Deep Learning Models As Approximate Unit Test Oracles

Anonymous Author(s)

ABSTRACT

Test oracle problem is one of the key open problems in software testing. It remains challenging to automatically verify test cases, i.e., whether they pass or fail. A major reason for the challenge is that formal specifications of systems under test (SUT) are often unavailable, and thus we cannot automatically infer its expected output for given input. In most cases, testers have to manually verify test cases or ignore most test cases that do not lead to runtime errors, e.g., crash. To this end, in this paper, we propose a novel approach to alleviate test oracle problem in unit testing at method level. The approach is based on a series of observations. First, before fully automated extensive unit testing is applied to a method, developers often manually execute and verify typical test cases covering the most common scenarios of the method. Consequently, the method should pass most test cases that are generated in random. The second observation is that advanced deep learning techniques are able to simulate most Java methods if trained with numerous input-output examples even if such examples contain minor noise. Based on such observations, we automatically generate millions of input-output pairs for the method under test (MUT), and train a deep neural network to simulate MUT's behaviors. For numerous new test cases automatically generated for MUT, we present developers with the top k most suspicious test cases with the greatest distance between their actual output and expected output (i.e., network output). Our evaluation results on real-world faulty methods suggest that the proposed approach is accurate. According to its ranking, on 12 of the 18 faulty methods we can find the methods faulty by manually verifying only a single test case for each of the methods.

KEYWORDS

Test Oracle, Unit Test, Deep Learning, Label Noise

1 INTRODUCTION

Automation of test oracles is critical for software testing whereas it remains comparatively less well-solved [2]. Software testing is one of the most time-consuming activities in software development, and thus numerous approaches have been proposed to (partially) automated software testing [4, 7, 20, 21, 24]. Most of such approaches focus on the automated generation of test cases (inputs). However, it remains challenging to automatically verify the generated test cases, i.e., whether and which test cases are revealing defects in the systems, because most test cases generation tools do not generate the expected outputs of the system under test (SUT). As a result, most of the automatically generated and executed test cases are useless and discarded unless they result in runtime errors, e.g., crash. As a result, it remains difficult to automatically identify faulty software entities (that do not crash) although numerous test cases could be generated automatically. This problem is well-known as *test oracle problem* [2] that serves as the bottleneck to test automation.

A large number of novel approaches have been proposed to solve/alleviate the test oracle problem [2]. One of the most intuitive approach is to specify formal specifications of the SUT, and to infer expected output for a given input based on the formal specification [25, 28]. Such oracles are called *specified test oracles*. Although this approach is intuitive and effective, it is challenging to specify the formal specification of software systems [22]. Metamorphic relations [23, 41], the key of metamorphic testing, often serve as an alternative to test oracles. Metamorphic relations do not specify explicitly the expected output of the SUT. Instead, they explicitly specify the relations among multiple runs of the system. Metamorphic testing significantly alleviates the test oracle problem because metamorphic relations are much easier to specified than formal specification of the systems. However, it remains challenging and time-consuming to identify all critical metamorphic relations manually [5].

Another way to test oracle problem is to derive test oracles from other artifacts/information, e.g., informal documents [9, 26, 37] and dynamic executions of the system [12, 38]. The resulting test oracles are called *derived test oracles*. For example, Goffi et al. [13] extracted comments on exceptional behavior from Javadoc, and leveraged natural language processing (NLP) and pattern matching technologies to transform the resulting comments into Java Boolean expressions. However, not all methods are associated with such well-formed comments, and thus its usability is limited. Dynamic execution of software systems could also be exploited to derive test oracles [38]. Vanmali et al. [34] leveraged neural networks to learn from the automatically generated input-output pairs of the original version of the system, and verify test cases on the new version of the system. Zhang et al. [40] proposed an approach to infer polynomial metamorphic relations automatically based on dynamic executions of the correct version of the system. Such approaches [34, 40] are effective in detecting unintended modification of the original functionalities. However, they are unable to verify new test cases designed to test newly added/modified functionalities introduced by the new version of the system. Different from these approaches [34, 40], Shahamiri et al. [30, 31] and Gholami et al. [12] learned from dynamic execution (test cases) of the SUT itself. Shahamiri et al. [30] constructed sample input-output pairs for SUT by manually providing the expected outputs for the randomly generated inputs. Such input-output pairs were leveraged to train a neural network, and the resulting network was used to identify failed test cases for the same SUT. Gholami et al. [12] represented test oracles as a binary classifier, and trained this classifier (a neural network) with failed test cases and passed ones of the same SUT. Notably, this failed/passed test cases were labelled manually. The resulting classifier was finally leveraged to identify failed test cases of the SUT. As a conclusion, none of the existing approaches [12, 30, 31, 34, 40] could learn fully automatically from the execution of the SUT itself without extensive human intervention (i.e., labeling test cases or presenting expected outputs).

In this paper, we propose a novel approach (called *DeepOracle*) to semi-automatically verify whether automatically generated unit tests are revealing defects in methods. The approach is based on a series of observations. First, methods under test (called MUT for short) are often almost correct when fully automated unit testing is requested. Unit test is an indispensable responsibility of software developers [11], and thus they often manually execute and verify some test cases covering the most common scenarios of the method invocation before fully automated extensive testing is applied to the method. Passing such early testing suggests that the methods work properly in most common scenarios [14] and the method should pass most test cases generated in random. For example, if a method contains an *if* statement, the manual early testing should at leat cover both branches of the *if* statement. Only if the method passes all such manual testing, more extensive and resource-consuming fully automated testing could be applied. Otherwise, the automated testing could be broken soon because of obvious defects in the MUT.

The second observation is that advanced deep learning techniques are able to simulate most Java methods if trained with numerous input-output examples even if such large training data contain minor noise. On one side, with the significant advances in deep learning, state-of-the-art deep learning techniques can simulate complex mapping (functionaries) from input to output. On the other side, most methods in object-oriented programming languages are short and their implemented business logic is often simple and straightforward. Besides that, the training data, i.e., input-output of the method, could be generated automatically, which may result in large (if not infinite) training data. Such large training data significantly facilitate model training. Although input-output pairs automatically generated based on almost-correct methods contain a small part of inaccurate data because of the defects in the methods, advanced deep learning techniques are able to handle such minor noise [15, 33].

Based on such observations, we automatically generate millions of input-output pairs for the method under test, and train a deep neural network to simulate the method's behavior. If the training succeeds (i.e., resulting in small loss), the resulting network could be leveraged to verify unit tests on the same method. For a new test case of the method, we compare the *actual output* of method against the output of the trained network (called *estimated output*). The larger the distance between the two outputs (i.e., actual output and estimated output) is, the more suspicious (fault-revealing) the test case is. Consequently, we sort all test cases by the distance between their actual output and estimated output in descending order. From the sorted test cases, we can pick up and manually verify the top k most suspicious test cases: If any of them are fault-revealing (i.e., its actual output is different from expected), the method under test is faulty. We evaluate the proposed approach on 18 real-world faulty methods in Defects4J. Our evaluation results suggest that according to its ranking, on 12 of the 18 faulty methods we can find the methods faulty by manually verifying only a single test case for each of the methods.

