen, Zhanwei Hui^{*}, and Jialuo Liu

Command and Control Engineering College, Army Engineering University of PLA, Nanjing, 210007, China

Abstract

Widely applied in various fields, deep learning (DL) is becoming the key driving force in the industry. Although it has achieved great success in artificial intelligence tasks, similar to traditional software, it has defects involving unpredictable accidents and losses due to failure. To ensure the quality of DL software, adequate testing needs to be carried out. In this paper, we propose a test set augmentation technique based on an adversarial example generation algorithm for image classification deep neural networks (DNNs). It can generate a large number of useful test cases, especially when test cases are insufficient. We briefly introduce the adversarial example generation algorithm and implement the framework of our method. We conduct experiments on classic DNN models and datasets. We further evaluate the test set by using a coverage metric based on states of the DNN.

Keywords: DNN; defects; adversarial examples; coverage metric; test case generation

(Submitted on May 16, 2019; Revised on June 30, 2019; Accepted on July 20, 2019)

© 2019 Totem Publisher, Inc. All rights reserved.

1. Introduction

DNNs [1] have been gradually applied in various fields, including speech recognition [2], image classification [3], text processing [4], and other areas. Especially in the field of image classification, DNNs show significant performance advantages and surpasses the recognition accuracy of human eyes [5]. This performance advantage makes the DL system widely applied in various scenarios and applications, such as face recognition [6] and satellite image recognition [7]. In safety-critical applications, such as medical diagnostics [8] and automatic driving [9], DNNs also play a vital and critical role. Although DNNs provide excellent convenience for people's daily lives, they also lead to blind pursuit and trust. However, some research has proven that DL systems are not as reliable as people think. The image classifiers based on DL can be easily fooled and produce confusing classification results when artificially crafted perturbation unperceived by humans is added to the natural images initially classified correctly. The crafted images are called adversarial examples. In 2014, Szegedy et al. [10] published a paper in ICLR2014 that officially proposed the concept of adversarial examples for the first time, and they gave a specific adversarial example generation algorithm: L-BFGS. The method crafted adversarial examples by finding the minimum perturbation in a limited sample space and used the Box-L-BFGS algorithm to get solutions. In the same year, Goodfellow et al. [11] proposed a fast-gradient sign method (FGSM) that attacks the loss function of the classifier. Different from the first few methods, which need to access the attacked DL models' internal white-box information, Papernot et al. [12] proposed a Jacobian-based saliency map approach (JSMA) for black-box attacks in 2015. Baluja and Fischer [13] trained a feedforward neural network to generate targeted adversarial examples; the trained model is called adversarial transformation networks (ATNs), which are somewhat similar to generation adversarial networks (GAN).

With the introduction of adversarial examples and the exposure of DNNs' defects, ensuring the quality of DL software has become the focus of people's attention [14]. However, DL software is different from traditional software, and, in a sense, it is not written by programmers but learned by machines from massive data. The internal logic cannot be explained [15], which disables the traditional software testing methods and techniques. On the other hand, we require many test cases to

^{*} Corresponding author.

E-mail address: 359435482@qq.com

accomplish the testing of DL software. Therefore, the problem of test case generation urgently needs to be solved, especially when the test cases are insufficient. This paper proposes a test case generation technique that can effectively augment the test corpus, based on an adversarial example generation algorithm for image classification DNNs. In this work, based on an existing adversarial example generation algorithm, DeepFool [16], we implement our test case generation method and apply it to DL image classifiers. The main contributions of this paper are as follows:

- This paper proposes a test case generation method for DNNs based on an adversarial example generation algorithm, which provides a new idea for DL software test case generation;
- We then apply a test coverage metric based on the "states" of the DL system to evaluate the generated test set;
- We conduct experiments on the MNIST [17] handwritten digit dataset and LeNet models, and we use an effective coverage metric to evaluate the generated corpus.

2. Background and Related Work

This section firstly introduces the existing test case generation methods for the image classification DL system, which is shown in Figure 1. Then, we propose our test case generation method, compare it with the current techniques, and analyze their connection and differences.

