Implementation & Improvement of Deepxplore

1. Introduction

In this section we specify the paper we chose and the reason why we chose this paper. Section 1.1 specifies the paper selection. Section 1.2 illustrates the reason of selecting the paper by pointing out why this paper is important.

1.1 Paper Selection

The selected paper is "Deepxplore: Automated whitebox testing of deep learning systems" published in *Symposium on Operating Systems Principles*(SOSP) *2017.* SOSP is considered to be one of the top conference.

For full paper please check :
https://dl.acm.org/doi/abs/10.1145/3132747.3132785?casa_token=UFCXP8YTVy4AAAAA:QlIUH5syt08DpvQ68xmAWLJniD9CfqQTLrPJGcp1qApaIkTHN3RRm3ZLByph6H2f9QexOVsfHI8efA

1.2 Reason of Selection of the Paper

Deep Neural Networks (DNNs) have been widely implemented in safety- and security-critical fields, where the robustness of the system is of great significance, especially for corner case inputs. Traditionally, a DNN is tested by feeding manually labeled data, which is not only labor-consuming, but also unable to simulate statistically rare case inputs.

In this paper, the author proposes, implements and evaluates the first whitebox framework for systematically testing DNNs. The author is the first one who proposes the concep of neuron coverage, analogous to code coverage in traditional software testing, which is used to reveal to what extent a DNN system is tested. This paper also leverages differential testing technique to solve the oracle problem and to identify corner cases in testing a DNN system.

Finally the proposed testing framework Deepxplore successfully generates thousands of corner case test inputs that trigger the system to behave differently. This paper further demonstrates that the generated test inputs can facilitate to improve the system's accuracy.

2. Terminology

In this section we introduce some basic terminology.

2.1 Differential Testing

There exists an oracle problem in software testing that is "given an input for a system, the challenge of distinguishing the corresponding desired, correct behaviour from potentially incorrect behavior is called the 'test oracle problem'"[1]. To be more intuitive, the oracle problem is " when we already know what is the correct and incorrect bahaviors of a system, we do not need such system". However, differential testing, as a software testing technique, can be leveraged as corss-referencing oracles. It can also be used to define the corner case inputs.

The differential testing works as if a given test input can trigger two or more systems with the same purpose (e.g., two image classifiers on the same training set but with different structures) to behave differently, we consider it as corner case. By implementing such testing technique, the corner case is well defined and the oracle problem is solved. **Figure 1** demonstrates the princle of differential testing in image classification.

Therefore one of the objectives of the framework is to generate test inputs that is "slightly" mutated and triggers two (or more) DNNs to behave differently.

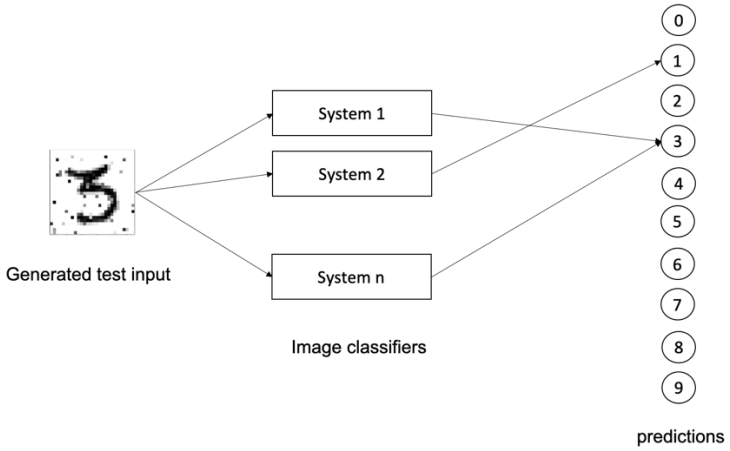


Figure 1. A generated test input triggers systems to make different predictions. Then it is considered to be a corner case.

2.2 Neuron Coverage

The neuron coverage is a core concept in this paper. The neuron coverage is the ratio of the number of activated neurons and the total number neurons in the DNN with a given test input. More formally, the neuron coverage is defined as:

$$NCov(T, x) = \frac{|\{n|\forall x \in T, out(n, t) > t\}|}{|N|},$$

where T indicates the set of neurons in a DNN and out(n,x) is the output of neuron n with a given test input x.