The paper makes the following contributions:

- A novel deep learning based approach to validate whether test cases are revealing faults in methods. To the best of our knowledge, it is the first approach to inferring test oracles from dynamic executions of the system under test with automatically generated training data. In contrast, existing approaches request manual labelling of training data or manual construction of expected outputs.
- An evaluation of the proposed approach on real-world methods in Defects4J. Evaluation results suggest that the proposed approach is accurate.

2 RELATED WORK

Test oracle problem is well-known and has been extensively studied. In-depth analysis on the study in this filed is referred to the survey paper by Barr et al [2]. In this section, we only discuss such work that is closely related to the proposed approach.

2.1 Deriving Test Oracles Based on Dynamic Execution

Vanmali et al. [34] leveraged neural network to learn from the input-output of the original version of the system, and verify test cases on the new version or mutated version of the system. The rationale of the proposed approach is that neural networks have the capability to simulate software systems. Shahamiri et al. [30] proposed an approach to automatically generate test oracles by learning from dynamic execution of SUT (software under test) itself. The key rationale of the approach is that we may train a machine-learning model to simulate the SUT with sample input-output, and the resulting model could be leveraged as test oracle to verify new test cases. To this end, they leveraged a Multilayered Perceptron, a special category of neural network, to model the functionality of the SUT. To train the network, they manually constructed training data: sample input-output of the SUT. Notably, the inputs could be generated automatically according to the domain of input variables. However, the expected outputs could not be generated automatically by the SUT because we cannot take the assumption that the SUT is bug-free. Later, they further improved the approach by leveraging multi neural networks to simulate more complex system with multiple outputs [31]. Our approach is similar to such approach in that all such approaches simulate the SUT with input-output of the system. However, our approach differs from theirs in that we do not request expected outputs. Notably, it is challenging (or time-consuming) to figure out the expected outputs even for sample test cases only.

Gholami et al. [12] leveraged Multilayered Perceptron to classify (verify) test cases. The rationale of the approach is that if we can provide a set of failed and passed test cases, we may train a neural network based classifier to classify new test cases. Different from those by Vanmali et al. [34] and Shahamiri et al. [30], this approach does not request output of the system. Consequently, this approach is especially useful for such systems (e.g., embedded systems) whose outputs are implicit or difficult to capture. However, it remains challenging and time-consuming to label the test cases manually.

Braga et al. [3] exploited neural network to generate test oracles for web applications. It captures the log information of the system. Assuming that the web application works properly, each of the test cases (logs) could be taken as a positive item (passed test case) as it, or be taken as a negative item (failed test cases) by manually or automatically modifying the logs associated with the test case.

In such a way, they created a large set of labelled dataset to train and evaluate a neural network based classifier. Notably, in the evaluation, they applied k-fold evaluation, and the labelled dataset is randomly partitioned into training data and testing data.

We conclude based on the preceding analysis that it is promising to derive test oracles based on dynamic execution of SUT. However, none of such approaches could learn fully automatically from the execution of SUT without human intervention (i.e., labeling test cases or presenting expected outputs).

2.2 Detection of Invariants

Invariants of programs specify properties that hold at every execution of the system under test. For example, for a method *sum(a,b)* that returns the sum (noted as c) of two positive integers a and b, the invariants $a > 0$ and $b > 0$ specify the pre-conditions of the method whereas $c = a + b$ and $c \geq 2$ specify the post-conditions of the method. Both invariants and metamorphic relations specify properties of the system under test, and violating any of them could suggest illegal input or defects of the system. Invariants differ from metamorphic relations in that invariants specifies properties that much hold at a single run of the systems whereas metaphoric relations specify the relations among multiple runs of the systems. Daikon is one of the most well-known approaches to detection of invariants [10]. It instruments a set of predefined test cases, runs the test cases, and collects the logs (e.g., inputs and outputs of each public methods). Based on the logs, Daikon validates a set of predefined potential invariants, and generates such invariants that hold on each run of the test cases. Such invariants could be leveraged to design or validate test cases [36].

The proposed approach infers the map from input to output of the methods under test, and the resulting map could be taken as a special kind of invariants of the methods. However, the proposed approach differs from such invariant detection approaches in that the latter requests manually predefined patterns of potential invariants. Consequently, such approaches often miss more advanced and less common properties of the methods under test, e.g., complex maps from input to output because such maps are often complex and rarely follow specific patterns.

2.3 Metamorphic Relations

Metamorphic relations could serve as effective test oracles as well [6]. Metamorphic relations are necessary properties of the target function in relation to multiple inputs and their expected outputs. Manual construction of such metamorphic relations could be tedious and time-consuming because the identification often involves much human intelligence for analyzing specifications and finding necessary properties of the system under test. Consequently, a few approaches have been proposed to derive such metamorphic relations automatically. For example, Liu et al. [23] proposed an automated approach to systematically constructing new metamorphic relations based on existing metamorphic relations that are identified manually or automatically. Notably, the approach compose several different metamorphic relations into one single metamorphic relation to improve the cost-effectiveness of metamorphic testing: We may use fewer metamorphic relations (i.e., the resulting new relations) to reveal most failures that are detected when all original metamorphic relations are used. The approach is not to derive brand new metamorphic relations. Zhang et al. [40] proposed an search-based approach to infer metamorphic relations automatically from scratch based on dynamic executions of SUT (software under test). The approach is confined to a special category of metamorphic relations, i.e., polynomial metamorphic relations. Such relations could be unified as a polynomial equation with parameters. For a given SUT, the approach leverages search-based techniques to find values of the parameters so that the equation holds for all executions of the SUT. Assigning the resulting values to the parameters results in metamorphic relations for the SUT. Their evaluation results suggest that this approach can derive metamorphic relations from correct version of the system and such relations could be leveraged to detect mutants.

Kanewala and Bieman [17] predicted whether given metamorphic relations hold for the system under test by an machine learning (i.e., SVM) based classifier. Input of the classifier is the control flow graph (CFG) of the SUT, and boolean output indicates whether a given metamorphic relation hold. Notably, this approach relies on labelled training data that are manually created in their evaluation.

The proposed approach differs from such approach in that they infer metamorphic relations whereas our approach infer test oracles that could be used to verify a single run of the system. Another difference is that such approaches request existing metamorphic relations [23], earlier correct version of the system [40], or manual labelling of training data and predefined metamorphic relations [17]. In contrast, our approach is fully automated and does not depend on such artifacts.