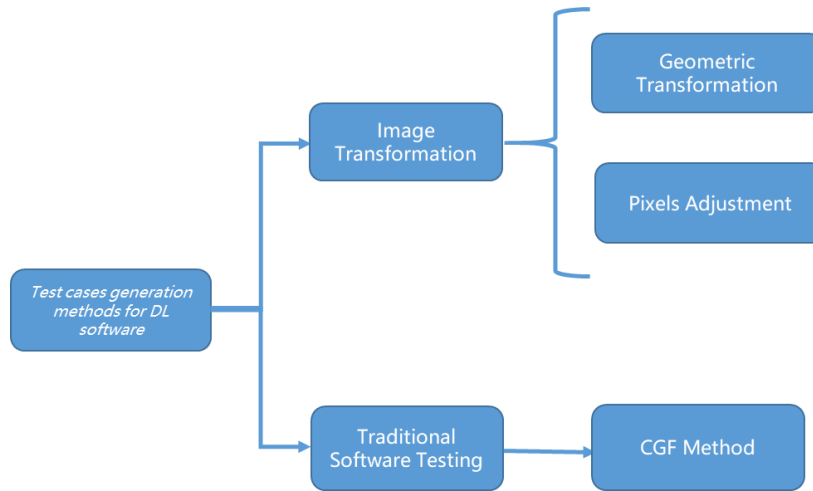


Figure 1. The current test case generation methods for DL image classifiers can be roughly classified into two types

The current test case generation methods for DL image classifiers can be roughly classified into two types. One is from the perspective of traditional software testing fields: the test coverage criteria applicable to DNNs are proposed, and new test cases are generated from the original seed test cases through the guidance of these test coverage criteria. We usually call this the coverage guided fuzzing (CGF) method [18-19]. The major related research works are as follows:

- Pei et al. [20] introduced the metric of neuron coverage for neural networks with rectified linear units (ReLU) as the activation functions. They believed a test suite achieves full coverage under this metric if some input exists, for which that hidden unit has a positive value for every unit in the neural network. Then, to find misbehavior, they cross-referenced multiple neural networks using gradient-based optimization.
- Ma et al. [21] generalized the neuron coverage criterion. In k-multisection coverage, for each neuron of the neural network, they took the range of values seen during training, divided it into k sections, and measured if every section had been "touched". In neuron boundary coverage, they measured if there were activations that have been made to go above and below a specific bound. Then, they evaluated how well these metrics were satisfied by the test suite.
- Sun et al. [22] introduced a family of metrics inspired by modified condition/decision coverage [23]. We describe their ss-coverage proposal as an example: given a neural network arranged into layers, a pair of neurons (n_1, n_2) in adjacent layers is said to be ss-covered by a pair (x,y) of inputs if the following three things are true: n_1 has a different sign for each of x, y; n_2 has a different sign for each of x, y; and all other elements of the layer containing n_1 have the same sign for x, y.

- Tian et al. [24] applied the neuron coverage metric to DNNs that are part of self-driving car software. Natural image transformations such as blurring, shearing, and transformation were performed, and the idea of metamorphic testing [25] was used to find misbehaviors and errors.
- Goodfellow et al. [26] proposed a CGF method for DNNs and implemented the open source tool TensorFuzz. The activation values of the logits layer or its previous layer of the neural network was used as a state of the neural network. The closest distance from a new state to the existing state set was calculated by the approximate nearest neighbor algorithm. When the closest distance exceeded a certain threshold, the new test case was considered to cover the new state of the DNN and joined the test corpus.
- Xie et al. [27] made new changes to TensorFuzz and designed a new framework for test case generation. They divided the process into transformation and mutation. The Transformation operation was mainly based on the nature of the image itself, including:

- a) Pixel value transformation P: change image contrast, image brightness, image blur, and image noise
- b) Affine transformation G: image translation, image scaling, image shearing, and image rotation

The Mutation operation was based on the CGF method, but with new test coverage criteria introduced.

Another type of method is from the perspective of the field of computer vision and image processing. These methods consider the nature of the image: by scaling, rotating, panning, changing the contrast, adding scenes, etc., images become different from the original ones with key semantics of the images seldom changed. This is beneficial in specific scenarios. For example, for automatic driving system testing, by adding different weather to the same road conditions or traffic signs, test cases in different scenarios can be efficiently generated; thus, considerable costs can be saved [28-29].

Researchers in the traditional software testing field pay more attention to the first type of method, because the methods are more in line with the research ideas of traditional software testing. Once better test coverage criteria for DL software are put forward, various test theories and techniques in the traditional software testing field can be migrated to the DL software testing work. However, in current research, it appears that the effectiveness of the proposed test coverage criteria cannot be interpreted and proved. There are still some scholars who reserve their views on the criteria's practical applicability. The CGF methods generate test cases by using random or heuristic algorithms, resulting in certain deficiencies in efficiency. Based on an adversarial example generation algorithm, this paper proposes a test case generation method for image classification DL software. This method considers that test cases close to the classification boundary are more likely to trigger DNN's new "states". The principle allows for faster approximation and generation of corner cases. Since no test coverage criteria are used in the test case generation process, it can be mutually verified and supplemented by the existing CGF methods. In the experimental part of this paper, a test coverage metric is used to evaluate the generated corpus. The rest of this paper is organized as follows: Section III illustrates the main algorithm principle and implementation, Section IV introduces the experimental process and presents the experimental results, and Section V concludes the paper and discusses the future work.

