Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1275-1282
DOI: 10.23940/ijpe.18.06.p18.12751282

Challenges of Testing Machine Learning Applications

Song Huang[a], Er-Hu Liu[b,*], Zhan-Wei Hui[a], Shi-Qi Tang[b], and Suo-Juan Zhang[b]

[a]*Software Testing and Evaluation Centre, Army Engineering University of PLA, Nanjing, 210001, China*
[b]*Command & Control Engineering College, Army Engineering University of PLA, Nanjing, 210001, China*

Abstract

Machine learning applications have achieved impressive results in many areas and provided effective solution to deal with image recognition, automatic driven, voice processing etc. problems. As these applications are adopted by multiple critical areas, their reliability and robustness becomes more and more important. Software testing is a typical way to ensure the quality of applications. Approaches for testing machine learning applications are needed. This paper *analyzes* the characteristics of several machine learning algorithms and concludes the main challenges of testing machine learning applications. Then, multiple preliminary techniques are presented according to the challenges. Moreover, the paper demonstrates how these techniques can be used to solve the problems during the testing of machine learning applications.

Keywords: machine learning application; test oracle; corner case; test coverage

(Submitted on March 21, 2018; Revised on April 20, 2018; Accepted on May 16, 2018)

1. Introduction

Learning from data without being explicitly programmed is a specialty of Machine Learning (ML) algorithms and also the major difference from other computer programs [1]. Many ML algorithms like Support Vector Machine (SVM), Naive Bayesian, K-Nearest Neighbors, Deep Learning, etc. have provided important functionality to support solution in applications [2].

Software test is an activity that can identify errors, faults, defects or anomalies in the application under test. Test oracle is a mechanism for determining whether a test has passed or failed comparing the outputs of the system under test for a given test-case input [3]. However, sometimes reliable test oracle of applications under test is unavailable or not easy to implement. This situation is widely known as “the oracle problem” [4]. Test coverage is a measure of how much of the application can been verified. Sometimes, the test coverage of an oracle problem is difficult to be calculated; this situation is called “the test coverage problem”. One goal of software testing is testing the application adequately using least test cases. The test coverage problem may lead to the failure of this goal. The test oracle and coverage problems are two major problems that have to be solved.

The oracle problem becomes prominent during the process of testing ML applications. It is because in unlike traditional applications, the rules and logic of a ML application are not only written manually by programmer but also learned from data [5]. According to this specialty of ML, applications programmers and test practitioners have no idea what correct output it should be. Intuitively, ML applications are often taken as non-testable applications [2] [4].

ML applications’ data-based specialty also causes test coverage problem. For an arbitrary input, in which rule and logic of the ML application will be activated is unavailable. For example, with deep neuron network (DNN), it is impossible for test practitioners to check the weight of every single neuron, which plays as the rule of the application. Oracle and coverage problems with ML applications test bring several new challenges different from traditional applications. In the past few

* Corresponding author.
E-mail address: waterworld625@126.com

years, many test practitioners have made great effort to test ML applications, and a variety of methods have been proposed according to different ML algorithms.

Due to the commonness of different ML algorithms, the test techniques may be crossly adopted. Due to the own specialty of different ML algorithms, practitioners have to find targeted method to make the test work more effective.

In this paper, we will focus on one important category of ML application called classifier, analyze the difficulties and challenges of ML application test, and summarize the techniques used in the test work. Though, test practitioners' previous works have given out some solutions to the problems, there are still a variety of unsolved problems we have to face. These unsolved problems may be the direction of our future work.

The rest of the paper is organized as follows: Section 2 introduces two typical machine learning algorithms (SVM and Deep Learning) used in ML applications. Section 3 identifies the challenges of ML application testing based on the algorithms introduced previously. Section 4 summarizes the techniques used in ML applications test and analyzes the unsolved problems. Section 5 concludes the paper and describes the direction of future work.

2. Background

In order to give a brief impression of ML algorithms, we will take Naive Bayesian classifier and Deep Neural Network (DNN) classifier for examples, which are used in solving classification problems, and present the basic ideas of these two algorithms.