2.4 Test Case Selection and Prioritization

Test case selection (TCS) is to select and execute only a subset of the regression test cases aiming to reduce the cost of regression testing [18]. TCS selects such test cases on version V_{n-1} whose execution may be influenced by the modification (i.e., difference between the current version V_n and the original version V_{n-1}). Other test cases are not influenced by the modification, and thus it unlikely for them to reveal new defects caused by the modification. Consequently, if test cases selection techniques are properly applied, TSC may significantly reduce the number of executed test cases whereas the fault-revealing capability of the regression testing is not significantly reduced [8].

Test case prioritization (TCP) is to schedule execution order of test cases to expose errors earlier [19]. TCP executes the most beneficial test cases first, expecting such test cases to reveal most defeats in the system under test. For example, we may execute such test cases first that could lead to the maximal increase in the coverage of the regression testing [35]. The key rationale is that higher coverage of the regression testing often results in greater capability in fault detection. Detailed introduction to test case prioritization is referred to the systematic review [19], and detailed introduction to test case selection is referred to the systematic review [18] and empirical study [8].

The major benefits of TCP and TCS include shorter regression testing time, and fewer computing resource allocated to regression

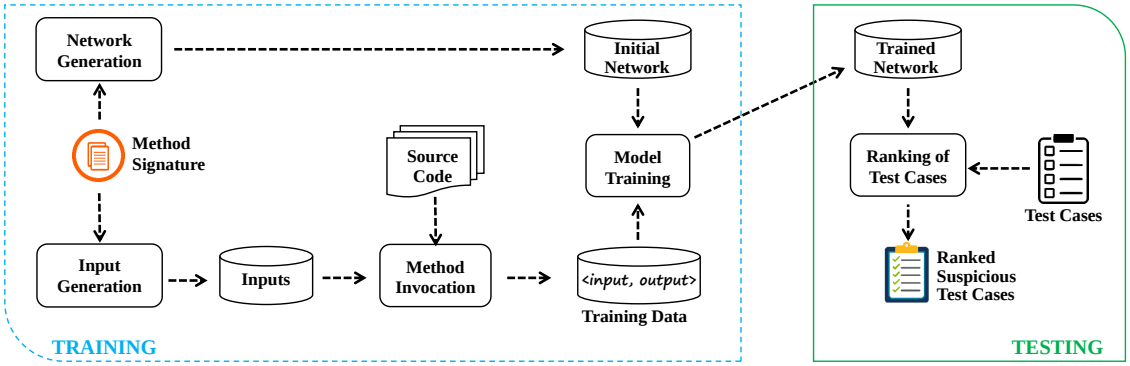

Figure 1: Overview of *DeepOracle*

testing. However, if testers have to manually verify test cases because of the lack of automated test oracles, TCP and TCS may also significantly reduce the effort in manual verification of test cases. TCS reduces the number of executed test cases and thus reduce the effort in manual verification of test cases. TCP executes and verifies the most beneficial test cases first and other test cases could be ignored/discarded if the preceding test cases successfully expose bugs. As a result, TCP often reduces the effort in manual verification of executed test cases.

Our approach is similar to TCP and TCS in that all of them could be leveraged to reduce the effort in verifying test cases. Our approach differs from TCP and TCS in that the latter is often confined to regression testing. Another difference is that our approach exploits the input-output of test cases whereas TCP and TCS depend on the traces (paths) of the test case executions.

3 APPROACH

An overview of the proposed approach (*DeepOracle*) is presented in Fig. 1. For a given method under test (noted as *MUT*), DeepOracle works as follows:

- First, DeepOracle randomly generates inputs for the method.
- Second, DeepOracle invokes the method with the inputs generated in the preceding step, and associates the output with the corresponding input. Such inputs and outputs constitute test cases for the method.
- Third, Based on the signature of the method, DeepOracle generates an initial neural network. The generation is based on predefined patterns, and the initialization of the patterns (especially the input layer and output layer) depends on the inputs and outputs of the method.
- Fourth, DeepOracle trains the generated neural network with the input and outputs generated in the second step. If the training cannot result in a stable and accurate network, DeepOracle fails and terminates the testing on the method. Otherwise, it goes to the next step.
- Fifth, DeepOracle generates new test cases in the same way as it generates training data on the first two steps, feeds the inputs in the test cases into the trained neural network, and compares the outputs in the test cases (called *actual output*) against the output of the neural network (called *estimated output*).
- Finally, DeepOracle ranks the test cases according to the distance (in descending order) between their actual outputs and estimated outputs, and presents developers with the top k most suspicious test cases. If any of the suspicious test cases reveal defects of the method, developers should fix it and repeat the preceding steps.

Details of the key steps are presented in the following sections.

3.1 Input Generation and Output Collection

Fully automated generation of training data is critical for the proposed approach because it relies on deep neural networks that request a large number of high quality training data. In this section, we explain how such training data are generated automatically.

Suppose the method under test (MUT) is declared as follows:

$$returnType\ MUT(type_1\ p1,\ \ldots,\ type_n\ p_n) \tag{1}$$

Input generation is to generate items as follows:

$$ipt = < v_1,\ \ldots,\ v_n > \tag{2}$$

$$v_i \in type_i \quad i \in [i, n] \tag{3}$$

where *ipt* is a generated input item that is composed of n values, and each of the values falls into the range defined by the corresponding parameter type. Notably, if necessary, developers can manually specify the range that is smaller than the one defined by the data type. For example, parameter "`int age`" in staff management system could be safely confined to [0,150] that is significantly smaller than the range of `int` because currently (and in the near feature) nobody could live for more than 150 years. DeepOracle generates value v_i by randomly picking a value from the domain (range) that is explicitly defined by data type $type_i$ or explicitly specified by developers. Notably, developers may also explicitly specify conditions that should hold on the generated inputs. For example, the second parameter should be greater than the first one (i.e., $v_2 > v_1$). In this case, DeepOracle automatically validates the conditions on generated input items, and discards such items where any of the conditions fails. Manually specified ranges and conditions are not indispensable, but they may significantly speed up the proposed approach by avoiding invalided inputs. DeepOracle allows developers to explicitly specify additional inputs besides the input parameters, e.g., accessed fields. By default, all fields assessed

by the method without initialization are counted in as inputs of the method.