3. Algorithm

3.1. Adversarial Example Generation Algorithm, DeepFool

DeepFool is a simple and effective adversarial example generation algorithm that was proposed by Seyed-Mohsen et al. [16]. Similar to the FSGM method, this algorithm is white-boxed and needs to acquire the network structure and activation function.

For a binary classification problem, we believe that there is an ideal classification hyperplane, and all samples can be correctly divided into positive and negative categories. Taking the simplest binary classification problem as an example, as shown in Figure 2, it is assumed that the decision surface is a straight line, and the two sides of the border correspond to different classification results. We denote $\hat{k}(x) = \text{sign}(\omega^T x + b) = \text{sign}(f(x))$ as a binary classifier, $\mathcal{g} \triangleq \{x: f(x) = 0\}$ as the ideal classification hyperplane, x_0 as the sample, and $\Delta(x_0; f)$ as the shortest distance from the sample x_0 to the classification plane. For the classification result of a certain point to change, it must cross the boundary. The shortest distance is to move along the vertical direction of the plane. Therefore, we can calculate the perturbation vector:

$$\begin{aligned}
r_*(x_0) &:= \operatorname{argmin} \|r\|_2 \\
\text{subject to } \operatorname{sign}(f(x_0 + r)) &\neq \operatorname{sign}(f(x_0)) \\
&= -\frac{f(x_0)}{\|\omega\|_2^2} \omega
\end{aligned} \tag{1}$$

The negative sign is due to the direction of the perturbation vector pointing to the classification boundary.

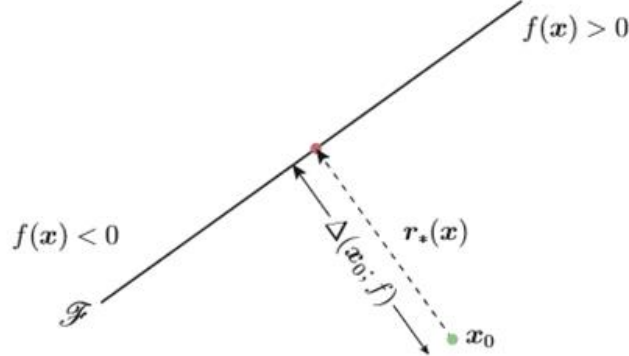


Figure 2. Adversarial examples for a linear binary classifier

For a multi-classifier, assuming that the class hyperplane is linear and the classifier has c outputs, then the classifier is $\hat{k}(x) = \operatorname{argmin}_k f_k(x)$. $f_k(x)$ is the k th dimension of $f(x)$, also regarded as the k sub-classifier. We can calculate the perturbation vector:

$$r_*(x_0) := \operatorname{argmin} \|r\|_2$$

s. t.

$$\exists k: \omega_k^T(x_0 + r) + b_k \geq \omega_{\hat{k}(x_0)}^T(x_0 + r) + b_{\hat{k}(x_0)} \tag{2}$$

Where ω_k is the k^{th} column of ω , that is, the weight vector of the k^{th} sub-classifier. For the classification result to change, it must be ensured that at least one classifier with a non-original label has greater value than the original classification function's outputs. The k^{th} classification boundary is:

$$\mathcal{G}_k = \{x: f_k(x) - f_{\hat{k}(x_0)}(x) = 0\} \tag{3}$$

The point x_0 and the convex area where it is located may be enclosed by the hyperplane P :

$$P = \bigcap_{k=1}^c \{x: f_{\hat{k}(x_0)}(x) \geq f_k(x)\} \tag{4}$$

At this time, the minimum distance from the point to the boundary of a classification function is:

$$\hat{l}(x_0) = \operatorname{arg} \min_{k \neq \hat{k}(x_0)} \frac{|f_k(x_0) - f_{\hat{k}(x_0)}(x_0)|}{\|\omega_k - \omega_{\hat{k}(x_0)}\|_2} \tag{5}$$

For a general nonlinear multi-classifier, the corpus P in the above formula, the region where the sample label is located is no longer a regular polyhedron. The DeepFool method uses an iterative approximation algorithm to approximate the region. \tilde{P}_i is used to replace and update P in the i th iteration:

$$\tilde{P}_i = \bigcap_{k=1}^c \{x: f_k(x_i) - f_{\hat{k}(x_0)}(x_i) + \nabla f_k(x_i)^T x - \nabla f_{\hat{k}(x_0)}(x_i)^T x \leq 0\} \tag{6}$$

It is worth mentioning that the DeepFool algorithm is greedy and cannot guarantee convergence to the optimal

solution. However, this algorithm can obtain a good approximation of the minimum perturbation. The pseudo code of DeepFool is as follows:

Procedure 1 The adversarial examples generation algorithm, DeepFool.