On high level, neuron coverage is analogous to code coverage in traditional software testing. It is an empirical metric for measuring the amount of code a test input covers the system. However, the author argues that traditional code coverage is no longer useful duo to the statistic nature of a DNN. Even a simple test input can reach 100% of code coverage. The author believes that since the DNN is composed of layers of neurons, it is acceptable to take the value of each neuron into consideration.

2.3 Joint-optimization Function

With the divergence (we discussed in section 2.2) and the neuron coverage (we discussed in 2.3), the author thinks we can form a joint-optimization function as :

$$Obj = \lambda 1 Obj1(x) + \lambda 2 Obj2(x), \quad (1)$$

where Obj1 indicates the divergence of the DNN systems with same purpose and Obj2 indicates the neuron coverage. The author argues that by simply implementing gradient descent, we can mutate the seed image that satisfies equation (1).

However, we argue that there are some disadvantages in optimizing the function (1) through gradient descent:

- It may fall into local optimization, which means that the generated test input is not truly optimal solution.
- The objective function is actually a multi-objective optimization problem. However combining multi objectives into one objective function by assigning weights does not “truly” solve the problem. The generated test input is merely an approximation.

Therefore, we implment the generation process by using multi-optimization algorithms.

3. Methodology

3.1 Workflow

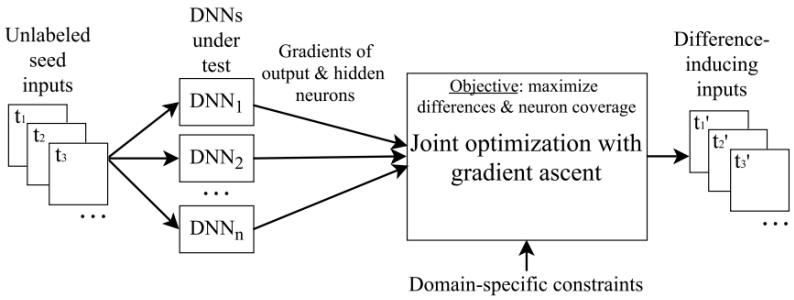


Figure 2. Workflow.

Figure 2 demonstrates the workflow of the generation process. Firstly, we select a seed test input from the test set. Then the seed is fed into the DNN systems to compute the corresponding optimization function. Afterwards, the optimization solver will mutate the seed test input through gradient descent. Finally, the new test input that maximize the objective function is generated.

4. Implementations

4.1 Data set

MNIST [2] is a large handwritten digit dataset containing 28x28 pixel images with class labels from 0 to 9 The dataset includes 60, 000 training samples and 10, 000 testing samples.

4.2 Improved Generation Process

As discussed in **Section 2.3,** the author optimized the function (1) by using gradient descent. We improved the generation process by implementing evolutonary algorithms (EAs) to resolve such multi-objective optimization.

Figure 4 demonstrates the workflow of our implementation. The description gos:

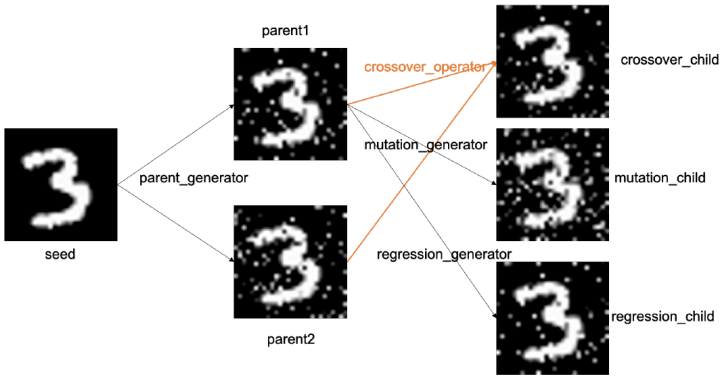
Step 1. We mutate the seed by randomly changing several pixil values;

Step 2. We generate offspring images by genetic operators;

Step 3. We select the optimal solutions from the combined set;

We iterate the above steps for several time.