For each of the input item, DeepOracle computes the corresponding (actual) output by running the method (i.e., MUT) and associates the output with the input:

$$tc_i \quad = \quad < ipt_i, act_i > \tag{4}$$

where tc_i is a generated test case containing both input (ipt_i) for the method and actual output (act_i) of the method. DeepOracle allows developers to explicitly specify the actual outputs (e.g., updated fields of the enclosing class and return values of MUT) that should be collected automatically. If the outputs are not explicitly specified by developers, the outputs collected automatically by DeepOracle include the return value of the method, fields updated by this method, and outputs to consoles. As a result, the actual outputs are represented as a sequence:

$$act_i =< opt_{i,1}, \ \ldots \ , opt_{i,m} > \tag{5}$$

3.2 Network Generation and Training

Neural networks request their inputs and outputs being presented as digital vectors. Consequently, before the test case tc in Equation 4 could be fed into the neural network, we have to turn the input and output of the test cases into digital vectors. To this end, non-digital value (strings and characters) in the test cases is transformed into digital representation by representing each character with its corresponding ASCII, and concatenating all such digital ASCIIs as a digital vector. For non-primitive parameters, we iteratively exploit their primitive fields, and represent them as a sequence of primitive fields. Each of the resulting fields is taken as new parameter. Similarly, arrays are decomposed and represented as sequences of primitive parameters that are in turn represented as digital vectors. Because different parameters have significantly different ranges, we normalize the digital values into the same range [0,1] with max-min normalization [ref]: for each value, we subtract the minimum value in the data set and divide by the difference between the maximum value and the minimum value. Consequently, we finally represent the test case $tc_i =< ipt_i, act_i >$ as follows:

$$ipt_i =< dip_{i,1}, \ \ldots \ , dip_{i,k} > \tag{6}$$

$$act_i =< dop_{i,1}, \ \ldots \ , dop_{i,j} > \tag{7}$$

where $dip_{i,l}$ ($l \in [1,k]$) represents a digital vector for an input parameter and $do_{i,l}$ (($l \in [1,j]$)) represents a vector for an output parameter. Both of them range between zero and one.

According to the generated test cases, DeepOracle generates neural networks automatically. To improve the generality of the employed deep neural network, DeepOracle only leverages the basic Fully Connected Network (called FNN for short) where each neuron in a layer is connected to all neurons in the next layer. The network is composed of one input layer, three hidden layers, and one output layer. The size of the input layer equals to the maximal length of the input vectors, the size of the output layer equals to the maximal length of the output vectors, and the sizes of the hidden layers are empirically set to 256.

DeepOracle trains the automatically generated neural network with the test cases generated in Section 3.1. Notably, training a deep neural network with the same training data for multiple times often results in more accurate network models. Consequently, we keep training the generated network with the generated test cases until the loss of the network is smaller than a predefined threshold β (empirically set to 0.02) or allocated time slots (empirically set to 30 minutes) are exhausted. A side effect of the iterative training is that it may result in over-fitting [1]. To this end, we take the following measures to prevent over-fitting: parameter regularization [39] and dropout [32]. For parameters within the same network, the larger a parameter is, the greater impact it has on the performance of the network. Parameter regularization [39] is to add a penalty term to large parameters to prevent them from dominating the network. Dropout [32] is a strategy to discard neurons in hidden layers randomly during training, which prevents neurons from co-adapting too much. Both parameter regularization and dropout have been proved effective in preventing over-fitting of networks[ref][ref].

The default loss function of the network computes the mean square error (MSE) between actual outputs (vectors) of the network and the expected outputs (vectors). Although MSE has been widely used to measure the performance of neural networks, it does not exactly measure the distance between the actual outputs (Java data types) and the expected outputs (Java data types) of the method. To this end, we define the following loss function:

$$loss = \frac{1}{n}\sum_{i=1}^{n}(Distance(act_i, \hat{act_i}))^2 \tag{8}$$

$$Distance(act_i, \hat{act_i}) = \frac{1}{m}\sum_{j=1}^{m} D(opt_{i,j}, \hat{opt}_{i,j}) \tag{9}$$

$$D(opt_{i,j}, \hat{opt}_{i,j}) = \begin{cases} \frac{opt_{i,j} - \hat{opt}_{i,j}}{\max\limits_{k\in[1,n]} opt_{k,j} - \min\limits_{k\in[1,n]} opt_{k,j}}, \text{if } opt_{i,j} \text{ is numeric} \\ \\ \frac{ED(opt_{i,j}, \hat{opt}_{i,j})}{|opt_{i,j}| + |\hat{opt}_{i,j}|}, \text{if } opt_{i,j} \text{ is textual} \end{cases} \tag{10}$$

where n is the size of the training data, m is the size of outputs of the method under test. act_i (as defined in Equation 7) and $\hat{act_i}$ are the actual outputs and estimated outputs (by the network) of the method, respectively. $\max\limits_{k\in[1,n]} opt_{k,j}$ and $\min\limits_{k\in[1,n]} opt_{k,j}$ are the maximal and minimal values of the given output parameter, respectively. $ED(opt_{i,j}, \hat{opt}_{i,j})$ is the edit distance [29] between two strings, and $|str|$ is the length of the string str.

Compared against MSE, our loss function has the following advantages. First, our loss function facilitates unified loss-based assessment of different networks. MSE accounts the absolute distance between actual outputs and estimated outputs, and thus the loss is significantly influenced by the ranges of the outputs. Considering that the outputs of different networks differ from each other significantly, our loss function replaces the absolute errors with relative errors. Second, our loss function computes the distance between two strings with edit distance, which is more accurate and explainable than embedding-based vector distances. As specified in the first paragraph of Section 3.2, strings and characters are represented by a sequence of ASCIIs that in turn is taken as a sequence of integers. Consequently, MSE measures the difference between two strings (actual output and estimated output) by computing the difference of two integer sequences. However, the distance between

two ASCIIs is nonsense: Although 'i' and 'h' have close ASCII values, generating 'i' is not more accurate than generating 'j' when 'h' is expected.

If the resulting network does not convergence (i.e., its loss remains no less than β), the deep neural network based simulation of the method under test fails. In this case, DeepOracle gives up, and it refuses to generate any suggest (oracle) for the method. If the resulting network does convergence, the resulting network could be leveraged as approximate test oracles for the unit testing of the method.

3.3 Ranking of Suspicious Test Cases

After running and validating a small number of manually constructed unit test cases, developers may leverage the resulting neural network to validate whether the method under test is faulty. They should automatically generate a large number of test cases by third-party test case generation tools (e.g., EvoSuite) or by DeepOracle (in the same way as training data are generated in Section 3.1). However, such generated test cases are not accompanied with accurate expected outputs to validate the results of the test case executions. Consequently, developers have to manually validate such numerous test cases, which could be tedious and time-consuming.

As an alternative, developers may leverage the neural network trained in Section 3.2 to tell which test cases are defect-revealing. A test case is defect-revealing if its actual output is different from its expected output. DeepOracle computes the suspicious scores of the test cases. The suspicious score of a test case equals to the loss of the network when the test case is fed into the network for testing. DeepOracle ranks suspicious test cases in descending order by suspicious scores, and presents only top k most suspicious ones to developers for manual checking. If any of the manually checked test cases reveals defects within the method, developers should fix the defects and repeat the unit test on the fixed version of the method.