Input: Image x , classifier f .

Output: Perturbation \hat{r} .

```

1: Initialize  $x_0 \leftarrow x, i \leftarrow 0$ 
2: while  $\hat{k}(x_i) = \hat{k}(x_0)$  do
3:   for  $k \neq \hat{k}(x_0)$  do
4:      $w'_k \leftarrow \nabla f_k(x_i) - \nabla f_{\hat{k}(x_0)}(x_i)$ 
5:      $f'_k \leftarrow f_k(x_i) - f_{\hat{k}(x_0)}$ 
6:   end for
7:    $\hat{l} \leftarrow \arg \min_{k \neq \hat{k}(x_0)} \frac{|f'_k|}{\|w'_k\|_2}$ 
8:    $r_i \leftarrow \frac{|f'_l|}{\|w'_l\|_2} w'_l$ 
9:    $x_{i+1} \leftarrow x_i + r_i$ 
10:   $i \leftarrow i + 1$ 
11: end while
12: return  $\hat{r} = \sum_i r_i$ 

```

3.2. Overall Framework

The underlying architecture of the test case generation method in this paper is shown in Figure 3. Firstly, the input corpus is sampled, and a certain number of test cases are selected as seed use cases. Then, the DeepFool algorithm is executed on the seed use cases to obtain the noise vectors. By examining each noise vector, we can determine whether the noise vector will be reserved or not. The reserved noise vectors that pass checking procedure will be used to craft new test cases, and the crafted test cases will be added to the original input corpus. During the above processes, the input corpus is maintained as a specific structure. After the execution number is sufficient or the executing time reaches the threshold, the test cases of the input corpus are screened out to obtain a new corpus of test cases. The specific procedures are introduced as follows, and the corresponding pseudo code is attached:

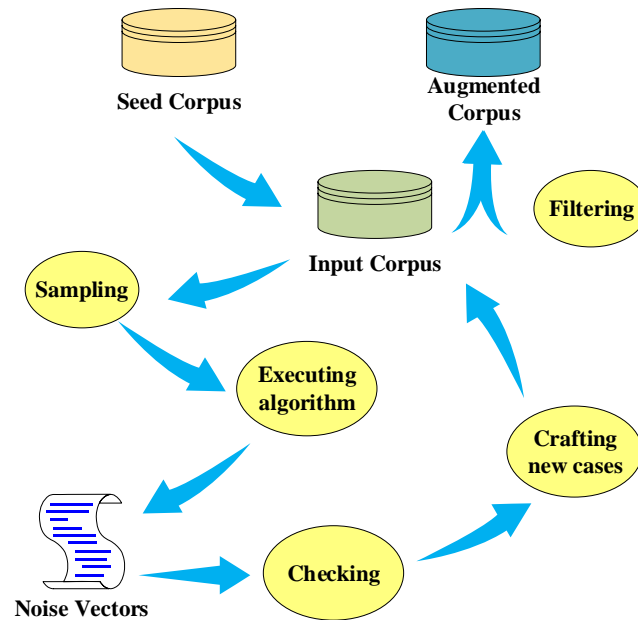


Figure 3. The overall framework of the method

Sampling procedure: The process of sampling the input corpus requires a specific strategy. This strategy is generally heuristic. In our algorithm, the input corpus is sampled according to the input domain data distribution. For example, the samples in the MNIST dataset have ten categories, and the quantities of samples in all categories are approximately equal. Therefore, we randomly pick samples from each category and ensure the samples are in equal amounts. This sampling

method can confirm that the generated new test case corpus is roughly subject to the distribution of the input domain, and the quality of the test case corpus is guaranteed to some extent.

Procedure 1 Sampling Procedure

Input: Input corpus S , case number N .

Output: Sampled Corpus S' .

```

1:  $S' \leftarrow \{\}$ 
2: for An arbitrary category  $S_i$  in  $S$  do
3:   while  $N > 0$  do
4:      $t \leftarrow \text{RandomPick}(S_i)$ 
5:     if  $t$  not in  $S'$  then
6:        $t$  is added to  $S'$ 
7:        $N \leftarrow N - 1$ 
8:     end if
9:   end while
10: end for
11: return  $S'$ 

```

Executing algorithm procedure: The executing algorithm procedure is to run the adversarial example generation algorithm to acquire noise vectors. However, there are some efficiency problems in the execution of the DeepFool algorithm. Generally, the noise vectors obtained by executing the DeepFool algorithm are tiny. Especially if the seed test case itself is far away from the classification boundary, such slight perturbation may not dramatically change the behaviors of the models under test. At the same time, within a certain test time, the generated test cases cannot approach the classification boundary at all. It deviates from our original intention. In order to solve this problem, we introduce a "step size" parameter n to control the executing number of the DeepFool algorithm. This means that the noise vector obtained is the accumulation of the noise vectors after executing the DeepFool algorithm n times. After the DeepFool algorithm is executed once, the sample moves slightly closer to the classification boundary. Owing to the introduction of the step size parameter, we do not need to calculate and check each slight perturbation but rather pay attention to the accumulation of perturbation. This will save computational resources and improve efficiency.