2.1. Classification problem

The classification problem is described as follows. Given a training data set which contains two k-size subsets. One subset is training sample subset, denoted as $X=<x_1, x_2, \dots, x_k>$. Each x ($x \in X$) has n attributes denoted as ($att_1, attr_2, \dots, attr_n$), represent n features from which to learn. The other subset is the corresponding label subset, denoted as $L=<l_1, l_2, \dots, l_k>$. Each l ($l \in L$) is chosen from the class label set C, $C=< c_1, c_2, \dots, c_m >$, m is the number of all the classes. After analyzing the training data set, a prediction model will be build, which is applied to predict the label of each individual example.

2.2. Naive Bayesian Classifier

Naive Bayesian classifier is a simple probabilistic classifier based on applying Bayes' theorem with naive independence assumptions between the features [6]. When Naive Bayesian classifier receives an individual example ($attr_1, attr_2, \dots, attr_n$), the classification problem is converted to computed the probabilities $P(c_i| att_1, attr_2, \dots, attr_n)$ for all the m possible classes. These conditional probabilities can be decomposed as:

$$P(c_i|attr_1, \dots, attr_n) = \frac{P(c_i)P(attr_1, attr_2, \dots, attr_n|c_i)}{P(attr_1, attr_2, \dots, attr_n)} \quad (1)$$

Because of the independence assumptions between the features, equation (1) can be changed into equation (2).

$$P(c_i|attr_1, \dots, attr_n) = \frac{P(c_i)\prod_{j=1}^{j=n} P(attr_j|c_i)}{\sum_k P(c_k)\prod_{j=1}^{j=n} P(attr_j|c_k)} \quad (2)$$

The equation (2) can be represented by a generalization equation as equation (3).

$$P(c_i|attr_1, \dots, attr_n) = \frac{prior\ probability * likelihood}{evidence} \quad (3)$$

The prior probability, likelihood and evidence in equation (3) are learned from the training data set, and the probabilities of the given example belonging to each class can be computed. After comparing the probabilities of all the classes, the Naive Bayesian classifier classifies the given example to the class with highest probability.

2.3. DNN Classifier

A DNN is an artificial neural network with multiple hidden layers between the input and output layer [7]. The DNN is the main component of a classifier. Figure 1 illustrates simple version architecture of a DNN. This DNN has three different

kinds of layers: one input layer, one output layer and multiple hidden layers. Each layer consists of a sequence of neurons. The number of the neurons in the output layer is always as same as the number of the label classes. The neurons play as the computing units, and the details of the neuron are shown on the right of Figure 1.

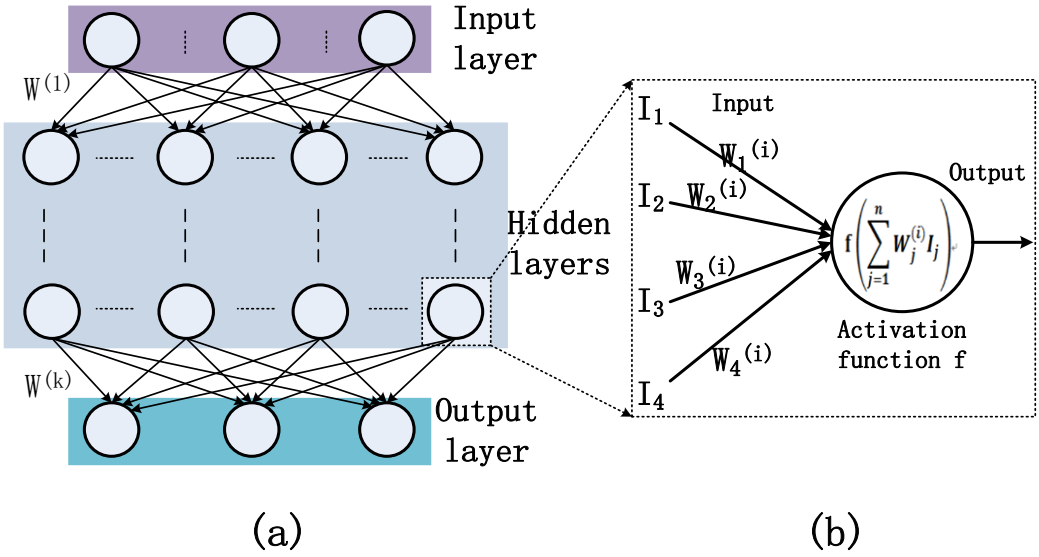

Figure 1: (a) A simple DNN classifier with multiple hidden layers. (b) Details of a single neuron

Each neuron is directly connected with other neurons in the next layer. The connection strength of two neurons is indicated by the corresponding weight, which is the parameter learned from the training data. After the training phase, all the weights are learned. When input with an individual example, the outputs of the neurons are computed by a non-linear activation function. The results are then passed into the connected neurons as inputs in the next layer. The output of the output layer is the confidence values of each class. Similar to the Naive Bayesian classier, the DNN classifier classifies the input example to the class with highest confidence value.