4 EVALUATION

4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: What percentage of Java methods could be simulated accurately by generic deep neural networks?
- **RQ2**: Is the proposed approach accurate in identifying failed test cases?
- **RQ3**: Does the proposed approach outperform existing approaches in identification of failed test cases?
- **RQ4**: Is the proposed approach efficient?

The proposed approach is based on the assumption that a large percentage of methods in object-oriented programming languages could be simulated accurately by simple deep neural networks without significant customization by experts of deep learning techniques. Answering RQ1 would validate this assumption.

RQ2 concerns the performance of the proposed approach, i.e., how well it works in identifying failed test cases. To measure the performance, we leverage the widely used precision and recall.

$$p@k = TP@k/k \quad (11)$$

$$r@k = IBM@k/BM \quad (12)$$

where $p@k$ is the precision when top k test cases are recommended to be validated, $TP@k$ is the number of true positives (i.e., failed test cases that are ranked in top k by the proposed approach) within the recommended ones. $r@k$ is recall when top k test cases are recommended to be validated, and $IBM@k$ is the number of buggy methods that are successfully identified by checking the top k test cases suggested by the proposed approach. BM is the total number of buggy methods involved in the evaluation.

RQ3 concerns the comparison between the proposed approach and existing approaches. We compared the proposed approach against three approaches:

Random-Checking: Randomly pick up k test cases for manual validation.

Coverage-Guided Checking: Pick up the test case that results in the maximal increase of code coverage, and repeat the selection until k test cases are selected for manual validation.

Invariant-Based Checking: Pick up the top k test cases (if there is any) that violate the most invariants.

Coverage-Guided Checking (called *CGC* for short) is widely used to maximize code coverage with the fewest test cases with the assumption that higher coverage of production code often leads to greater chance of finding defects. In the evaluation, CGC maximizes the coverage of statements. Invariant-Based Checking (called *IBC* for short) is based on the assumption that test cases violating identified invariants often reveal defects of the system under test. In the evaluation, we leverage the well-known Daikon to detect invariants automatically because existing evaluations suggest that it is effective and efficient [10]. Random-Checking (called *CC* for short) is selected because it frequently serves as a baseline in related work.

RQ4 measures how long it may take the proposed approach to generate test cases, train deep neural networks, and pick up fault-revealing test cases.

4.2 Subject Projects

The evaluation was conducted on the faulty methods in Defects4J [16]. Defects4J is a widely-used database and extensible framework providing real world defects to enable reproducible studies in software testing research. Defects4J is composed of defects from open source programs. We selected such faulty methods because of the following reasons. First, all of them are publicly available, which facilitates researchers to repeat the evaluation. Second, all of the defects are accompanied with bug reports and manually validated concise patches, which significantly facilitate manual validation of test cases.

Notably, not all of the faulty methods in Defects4J were suitable for the evaluation. The following faulty methods are excluded:

- Methods that are reported as faulty because unexpected exceptions. For such kind of faulty methods, we can distinguish failed test cases from pasted ones by catching the unexpected exceptions. Consequently, there is no need for more advanced (and thus less reliable) approaches for the same task. In total, such methods account for 38.1% of the faulty methods in Defects4J.
- Methods whose defects are not caused by incorrect implementation. Defects could be introduced because of various reasons,

Table 1: Selected Faulty Methods

ID	Method Signature	Size (LOC)	Project	Description Of Defeats
M1	void print(Object)	10	Csv	The method escapes the nullString value unnecessary.
M2	long addYears(int)	5	Time	When the current time is added by zero year, the time should remain untouched.
M3	long addMonths(int)	5	Time	When the current time is added by zero month, the time should remain untouched.
M4	long addWeeks(int)	5	Time	When the current time is added by zero week, the time should remain untouched.
M5	long addDays(int)	5	Time	When the current time is added by zero day, the time should remain untouched.
M6	long add(DurationFieldType,int)	5	Time	When the current time is added by zero millisecond, the time should remain untouched.
M7	MonthDay minusMonths(int)	9	Time	Incorrect return value when the resulting day is February 29
M8	MonthDay plusMonths(int)	9	Time	Incorrect return value when the resulting day is February 29
M9	JsonPointer compile(String)	26	JacksonCore	Fail to distinguish illegal input with leading '0'.
M10	int read(byte[],int,int)	24	Compress	Zero is expected when the third parameter (len) equals to zero.
M11	boolean isSameLocalTime(Calendar, Calendar)	13	Lang	24-hour format is taken as 12-hour format by mistake.
M12	Token nextToken(Token)	52	Csv	Special character $\backslash n$ is not handled correctly.
M13	double pow(double, double)	122	Math	Deviates from Math.pow for negative, finite base values with an exponent.
M14	String sanitize(String)	17	Compress	For strings longer than 255 characters, their sanitized version should have been returned.
M15	int anyInt()	3	Mockito	Argument matcher anyInt() should not match 'null'.
M16	String soundex(String)	25	Codec	A character preceded by two characters that are either H or W, is not encoded, regardless of what the last consonant is.
M17	String doubleMetaphone(String)	102	Codec	Incorrect return value when "G" is followed by "IER" in the input string.
M18	long readBits(int)	28	Compress	Incorrect return value when input parameter greater than 59.

e.g., incorrect implementation, incorrect specifications (requirements), and incorrect interpretation of the specifications. Notably, our approach is designed for developers for unit test, and thus it is intended to identify defects caused by incorrect implementation only. Consequently, a faulty method was excluded from the evaluation if a significant part (more than 40%) of its test cases failed. Such a high ratio of failed test cases suggest that the defect is rather obvious and should be easy to find. A reasonable reason for missing the defect is that the developers misunderstood the requirements and thus took all such test cases as 'pasted'. In total, such methods account for 26.4% of the faulty methods in Defects4J.

- Methods request external files or specially structured code as input or output. For example, in the project *Compress*, the method *getNextZipEntry()* requires a zip file as input. Such methods account for 32.5% of the faulty methods in Defects4J. We excluded such methods from evaluation because the current implementation of the proposed approach could not yet automatically generate such complex inputs.

After the exclusion, 18 faulty methods from 9 projects were selected for the evaluation. Table 1 presents an overview of the selected methods.