The genetic operator works as shown in figure 5.



4.3 Aogorithms

Our algorithms are shown in the following algorithms. Threshold,

Algorithm 11 Algorithm for generating test inputs for a DL system

Input: 1. *seed* the original test inputs; 2. *num_parents* the size of the set of parent solutions; 3. *num_mutation* the number of mutations in generating parents; 4. p_m theprobability of mutation; 5. p_c the probability of crossover; 6. $off\_mutation$ mutation number for offspring solutions; 7. $off\_regression$ regression number for offspring solutions; 8. *num_evolutions* the number of evolutions;

Output: The set of optimal generated test inputs.

```
parent ← parent_generator(seed, num_parents, num_mutation)
for evolution ∈ num_evolutions do
    selected_solutions ← ∅
    num_needed_solutions ← len(parents)
    offspring ← offspring_generator (parents, p_c, p_m, 1−p_c−p_m, off_mutation, off_regression)
    solutions ← concatenate(parents, offspring)
    objectives ← mapping(solutions) // mapping solutions into the objective space
    rank = nondominatedsorting(objectives)
    for i ∈ range(rank[−1][1] + 1) do
        candicates_index ← [a[0] for a ∈ rank if a[1] == i]
        num_candidates = len(candidates_index)
        if num_needed_solutions ≥ num_candidates then
            for j ∈ candidates do
                selected_solutions.append(solutions[j])
            end for
        else
            for all j ∈ candidates do
                tournament_solution.append(objectives[j])
            end for
            winners ← cording_distance_sorting (tournament_solution)
            for all i ∈ range(num_needed_solutions) do selected_solutions.append(winners[j])
            end for
        end if
    end forparents ← selected_solutions
    optimal_solutions ← ∅
    rank ← non_dominated_sorting(mapping(parents))
    for all i ∈ range(num_parents) do
        if rank[i] == 0 then
            optimal_solutions.append(parents[i])
        end if
    end for
end for
return optimal_solutions
```

Figure 5. Genetic operators.

4.4 Results

Here, we give some samples we generated in the implementations. Figure 3 shows some samples and their corresponding objective values.