In Naive Bayesian, DNN classifiers and other ML algorithms, the models extract and assemble the meaningful information to make the classification decision. Even though the forms of the meaningful information may be different (e.g. probabilities in Bayesian and weights in DNN), it is all learned from the training data, running as the rule and logic of the applications. Due to the self-learning characteristic, multiple new challenges are brought to the testing practitioners during the ML application testing process.

3. Challenges in machine learning application testing

This section provides the potential challenges when trying to test the ML applications.

3.1. Generating reliable test oracles

The term “test oracle” was first introduced by Howden in 1978 [8]; it indicates what the correct output of an application should be responding to an arbitrary input. Testing practitioners determine whether a test has passed or failed by comparing the output of the application under guidance of test oracle. The literature of test oracle has developed and several techniques of generating test oracle have been proposed. Earl et al. made a survey on the oracle problems, and classified the test oracle into different kinds [9]. Peters et al. proposed a method of generating test oracle from program documentation [10], and Lin et al. proposed a model-based test oracle generation technique [11]. Statistical methods can be used in generating test oracle [12]. The idea of consistency oracle is comparing the results of multiple implementations with a same function [13]. Metamorphic testing makes use of follow-up test cases to generate the test oracle [14, 15]. We will introduce more details in the next section. Humans may also extract information from the application under test and provide informal test oracle.

The rule and logic of traditional application are expressed in terms of control flow, which are encoded by programmers. When an input is received, a certain operation is used to transform a statement to another. It is possible to derive an oracle. When dealing with testing ML application, the methods introduced above may become invalid. The ML algorithms are data sensitive; the rule and logic (i.e. the probability in Naive Bayesian classifier and weights in DNN classifier) are learned

from the training data. For example, DNN classifier have no explicit branches but the weights which indicate the neurons' influence the following neurons. As a result of the uncertainty existing in the program, the test oracle is unavailable. Moreover, creating specifications that can be used as test oracle for complex DNN applications is infeasible as the logic is too complex to manually encode [10, 16]. So, it is absolutely a challenge to find the test oracle of a ML application.

3.2. Generating effective corner cases

Effective software testing can be described as detecting more defects with a small number of test cases. As a result, if a test case of a ML application is caused by incorrect behavior, this is a good test case. In the real world, ML applications' test cases are always chosen from the testing data sets, which heavily depend on manually labeled data. This kind of test case often failed to cause incorrect behaviors. Differently, a corner case involves variables or situations at extreme levels [17], takes high probability to result in error behavior, and exposes the defect of the ML application under test. Moreover, corner cases can be added to the training data set to retrain the system. After retraining work, the accuracy of the application can be enhanced, and the risk of over-fitting may be relieved.

Manual collection of corner case is impossible when the test conditions become complex. The method of generating corner cases automatically is needed. Generating corner cases automatically is also a challenge.

3.3. Improving test coverage

Test coverage is a measurement used to describe the degree to which the source code of a program is assessed with the supplied test inputs [4, 18]. To our intuition, the higher coverage a test has achieved, the lower probability that the application under test contained undetected bugs. Several metrics have been proposed to measure the test coverage, such as statement coverage, branch coverage, condition coverage, modified condition/decision coverage, multiple condition coverage, etc. [19, 20]. These metrics are effective in testing ML applications in code level.

Different from traditional application, even a few test inputs can achieve high statement coverage in ML application testing. However, only a low percentage of the application is assessed. Pei et al. found that a small number of test inputs can achieve 100% code coverage for all DNNs where neuron coverage is actually less than 34% [5]. By analyzing the details of several typical DNNs (e.g. LeNet-1, LeNet-4, VGG-16 [21]), we find the control flow programed by the developer is simple, and all the computation is influenced by the weights of the activated neurons'. For DNNs, the number of activated neurons may be a better metric of test coverage. Because of the non-linearity of the activation function, it is very difficult to enlarge the number of activated neurons. Generalizing to all the ML applications, a new metric of test coverage is needed which represents how much of the rule and logic learned from the training data is involved. Then, the next work is attempting to achieve higher test coverage as possible.