4.3 Process

For each of the selected methods, the evaluation of the proposed approach was conducted as follows:

- First, we leveraged DeepOracle to generate test cases (inputs) for the method. In total, RTG generated 1,010,000 test cases for each method. We randomly selected one million of them as training data (noted as *TRD*), and the other ten thousand test cases were taken as testing data (noted as *TSD*).
- Second, the training data *TRD* were fed into DeepOracle. If the loss of the trained network was greater than β, DeepOracle failed because of '*inaccurate simulation*' and the evaluation on this method terminated.
- Third, we validated the resulting network on the testing data *TSD*. A test case (input) failed if and only if the current (faulty) version of the method and its fixed version (publicly available for all faulty methods in Defects4J) generated different output for the same input (test case). Based on this ground truth, we computed the performance (e.g., precision) of DeepOracle.

The evaluation of the baseline approaches, i.e., random-checking, coverage-guided checking, and invariant-based checking, were conducted similarly on the same dataset. For invariant-based checking, we inferred invariants of a faulty method by Daikon with test

cases in the training data (*TRD*), and sorted test cases in the testing dataset (*TSD*) according to the number of violated invariants. Random-checking and coverage-guided checking do not request training, and thus we applied them directly to the testing data set.

4.4 Results and Analysis

4.5 RQ1:Simulation of Faulty Methods

We simulated the selected faulty methods with the networks automatically generated by DeepOracle. Our evaluation results are presented in Table 2. Columns 2-4 present the average, maximal and minimal loss of the networks in training process. The fifth column presents mean square error (MSE) of the network. The last column presents whether the network is converged. A network dose not converge if its average loss remains great (greater then 0.02) regardless of the increasing number of training iterations.

From Table 2, we observe that 15 out of the 18 selected methods (i.e., M1-M15) were simulated accurately. Their average loss (the second column) varies from 0.0001 to 0.0108, and their MSE varies from 0.0001 to 0.0081. Such mall errors suggest that the neural networks are converged and the network-based simulation is highly accurate.

From the table, we also observe that the simulation failed on 3 out of the 18 selected methods (i.e., M16-M18). The average loss of such methods varies from 0.0426 to 0.1321, significantly higher than that of other methods. A possible reason for failing to simulate such methods is that such methods are complex. For example, M17 (*String doubleMetaphone(String)*) is the implementation of the well-known double metaphone search algorithm proposed by Philips [27]. This algorithm is to encode a given string into a double metaphone value and to return the value as a string. Such algorithm is rather complex. The method itself is composed of 102 lines of source code, containing 23 branches within a *while* iteration where most of the branches call different methods outside the method. In total, the method invokes 15 methods from its enclosing class. The whole class (containing more than one thousand lines of source code) does nothing but implementing the double metaphone search algorithm, i.e., supporting the method M17. Such a complex algorithm is difficult to simulated accurately with general fully connected neural networks without in-depth customization.

We conclude based on the preceding analysis that most (83%=15/18) of the faulty methods could be simulated accurately by the proposed template-based simple neural networks, which validates the fundamental assumption of the proposed approach.

4.6 RQ2: The Proposed Approach Is Accurate

To answer RQ2, we applied the proposed approach to the 15 faulty methods whose corresponding neural networks convergence. The evaluation results are presented in Table 3. The last four columns of the table present how many test cases ranked in top K (K=1, 5, 10, and 100, respectively) are fault-revealing. Notably, for each faulty method, we had generated 10,0000 test cases, and the top K test cases were selected by the proposed approach from such large amount of candidates. Consequently, top 1 and top 100 are equivalent to top 0.001%=1/10,0000, and top 0.1%=100/100,000, respectively.

From the table, we make the following observations:

Table 2: Simulation of Faulty Methods

ID	Average Loss	Maximal Loss	Minimal Loss	MSE	Converged
M1	0.0051	0.2500	0.0000	0.0003	Yes
M2	0.0013	0.0061	0.0000	0.0013	Yes
M3	0.0008	0.0168	0.0000	0.0008	Yes
M4	0.0024	0.0357	0.0000	0.0024	Yes
M5	0.0004	0.0141	0.0000	0.0004	Yes
M6	0.0046	0.0523	0.0000	0.0046	Yes
M7	0.0086	0.0310	0.0000	0.0086	Yes
M8	0.0081	0.0279	0.0000	0.0081	Yes
M9	0.0014	0.1600	0.0000	0.0003	Yes
M10	0.0001	0.0002	0.0000	0.0001	Yes
M11	0.0080	0.1736	0.0000	0.0006	Yes
M12	0.0108	0.1975	0.0000	0.0007	Yes
M13	0.0010	0.1207	0.0001	0.0010	Yes
M14	0.0097	0.0524	0.0001	0.0001	Yes
M15	0.0036	0.1406	0.0000	0.0001	Yes
M16	0.0711	0.1600	0.0000	0.0002	No
M17	0.1321	0.2500	0.0000	0.0035	No
M18	0.0426	0.7127	0.0009	0.0426	No

Table 3: Evaluation Results

ID	Error rate	Top 1	Top 5	Top 10	Top 100
M1	0.90%	1	5	10	100
M2	0.98%	1	5	10	100
M3	0.99%	1	5	10	100
M4	1.01%	1	5	10	100
M5	1.00%	1	5	10	100
M6	1.00%	1	5	10	100
M7	6.46%	1	5	10	100
M8	7.01%	1	5	10	100
M9	0.99%	1	5	10	100
M10	2.40%	1	5	10	100
M11	2.92%	1	5	10	100
M12	3.00%	1	5	9	69
M13	2.00%	0	0	0	0
M14	2.51%	0	0	0	0
M15	1.01%	0	0	0	0

- First, on 12 out of the 15 faulty methods (i.e., M1-M12), the top 1 and top 5 accuracy is 100%, suggesting that all of the suggested suspicious test cases are fault-revealing. For the first 11 methods (i.e., M1-M11), all of the suggested top 100 test cases are fault-revealing. Such hight accuracy suggests that developers can find the method under test faulty by checking only a single test case suggested by the proposed approach. Notably, only a small part (2.08% on average) of the test cases were fault-revealing. Consequently, it is challenging to accurately pick up a fault-revealing test cases surrounded by a large amount of passed test cases (that cannot reveal faults). DeepOracle, however, successfully picked up fault-revealing test cases, which significantly facilitates unit testing.

Table 4: Comparison Again Baseline Approaches

	Top 1		Top 5		Top 10		Top 100	
	p	r	p	r	p	r	p	r
Proposed Approach	66.67%	66.67%	66.67%	66.67%	66.11%	66.67%	64.94%	66.67%
Random	5.56%	5.56%	1.11%	5.56%	1.67%	11.11%	1.89%	61.11%
Coverage-based	5.56%	5.56%	4.44%	22.22%	2.22%	22.22%	2.28%	66.67%
Invariant-based	17.64%	17.64%	18.82%	23.53%	18.82%	29.41%	18.94%	64.71%

- Second, on 3 out of the 15 faulty methods (i.e., M13-15), none of the suspicious test cases suggested by DeepOracle is fault-revealing, suggesting the DeepOracle does not work on such methods.