Procedure 2 Executing Algorithm Procedure

Input: Image x , classifier f , step size n .

Output: Perturbation \hat{r} .

```

1: while  $n > 0$  do
2:    $\hat{r} \leftarrow \hat{r} + \text{DeepFool}(x, f)$ 
3:    $n \leftarrow n - 1$ 
4: end while
5: return  $\hat{r}$ 

```

Checking procedure: The process of checking the noise vectors determines whether the noise vectors are reserved for subsequent synthesis of new test cases. In the method implemented in this paper, we use the deformation degree of the seed use case to the new test case as the inspection standard. Papernot et al. proposed in [12] that as long as the deformation degree is less than 14.29%, human eyes can still correctly identify images and classify them. We apply this metric to our algorithm. In the checking procedure, we calculate the deformation degree of every noise vector and judge whether it overpasses the threshold or not. After this procedure, a zero-one vector can be generated, which determines whether the corresponding noise vector should be reserved.

Procedure 3 Checking Procedure

Input: Noise vectors R , deformation degree threshold c .

Output: True or False.

```

1: for  $r$  in  $R$  do
2:    $c' \leftarrow \text{CalculateDeformationDegree}(r)$ 
3:   if  $c' < c$  then
4:     return True
5:   else
6:     return False
7:   end if
8: end for

```

Filtering procedure: The filtering process involves reducing the unnecessary waste of test resources. Although the input corpus is sampled and the "step size" parameter is introduced, a vast number of test cases will be added. Many of these

test cases are redundant. If not processed and directly used to test the DL software, considerable test resources will be wasted. Therefore, we have applied a filtering process to the framework. The test cases are screened out based on the time-series attributes of each test case. While maintaining the input corpus, we use the attribute to indicate how many times of mutation has been carried out in each test case. Initially, all seed use cases in the input corpus have attribute values of 0. When a new test case is added to the input corpus, the attribute value will add one to the original test case's attribute value. The filtering strategy is to randomly pick samples from each of the categories proportionally according to the magnitude of attribute values.

Procedure 4 Filtering Procedure

Input: Input corpus S , case number N_S , pick rate η .

Output: Augmented corpus S_A .

- 1: $S_A \leftarrow \{\}$
 - 2: $ts_m \leftarrow \text{median of time-series attributes in } S$
 - 3: $N_0 \leftarrow N_S \cdot (1 - \eta)$
 - 4: $N_1 \leftarrow N_S \cdot \eta$
 - 5: $S_0 \leftarrow \text{the subset in which the attributes are less than } ts_m$
 - 6: $S_1 \leftarrow \text{the subset in which the attributes are greater than } ts_m$
 - 7: $S_A \leftarrow (\text{RandomPick}(S_0, N_0) \cup (\text{RandomPick}(S_1, N_1)))$
 - 8: **return** S_A
-

4. Experiments

4.1. Dataset and DNN Models

In order to test the effectiveness of the proposed algorithm, we implement the proposed method and conduct experiments on some DNNs and datasets.

Dataset: We select an available public dataset, MNIST, as the experimental dataset. The MNIST dataset is a handwritten digital dataset, containing 70,000 pieces of data. 60,000 pieces of data are used to train the models, and 10,000 are used to evaluate the performance of the models. Each piece of data in the dataset is a 28*28*1 grayscale image. The MNIST dataset is widely used in the training and evaluation tasks of image recognition DNN models because it is classic, open, and reliable.

DNN Models: We used three classic LeNet family DNN models (LeNet-1, LeNet-4, and LeNet-5 [30]) for analysis. The LeNet models are typical convolutional neural networks in which the main key components of the structure are the convolutional layer, pooling layer, activation function, full dense layer, and output layer. The LeNet family DNN models all have the above structure, and only the number of layers and convolution kernels are different. The structures of the three models LeNet-1, LeNet-4, and LeNet-5 are shown in the following table.