3.4. Testing the ML applications with millions of parameters

As the ML applications have achieved great success in many fields, more complex problems are expected to be solved using ML applications. Moreover, influenced by the idea of higher depth making more success, the scale of the ML applications becomes larger and larger. The champion of the ILSVRC 2015 classification task is a deep neural networks with 152 layers and millions of parameters [22]. The parameters are part of the system that needs to be tested.

The specialty of large scale brings multiple difficulties to testing the systems. First, such large numbers of parameters need a mountain of test cases to be executed. Test cases, especially the corner cases, are not easily generated. Second, even though much more attention is paid to coverage all the parameters, it is impossible to evaluate all the parameters. Third, methods of identifying the erroneous behavior of the system automatically are desperately needed. Finally, a vast amount of time and memory space are consumed during the testing work.

4. Techniques used in testing applications

Over the past several years, different approaches to test the ML applications have been proposed that attempt to overcome the above-mentioned difficulties. This section will present multiple approaches against part of all the challenges introduced in section 3.

4.1. Approaches for alleviating the oracle problems

To relieve the test oracle problem, one recommended approach is testing with pseudo oracles [23]. Another approach is metamorphic testing. We will introduce two methods to find the test oracles.

4.1.1. Multiple implementations with same inputs

The idea of pseudo oracles is generating multiple independent implementations of same algorithm, and comparing the outputs of these implementations according to same input. If the outputs are different, there may be a defect in one or more implementations. Pei et al. developed a white-box framework for systematically testing real-world deep learning systems [4]. This framework uses the results' consistency of different implementations with same function as the test oracle. The framework takes the result advocated by most implementations as the correct outputs responding to the input. It is not always easy to generate other implementations base on the same algorithm. Code search engines and the open source software are widely adopted to generate different implementations. As a result, we must pay attention to ensure the independence between different implementations [24-27].

4.1.2. Metamorphic testing

Another method to generate test oracles is metamorphic testing, which we have mentioned in Section 3. During the metamorphic testing, several properties of the program should first be identified [28-33]. Properties called metamorphic relationships indicate the relations over two or more inputs and corresponding outputs. Then, the following cases are generated base on the given source cases. After the executions of the source and following cases, the satisfaction of the metamorphic relationships is checked. If the metamorphic relationships are broken, the program under test may contain a defect. Murphy et al. sought several metamorphic relationships for test ML applications in 2008 [23]. They classified these relationships into six classes (i.e. additive, multiplicative, permutative, invertive, inclusive, and exclusive) and used them to test two ML applications. Murphy's work provided a foundation for testing ML applications with metamorphic testing. Xie et al. broadened Murphy's work by adding some new metamorphic relationships [2]. Additionally, they identified 6 metamorphic relationships for K-Nearest Neighbors classifier and 9 for Naive Bayesian classifier. A systematic testing tool named DeepTest for testing DNN-driven vehicles also used metamorphic testing [16]. The tool generated a corner test case from a source test case. The source and following test cases have the same label, and should triggered the same behavior. The metamorphic relationship used in DeepTest is the consistency of the behaviors corresponding to the source and following test cases.

Intuitionally, metamorphic is an effective method to generation the test oracles; however, the generality of metamorphic relationships is low, and we have to identify the corresponding metamorphic relationships by analyzing the characteristics of ML applications under test.

4.2. Approaches for increasing the test coverage

Due to the different rule and logic expressions between ML application and traditional application, new coverage metrics are needed. Pei et al. proposed neuron coverage for the deep neuron networks as the new metric [5]. The neuron coverage can be computed as equation (4).

$$neuron\ coverage = \frac{number\ of\ activated\ neurons}{number\ of\ total\ neurons} \tag{4}$$

An activated neuron is the neuron whose output is larger than the hyperparameter: threshold. The higher neuron coverage is, the more weights, learned from training data, can be evaluated. Generalizing other ML applications, the new metric should indicate the degree of how much of the knowledge learned from the data assessed.