We take the faulty method M10 (read)[ref] to illustrate why DeepOracle works. This method transfers bytes from an input stream to an array (*dest*). The number of bytes to be transferred is specified by parameter *len* whereas parameter *offs* specifies the location (index) where the array *dest* could be used to store the bytes. The number of bytes that have been actually transferred is returned as the return value of the method. When *len* equals 0, the method should not transfer any bytes to the array, and thus it should return zero suggesting that 0 bytes have been transferred. However, the faulty method return -1 by mistake. In the automatically generated training data, fault-revealing test cases (where $len == 0 \wedge ReturnValue == -1$ holds) account for only 2.40% of the test cases whereas $ReturnValue == len$ holds on most of the test cases. As a result, the corresponding neural network learns that the return value should equal to parameter *len*, and fault-revealing test cases where the return value does not equal to *len* is reported as suspicious.

We take the faulty method M14 (*String sanitize(final String s)*) to illustrate why DeepOracle fails on some methods. This method returns a "sanitized" version of the string given by replacing non-printable characters with a question mark and cutting it off when it is longer than 255 chars. However, the faulty version of the method fails to cut off the string, which results in returned strings longer than 255 chars. The resulting neural network learns "the output string should be identical to the input strings". However, it does no learn "the input string should be cut of when it is longer than 255 characters" because for a large part (2.51%) of train data (test cases) the returned string is longer than 255 characters. As a result, the trained network would not suggest the returned long strings as *suspicious* and thus failed to identify fault-revealing test cases.

Based on the preceding analysis, we conclude that the proposed approach is accurate in suggesting fault-revealing test cases.

4.7 RQ3: The Proposed Approach Outperform Baseline Approaches

We compared the proposed approach against existing approaches, and our evaluation results are presented in Table 5 and Table 4. Table 5 presents how many fault-revealing test cases are ranked by different approaches on top k whereas Table 4 presents the precision and recall (computed according to Equation 11 and Equation 12) of the evaluated approaches. Notably, DeepOracle did not make any recommendation for the last three methods (M16-M18) because the generated neural networks did not converge (as introduced in Section 4.5). Daikon resulted in runtime exceptions on method M18, and thus this method was not counted in while computing its average precision and recall.

From these table we make the following observations:

- First, the proposed approach significantly outperformed baseline approaches with regard to both precision and recall on top k (k=1, 10, and 100, respectively). For example, its precision (and recall) @top 1 is 67%, significantly higher than that of the random selection (6%), coverage-based approach (6%), and invariant-based approach (18%).
- The performance of coverage-based approach is close to that of Random Selection. One possible reason is that the involved faulty methods contain few branch nodes, and thus only a few test cases can maximize the coverage. However, once the coverage is maximized, coverage could not guide the selection of test cases anymore. The performance of Random Selection is poor because only a small part (**% on average) of the test cases are fault-revealing.
- Invariant-based approach significantly outperformed Random Selection and coverage-based approach. Compared to Random Selection and coverage-based approach, it improved the precision @top 1 by 200%. The evaluation results suggest that discovering invariants by mining executions of test cases is beneficial in picking fault-revealing test cases. It also confirms the conclusion in Section 2.1 that it is promising to derive test oracles by mining dynamic executions of software systems.

We conclude based on the preceding analysis that the proposed approach significantly improves the performance in selecting fault-revealing test cases.

4.8 RQ4: The Proposed Approach Is Efficient

Generating a large number of test cases and training deep neural networks with such test cases could be time and resource consuming. To reveal how efficient the proposed approach is, we record the training time and testing time for each of the faulty methods. The evaluation was conducted on a personal computer with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz and 16GB RAM.

Our evaluation results suggest that the proposed approach can generate a million test cases and train generated network within 10 minutes in case the network converges. In case the network does not converge, we allocate the maximal time slot (30 minutes). Our evaluation results also suggest that it took DeepOracle less than 10 seconds to select fault-revealing test cases from 10,000 candidate test cases for a given method.

Table 5: Performance of Evaluated Approaches

ID	Network-based			Random Selection			Coverage-based			Invariant-based		
	Top 1	Top 10	Top 100	Top 1	Top 10	Top 100	Top 1	Top 10	Top 100	Top 1	Top 10	Top 100
M1	1	10	100	0	0	1	0	0	3	1	10	100
M2	1	10	100	0	0	0	0	0	3	0	0	1
M3	1	10	100	0	0	0	0	0	0	0	0	0
M4	1	10	100	0	0	0	0	0	0	0	0	0
M5	1	10	100	0	0	1	0	0	1	0	0	0
M6	1	10	100	0	0	0	0	0	0	0	0	0
M7	1	10	100	0	0	6	1	1	7	0	1	5
M8	1	10	100	1	2	10	0	1	8	0	0	4
M9	1	10	100	0	0	0	0	0	0	0	0	2
M10	1	10	100	0	0	1	0	1	3	1	10	100
M11	1	10	100	0	0	2	0	0	4	0	0	4
M12	1	9	69	0	0	1	0	0	2	1	10	100
M13	0	0	0	0	0	2	0	0	3	0	0	0
M14	0	0	0	0	0	5	0	1	3	0	1	4
M15	0	0	0	0	0	2	0	0	2	0	0	1
M16	0	0	0	0	1	3	0	0	2	0	0	1
M17	0	0	0	0	0	0	0	0	0	0	0	0
M18	0	0	0	0	0	0	0	0	0	X	X	X
Average	0.67	6.61	64.94	0.06	0.17	1.89	0.06	0.22	2.28	0.18	1.88	18.94

We conclude based on the preceding analysis that the proposed approach is efficient.

4.9 Threats to Validity

A threat to external validity is that the proposed approach was evaluated with only 18 faulty methods. Some special characters of such methods may have biased the conclusions drawn on the evaluation. To reduce the threat, we have tried all of the faulty methods (around eight hundreds) in Defects4J. To the best of our knowledge, it is the largest repository of real-world defects with accurate patches. However, for various reasons discussed in Section 4.2, only 18 faulty methods were selected. For example, on 38% of the faulty methods, we can accurately distinguish failed test cases from passed ones solely by runtime exceptions, which makes complex alternative approaches (e.g., ours) useless and superfluous. Consequently, we excluded such methods from the evaluation although it is likely that our approach (and baseline approaches) may work well on such methods. To further improve the external validity, however, we should evaluate the approach with more faulty methods in future.

A threat to internal validity is that the ground truth (i.e., which test cases are fault-revealing) could be inaccurate. A test cases was taken as "fault-revealing" test case if it led to different outputs on the faulty method and its fixed version. However, the difference could be caused by randomness of the method (e.g., random numbers generated and exploited by the method), instead of the bug-fixing. Consequently, such "fault-revealing" test cases may fail to reveal faults in practise. To reduce the threat, we manually check the involved methods to make sure that they are deterministic and their outputs are not dependent on randomness.