Table 1. The structures of subject DNN models

<i>DNN Model</i>	<i>#Neuron</i>	<i>#Layer</i>
LeNet-1	52	7
LeNet-4	148	8
LeNet-5	268	9

4.2. Evaluation Criteria

In order to evaluate the quality of the generated test set, we use a useful coverage metric that was proposed by Odena and Goodfellow in [26]. We will briefly introduce this coverage metric.

The DNN accepts input, and all the activation values of the neurons constitute an activation value vector av . At this time, we believe that the DNN is in a "state", which can be characterized by the activation value vector av . In the following description, we refer to the activation value vector as the state of DNN. For ease of understanding, we give the following definition: the state distance ds represents the gap between two states of the DNN. The computation formula is:

$$ds_{12} = ||av_1 - av_2||_n \quad (7)$$

Where n represents the n norm (we usually use the Euclidean distance, that is, $n = 2$), the state set S_{ns} represents a collection of all the DNN states that have been discovered so far, and the minimal distance mds_x represents the shortest distance from state av_x to S_{ns} . It can be calculated as follows:

$$mds_x = \min\{\|av_x - av\|_n, \forall av \in S_{ns}\} \quad (8)$$

The distance threshold L is used to evaluate whether the two states of the DNN are close enough. For a new state av_{new} , if $mds_{new} \geq L$, it is considered that the test case that makes the DNN in the state av_{new} reaches a new coverage. Otherwise, it is believed that the test case does not reach a new coverage. The distance threshold is always an empirical value. Odena and Goodfellow stated in their paper that it is usually not necessary to obtain the activation values of all neurons. Through empirical tuning, they found that it was often possible to achieve good results by tracking only the logits or the layer before the logits.

4.3. Procedures and Results

We train the DNN models with data in MNIST. Drawing the *loss curves* and the *accuracy curves*, we can see from Figure 4 that after 60 epochs of training, the accuracy and loss value of the DNN models are optimal. Continuing to increase the number of training epochs cannot improve the performance of the LeNet models. Finally, we train the LeNet-1, LeNet-4, and LeNet-5 DNN models for 100 epochs, and the performance of the three models obtained is shown in Table 2.