4.3. Approaches for generating corner cases

Three different approaches for generating corner cases will be presented in the following passage. The first approach is changing the value of source case along the direction of objective function's gradient ascent. The second is transformation due to application scenarios, and the last is generating adversarial examples. Though the approaches were validated based on deep learning architecture, they can be generalized to ML applications with other architectures.

4.3.1. Gradient ascent algorithm

A gradient ascent algorithm for generating corner test cases was proposed by Pei et al. in the design of a deep learning system testing framework [5]. In this framework, input X was executed by three different implementations using the same algorithm. In Pei's idea, a corner case was the input, which resulted in different outputs. They defined an objective function

obj(X) which presented the difference between the three outputs, and made the input X as the variable. When a seed input was executed, the framework would change the input value along the gradient ascent direction iteratively to enlarge value of the objective function. Until the changed seed input X' resulted in different outputs, X' was considered a corner case.

In Pei's experiments of MNIST, ImageNet, Driving, etc., five popular datasets with different type of datasets were adopted, and each dataset were implemented with three different architectures. All these datasets and architectures are open source, meaning we can get them from the Internet. We are given a new dataset and implementation to build two other implementations based on the dataset may cost more time and energy. 2000 seed inputs were selected randomly from each dataset, and more than 1500 corner cases were generated on average. Obviously, it is an effective method for generating corner cases. However, these corner cases were generated without thinking about the realistic condition; the ML applications would never face some of these corner cases in the real world.

4.3.2. Transformations due to applications scenarios

ML applications are used in their own scenarios. The corner cases based on the realistic condition may be more significant. For example, generating handwritten digit corner cases should consider different writing styles (see the left part of figure 2), the image corner case may come from the camera lens distortions or the lighting condition (see the right part of figure 2), and the noise from the environment is prone to be the corner case of voice.

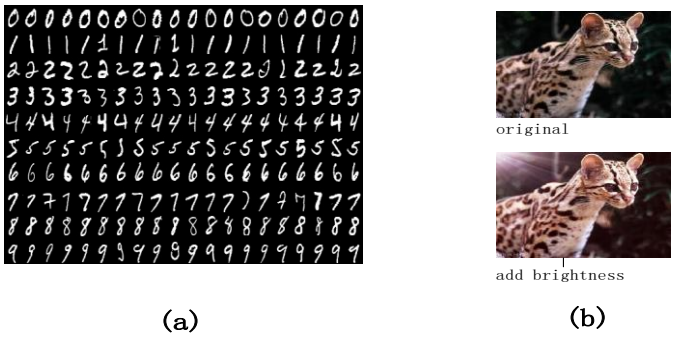

Figure 2: (a) A group of handwritten digits based on different writing style. (b) Images of a wild cat taken under different bright conditions.

Tian et al. made eight different transformations (e.g. fog, rain, translation, scale, shear, rotation, contrast, brightness) due to the driven scenarios based on a given image input to generate the corner cases for the DNN driven system [16]. The eight different transformations simulated eight realistic conditions. The realistic conditions are uncountable and contain uncertainty. It is impossible to enumerate the corner cases based on all the realistic conditions.

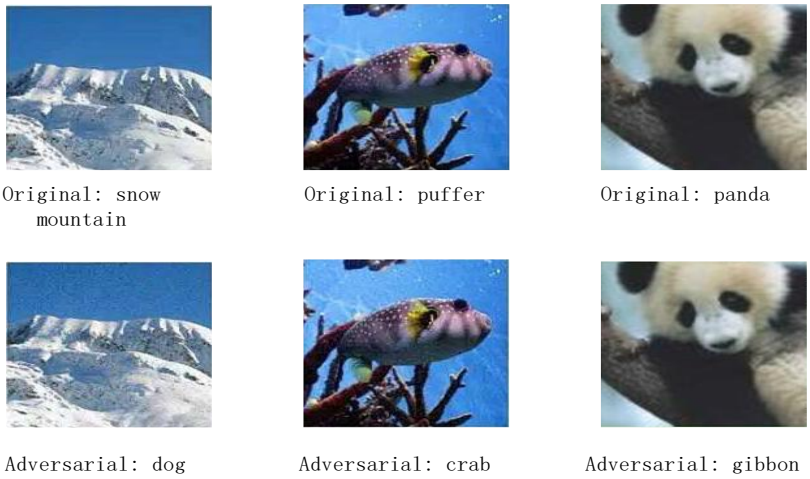

Figure 3: The first line is the original images, and the second line is their corresponding adversarial examples.