5 CONCLUSION AND FUTURE WORK

REFERENCES

[1] Babyak and A. Michael. 2004. What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models. *Psychosomatic Medicine* 66, 3 (2004), 411–421.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[3] Ronyérison Braga, Pedro Santos Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. 2018. A Machine Learning Approach to Generate Test Oracles. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering* (Sao Carlos, Brazil) *(SBES '18)*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/3266237.3266273

[4] George Candea, Stefan Bucur, and Cristian Zamfir. 2010. Automated Software Testing as a Service. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 155–160. https://doi.org/10.1145/1807128.1807153

[5] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

[7] E. Diaz, J. Tuya, and R. Blanco. 2003. Automated software testing using a metaheuristic technique based on Tabu search. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 310–313. https://doi.org/10.1109/ASE.2003.1240327

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. 2002. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering* 28, 2 (Feb 2002), 159–182. https://doi.org/10.1109/32.988497

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb 2001), 99–123. https://doi.org/10.1109/32.908957

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb 2001), 99–123. https://doi.org/10.1109/32.908957

[11] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A

Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 24, 4, Article 23 (Sept. 2015), 49 pages. https://doi.org/10.1145/2699688

[12] F. Gholami, N. Attar, H. Haghighi, M. V. Asl, M. Valueian, and S. Mohamadyari. 2018. A classifier-based test oracle for embedded software. In *2018 Real-Time and Embedded Systems and Technologies (RTEST)*. 104–111. https://doi.org/10.1109/RTEST.2018.8397165

[13] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/2931037.2931061

[14] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto. 2018. An Empirical Investigation on the Readability of Manual and Generated Test Cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 348–3483.

[15] J. Huang, L. Qu, R. Jia, and B. Zhao. 2019. O2U-Net: A Simple Noisy Label Detection Approach for Deep Neural Networks. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 3325–3333. https://doi.org/10.1109/ICCV.2019.00342

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[17] U. Kanewala and J. M. Bieman. 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 1–10. https://doi.org/10.1109/ISSRE.2013.6698899

[18] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. 2018. Test Case Prioritization Approaches in Regression Testing. *Inf. Softw. Technol.* 93, C (Jan. 2018), 74–93. https://doi.org/10.1016/j.infsof.2017.08.014

[19] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93. https://doi.org/10.1016/j.infsof.2017.08.014

[20] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. 2003. Gast: Generic Automated Software Testing. In *Implementation of Functional Languages*, Ricardo Peña and Thomas Arts (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–100.

[21] Mark Last, Menahem Friedman, and Abraham Kandel. 2003. The Data Mining Approach to Automated Software Testing. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, D.C.) *(KDD '03)*. Association for Computing Machinery, New York, NY, USA, 388–396. https://doi.org/10.1145/956750.956795

[22] Pascale Le Gall and Agnès Arnould. 1996. Formal specifications and test: Correctness and oracle. In *Recent Trends in Data Type Specification*, Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 342–358.

[23] H. Liu, X. Liu, and T. Y. Chen. 2012. A New Method for Constructing Metamorphic Relations. In *2012 12th International Conference on Quality Software*. 59–68. https://doi.org/10.1109/QSIC.2012.10

[24] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. 2000. Automated Test Oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-First Century Applications* (San Diego, California, USA) *(SIGSOFT '00/FSE-8)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/355045.355050

[25] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. 2013. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. *IEEE Transactions on Software Engineering* 39, 9 (Sep. 2013), 1230–1244. https://doi.org/10.1109/TSE.2013.10

[26] Dennis Peters and David L. Parnas. 1994. Generating a Test Oracle from Program Documentation: Work in Progress. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, Washington, USA) *(ISSTA '94)*. Association for Computing Machinery, New York, NY, USA, 58–65. https://doi.org/10.1145/186258.186508

[27] Lawrence Philips. 2000. The Double Metaphone Search Algorithm. *C/C++ Users Journal* 18 (06 2000), 38–43.

[28] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. 1992. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia) *(ICSE '92)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/143062.143100

[29] E. S. Ristad and P. N. Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 5 (May 1998), 522–532. https://doi.org/10.1109/34.682181

[30] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. 2011. An Automated Framework for Software Test Oracle. *Inf. Softw. Technol.* 53, 7 (July 2011), 774–788. https://doi.org/10.1016/j.infsof.2011.02.006

[31] Seyed Reza Shahamiri, Wan M. Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Hashim. 2012. Artificial Neural Networks as Multi-Networks Automated Test Oracle. *Automated Software Engg.* 19, 3 (Sept. 2012), 303–334. https://doi.org/10.1007/s10515-011-0094-z

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[33] Sainbayar Sukhbaatar, Joan Bruna, Manohar Paluri, Lubomir Bourdev, and Rob Fergus. 2015. Training Convolutional Networks with Noisy Labels. arXiv:1406.2080 [cs.CV]

[34] Meenakshi Vanmali, Mark Last, and Abraham Kandel. 2002. Using a neural network in the software testing process. *International Journal of Intelligent Systems* 17, 1 (2002), 45–62. https://doi.org/10.1002/int.1002 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/int.1002

[35] Song Wang, Jaechang Nam, and Lin Tan. 2017. QTEP: Quality-Aware Test Case Prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 523–534. https://doi.org/10.1145/3106237.3106258

[36] Tao Xie and David Notkin. 2003. Tool-Assisted Unit Test Selection Based on Operational Violations. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (Montreal, Quebec, Canada) *(ASE'03)*. IEEE Press, 40–48.

[37] T. Xie and D. Notkin. 2005. Checking inside the black box: regression testing by comparing value spectra. *IEEE Transactions on Software Engineering* 31, 10 (Oct 2005), 869–883. https://doi.org/10.1109/TSE.2005.107

[38] T. Xie and D. Notkin. 2005. Checking inside the black box: regression testing by comparing value spectra. *IEEE Transactions on Software Engineering* 31, 10 (Oct 2005), 869–883. https://doi.org/10.1109/TSE.2005.107

[39] Z. Xu, X. Chang, F. Xu, and H. Zhang. 2012. L-1/2 Regularization: A Thresholding Representation Theory and a Fast Solver. *IEEE Transactions on Neural Networks and Learning Systems* 23, 7 (2012), 1013–1027.

[40] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-Based Inference of Polynomial Metamorphic Relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 701–712. https://doi.org/10.1145/2642937.2642994

[41] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey. 2020. Metamorphic Relations for Enhancing System Understanding and Use. *IEEE Transactions on Software Engineering* 46, 10 (Oct 2020), 1120–1154. https://doi.org/10.1109/TSE.2018.2876433