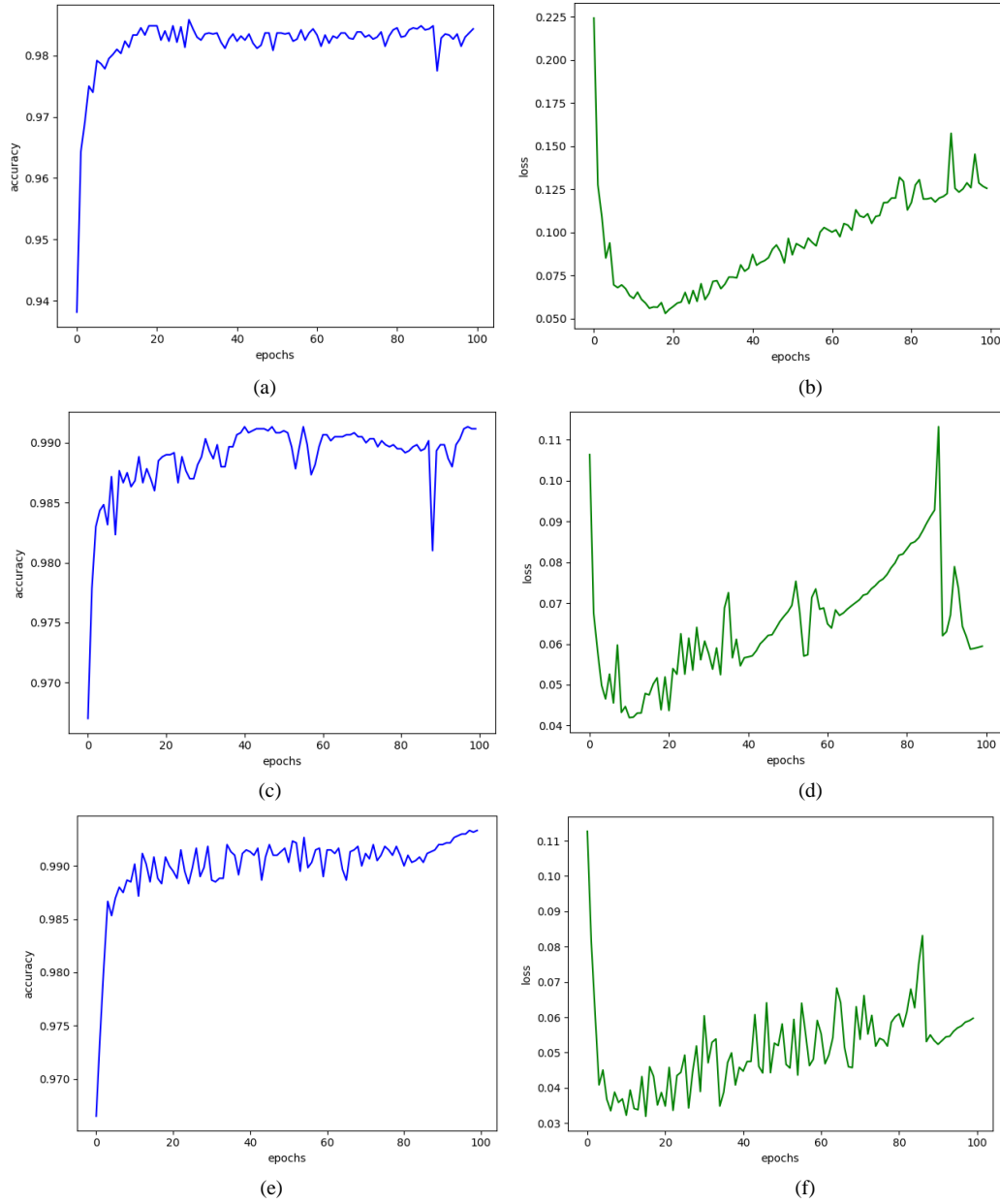


Figure 4. The training performance of the LeNet models: (a) and (b) show the accuracy curve and loss curve of LeNet-1; (c) and (d) show the accuracy curve and loss curve of LeNet-4; (e) and (f) show the accuracy curve and loss curve of LeNet-5

It can be seen that the classification accuracies of the trained models are over 0.985, which meets our expectations. From the 10,000 test samples, we randomly select 200 samples from each category, totaling 2,000 samples, as the seed corpus. After 50% sampling, we obtain 1,000 seed samples. We set the "step size" parameter of the DeepFool algorithm to $n=3, 5$, and 10. During the final filtering procedure, we screen out 4,000 test cases. 20% of required test cases (the 4,000 test cases) comes from test cases with time-series attributes less than the median value, and 80% comes from test cases greater than the median value (we believe that test cases with more obvious changes are closer to the classification boundary and are more likely to find errors, which is consistent with the basis of our algorithm).

Table 2. DNN models and their training performance

<i>Model</i>	<i>Epoch</i>	<i>Train Loss</i>	<i>Train Acc.</i>	<i>Test Acc.</i>
LeNet-1	100	0.125	0.987	0.985
LeNet-4	100	0.065	0.995	0.994
LeNet-5	100	0.063	0.997	0.995

We use the coverage metric based on the states of the DNN to evaluate our obtained test set. It needs to be stated that although the rationality of the current various test coverage criteria is controversial, the existing experiments show that they have certain guiding significance. Therefore, using the test coverage criteria for DL software to evaluate our test suite does not contradict the disadvantages of the test coverage criteria mentioned in the previous section. The results of the experiment are shown in Table 3.

Table 3. NGTC for original set and mutated set

<i>DNN Model</i>	<i>Step Size</i>	<i>NGTC^a for Original Set</i>	<i>NGTC for Mutated Set</i>	<i>Difference</i>
LeNet-1	$n = 3$	926	1924	998
	$n = 5$	926	1943	1017
	$n = 10$	926	1928	1002
LeNet-4	$n = 3$	861	1719	858
	$n = 5$	861	1757	896
	$n = 10$	861	1750	889
LeNet-5	$n = 3$	781	1455	674
	$n = 5$	781	1502	721
	$n = 10$	781	1486	705

^a: NGTC refers to number of good test cases.

We call the test case a *good test case* if it makes the DNN produce a new coverage. As shown in Figure 5, several good test cases are picked from the mutated set. The original set consists of the original images picked from the MNIST dataset, and the mutated set is made up of the original images and their mutated images. It can be seen from the data in the table that many good test cases have been generated because the mutated set achieves more new coverages than the original set. Thus, our method can generate high-quality test sets to a certain extent and can effectively expand the test case sets when the test data is insufficient. The adjustment of parameters during the experiment may cause significant differences in results. Some critical parameters in this experiment, such as the values of "step size" and the distance threshold, are empirical values. We need to pay more attention to the selection of their values in further studies. In future work, the values of these parameters will be studied to optimize the method proposed in this paper.