4.3.3. Adversarial example

The definition of adversarial example was first introduced and proved to exist by Szegedy et al. in 2013 [34]. He found that adding an imperceptible non-random perturbation to a source test can fool the deep learning system under test. The new test case is called adversarial example. Adversarial examples are always slightly different from the source cases [35-43]. Three

pairs of image test case and corresponding adversarial example are shown in Figure 3. By adding some perturbations, the classifiers will mistake the contents in the original image to others.

In this paper, we treat the adversarial examples as a special category of corner cases because malicious attacks based on adversarial against ML applications exists, and can always trigger the erroneous behaviors. Many methods have been proposed for generating adversarial examples. Some of them computed gradients directly with respect to the image pixels, and others expressed the problem as optimization on the image pixels [34, 36, 44, 45]. Before employment, we need to use the adversarial examples to find the defect of the ML application, and improve the robustness.

5. Discussions

As the development of software engineering deepens, the scale of the software application expands rapidly. Testing a ML application with millions of parameters manually will cost too much time and manual labour. Automatic test is an effective method to reduce the cost [46, 47]. However, the realization of automatic test is a challenge not only for traditional applications but also for ML applications, and depends on the solutions of all challenges discussed above. We discuss the probability of testing ML applications automatically in this section.

It is impossible to completely test a given ML application automatically. We can divide the whole test work into several different phases [46, 48, 49]. Some decisive work can be completed by humans, and the repetitive work is assigned to a specific program, which can complete the work automatically. Take the generation of test cases for example. First, the test practitioner chose an appropriate algorithm (e.g. gradient ascent algorithm, different transformations, several adversarial example algorithms, etc. [5, 16, 37]); then, the implementation of the algorithm generated a larger number of test cases automatically. Similarly, the execution of the test cases, the identification of the test oracle, the prediction of the defects and other work containing repetitiveness may be completed by corresponding programs.

6. Conclusions

In this paper, we introduced several difficulties in traditional software testing work. Then, we analyzed the challenges of testing the ML applications based on Naive Bayesian classifier and DNN classifier. Two techniques (i.e. comparing results of different implementations and metamorphic testing) were presented to relieve the test oracle problem. Using gradient ascent algorithm, transformations due to the application scenarios and generating adversarial examples etc. three methods for generating corner test cases were introduced. Lastly, we discussed the probability of automatic test for ML applications.

Acknowledgements

This work is supported by the Natural Science Foundation of China (No: 61702544), the Natural Science Foundation of Jiangsu Province, China (No: BK20160769, BK20141072), China Postdoctoral Science Foundation (No: 2016M603031).

References

1. *Machine Learning*. Available: https://en.wikipedia.org/wiki/Machine_learning
2. X. Y. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. W. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, pp. 544-558, Apr 2011.
3. *Test oracle*. Available: https://en.wikipedia.org/wiki/Test_oracle#cite_note-2
4. S. Nakajima and H. N. Bui, "Dataset Coverage for Testing Machine Learning Computer Programs," in *2016 23rd Asia-Pacific Software Engineering Conference*, A. Potanin, G. C. Murphy, S. Reeves, and J. Dietrich, Eds., ed New York: Ieee, 2016, pp. 297-304.
5. K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," pp. 1-18, 2017.
6. *Naive Bayes classifier*. Available: https://en.wikipedia.org/wiki/Naive_Bayes_classifier
7. *Deep Neural Network*. Available: https://en.wikipedia.org/wiki/Deep_learning
8. W. E. Howden, "Theoretical and Empirical Studies of Program Testing," *IEEE Trans.softw.eng*, vol. 4, pp. 305-311, 1978.
9. E. T. Barr, M. Harman, P. Mcminn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, pp. 507-525, 2015.
10. D. Peters and D. L. Parnas, "Generating a test oracle from program documentation:work in progress," in *International Symposium on Software Testing and Analysis*, 1994, pp. 58-65.
11. P. Lin, J. Thangarajah, Z. Zhang, and T. Miller, "Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1230-1244, 2013.
12. J. Mayer and R. Guderlei, "Test Oracles Using Statistical Methods," in *Testing of Component-Based Systems and Software Quality, Proceedings of Soqua*, 2004, pp. 179-189.
13. A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans.softw.eng*, vol. 11, pp. 1491-1501, 1985.