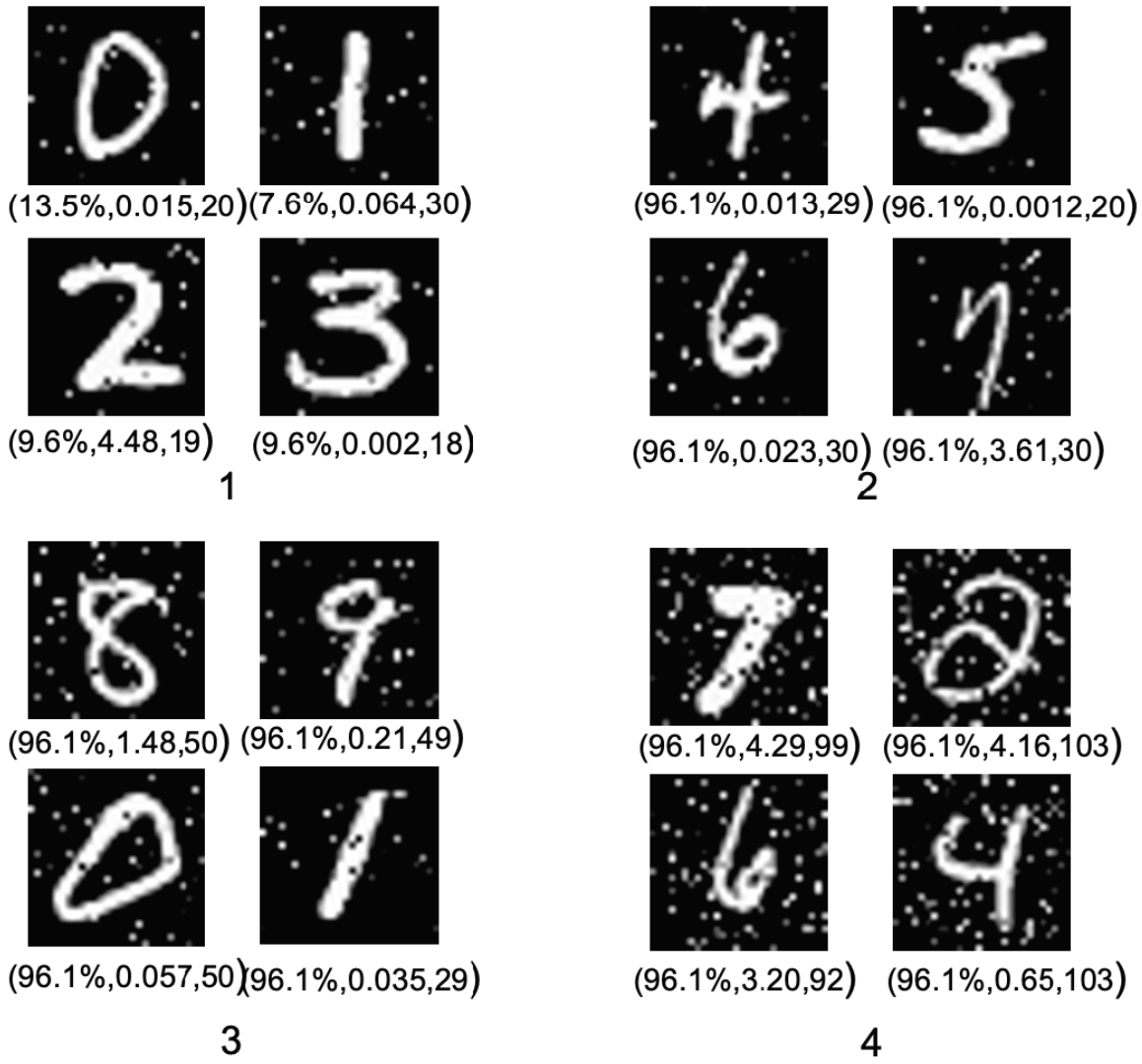


Figure 3. Some Sample Results.

We conducted 4 trials, and the number of images generated in each trial is summerized in table 1. Note that in each tiral we have 200 seeds. We generated 6079 results in total.

Trial num.	parameters	#generated test inputs
1	Threshold = 0.5	1735
	$P_{mutation}$ (parent mutations}	
	#parents = 100	
	$Off_{mutations}$ = 10	
	off $_{regressions}$ = 5	
	#evolutions = 20	
2	Threshold = 0.5	1330
	$P_{mutation}$ (parent mutations}	
	#parents = 100	
	$Off_{mutations}$ = 10	

	$off_{regressions}$ = 5	
	#evolutions = 20	
3	Threshold = 0.5	1568
	$P_{mutation}$ (parent mutations}	
	#parents = 100	
	$Off_{mutations}$ = 10	
	$off_{regressions}$ = 5	
	#evolutions = 20	
4	Threshold = 0.5	1446
	$P_{mutation}$ (parent mutations}	
	#parents = 100	
	$Off_{mutations}$ = 10	
	$off_{regressions}$ = 5	
	#evolutions = 20	

Table 1. Summery of results.

5. Conslusion

In this report, we discuss a paper relating to automatic testing input generation for a DNN system. We illustrated its methodology. We further improved its generation process. We finally derived some results.

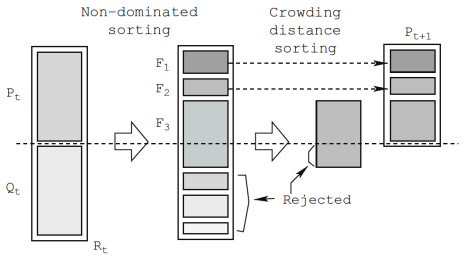


Figure 4. Evolutionary Algorithms

Reference

[1] Barr E T, Harman M, McMinn P, et al. The oracle problem in software testing: A survey[J]. IEEE transactions on software engineering, 2014, 41(5): 507-525.

[2] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 2010. MNIST handwritten digit database. AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist 2 (2010).