Figure 5. Several good test cases selected from the mutated set

5. Conclusions

We have proposed a test case generation technique based on an adversarial example generation algorithm for image classification DNNs. It can effectively solve the problem of the augmentation of the test set. We have explained the inspiration that the failure of traditional software often occurs in extreme cases. We have also proposed the implementation framework of the method and carried out a set of simple experiments on the MNIST dataset and LeNet models. The experimental results show that the method is useful for the proposed problems. Current research on the test of DL software seems to have entered a bottleneck. The theoretical basis of the proposed methods cannot be proved and explained, and the practicality of the proposed algorithms is relatively poor. Based on an adversarial sample generation algorithm, we have introduced a feasible method for generating test cases. This method also has defects, but it can be combined with the CGF methods. Both our methods and the CGF methods make pixel-level modifications to the images, and our future work will consider the possibility of combining our method with image transformation methods.

Acknowledgements

This research is supported by the National Key R&D Program of China (No. 2018YFB1403400).

References

1. G. Marcus, "DL: A Critical Appraisal," arXiv Preprint arXiv:1801.00631, 2018
2. G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, et al., "DNNs for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," *IEEE Signal Processing Magazine*, Vol. 29, No. 6, pp. 82-97, 2012
3. D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-Column DNNs for Image Classification," in *Proceedings of 2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3642-3649, 2012
4. Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, et al., "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation," arXiv preprint arXiv:1609.08144, 2016
5. A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, Vol. 1, pp. 1097-1105, 2012
6. M. Wang and W. Deng, "Deep Face Recognition: A Survey," arXiv e-print arXiv:1804.06655, April 2018
7. M. Pritt and G. Chern, "Satellite Image Classification with DL," in *Proceedings of 2017 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, IEEE Computer Society, 2017
8. V. Gulshan, L. Peng, M. Coram, M. C. Stumpe, D. Wu, A. Narayanaswamy, et al., "Development and Validation of a DL Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs," *Jama*, Vol. 316, No. 22, pp. 2402-2410, 2016
9. M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, et al., "End to End Learning for Self-Driving Cars," arXiv e-print arXiv:1312.6199, April 2016
10. C. Szegedy, W. Zaremba, and I. Sutskever, "Intriguing Properties of Neural Networks," arXiv e-print arXiv:1604.07316, 2013
11. I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," in *Proceedings of International Conference on Learning Representations*, 2015
12. N. Papernot, P. McDaniel, and S. Jha, "The Limitations of DL in Adversarial Settings," in *Proceedings of 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 372-387, Saarbrücken, Germany, March 2016
13. S. Baluja and I. Fischer, "Adversarial Transformation Networks: Learning to Generate Adversarial Examples," arXiv e-print arXiv:1703.09387, March 2017
14. L. Ma, F. J. Xu, M. Xue, Q. Hu, S. Chen, B. Li, et al., "Secure Deep Learning Engineering: A Software Quality Assurance Perspective," arXiv e-print arXiv:1810.04538, October 2018
15. Z. C. Lipton, "The Mythos of Model Interpretability," *Queue - Machine Learning*, Vol. 16, No. 3, pp. 30, June 2018
16. S. M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: A Simple and Accurate Method to Fool DNNs," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2574-2582, 2015
17. Y. LeCun, C. Cortes, and C. J. C. Burges, "MNIST, a Database of Handwritten Digits," (<http://yann.lecun.com/exdb/mnist/>, accessed 1998)
18. K. Serebryany, "Libfuzzer, A library for Coverage-Guided Fuzz Testing," (<http://lvm.org/docs/LibFuzzer.html>, accessed 2016)
19. H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: A Versatile, Configurable, Extensible Fuzzing Framework," in *Proceedings of The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, November 2018
20. K. X. Pei, Y. Z. Cao, J. F. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1-18, Shanghai, China, 2017
21. L. Ma, F. J. Xu, F. Y. Zhang, J. Y. Sun, M. H. Xue, B. Li, et al., "DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 120-131, Montpellier, France, 2018
22. Y. Sun, X. Huang, and D. Kroening, "Testing Deep Neural Networks," arXiv e-print arXiv:1803.04792, March 2018
23. K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierison, "A Practical Tutorial on Modified Condition/Decision Coverage," NASA Langley Technical Report Server, 2001
24. Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 303-314, Gothenburg, Sweden, 2018
25. O. Russakovsky, J. Deng, and H. Su, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, Vol. 115, No. 3, pp. 211-252, 2014
26. A. Odena and I. Goodfellow, "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing," arXiv e-print arXiv:1807.10875, July 2018
27. X. Xie, L. Ma, F. Juefei-Xu, H. Chen, M. Xue, B. Li, et al., "DeepHunter: Hunting DNN Defects via Coverage-Guided Fuzzing," arXiv e-print arXiv:1809.01266, September 2018
28. Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg, Sweden, May 27 - June 3, 2018
29. M. Zhang, Y. Zhang, and L. Zhang, "DeepRoad: GAN-based Metamorphic Autonomous Driving System Testing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 132-142, Montpellier, France, 2018
30. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278-2324, 1998