14. T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," 1998.
15. Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *Acm Computing Surveys*, 2018.
16. Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars," 2017.
17. *Corner Case*. Available: https://en.wikipedia.org/wiki/Corner_case
18. *test coverage*. Available: https://en.wikipedia.org/wiki/Code_coverage
19. G. J. Myers and C. Sandler, *The Art of Software Testing*: Wiley, 1979.
20. S. Godboley, G. S. Prashanth, D. P. Mohapatro, and B. Majhi, "Increase in Modified Condition/Decision Coverage using program code transformer," in *Advance Computing Conference*, 2013, pp. 1400-1407.
21. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *Computer Science*, 2014.
22. K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Computer Vision and Pattern Recognition*, 2016, pp. 770-778.
23. C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of Machine Learning Applications for Use in Metamorphic Testing," *Department of Computer Science Columbia University*, pp. 867-872, 2008.
24. W. B. Langdon, S. Yoo, and M. Harman, "Inferring automatic test oracles," in *Ieee/acm International Workshop on Search-Based Software Testing*, 2017, pp. 5-6.
25. N. Sahavechaphan and K. Claypool, "XSnippet: mining For sample code," in *ACM Sigplan Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006, pp. 413-430.
26. P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti, "PODS — A project on diverse software," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 929-940, 2012.
27. S. Bajracharya and C. Lopes, "Mining search topics from a code search engine usage log," in *IEEE International Working Conference on Mining Software Repositories*, 2010, pp. 111-120.
28. Y. Cao, Z. Q. Zhou, and T. Y. Chen, "On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions," in *International Conference on Quality Software*, 2013, pp. 153-162.
29. T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou, "Case Studies on the Selection of Useful Relations in Metamorphic Testing," 2004.
30. T. Y. Chen, P. L. Poon, and X. Xie, "METRIC: METamorphic Relation Identification based on the Category-choice framework ☆ ," *Journal of Systems & Software*, vol. 116, p. 0000, 2016.
31. H. Liu, F. C. Kuo, D. Towey, and T. Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?," *IEEE Transactions on Software Engineering*, vol. 40, pp. 4-22, 2014.
32. H. Liu, X. Liu, and T. Y. Chen, "A New Method for Constructing Metamorphic Relations," in *International Conference on Quality Software*, 2012, pp. 59-68.
33. J. Mayer and R. Guderlei, "An Empirical Study on the Selection of Good Metamorphic Relations," in *Computer Software and Applications Conference, 2006. COMPSAC '06. International*, 2006, pp. 475-484.
34. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, *et al.*, "Intriguing properties of neural networks," *Computer Science*, 2013.
35. X. Xu, X. Chen, C. Liu, A. Rohrbach, T. Darell, and D. Song, "Can you fool AI with adversarial examples on a visual Turing test?," 2017.
36. Z. Zhao, D. Dua, and S. Singh, "Generating Natural Adversarial Examples," 2017.
37. L. Metz, B. Poole, D. Pfau, and J. Sohldickstein, "Unrolled Generative Adversarial Networks," 2017.
38. N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," pp. 39-57, 2016.
39. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring Neural Net Robustness with Constraints," 2016.
40. K. Chalupka, P. Perona, and F. Eberhardt, "Visual Causal Feature Learning," *Computer Science*, 2015.
41. I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *Computer Science*, 2014.
42. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," 2016.
43. U. Shaham, Y. Yamada, and S. Negahban, "Understanding Adversarial Training: Increasing Local Stability of Neural Nets through Robust Optimization," *Computer Science*, 2015.
44. S. Baluja and I. Fischer, "Adversarial Transformation Networks: Learning to Generate Adversarial Examples," 2017.
45. A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial Machine Learning at Scale," 2016.
46. J. Edvardsson, "A Survey on Automatic Test Data Generation," 1999.
47. F. Corno, E. Snchez, M. S. Reorda, and G. Squillero, "Automatic Test Program Generation: A Case Study," *IEEE Design & Test*, vol. 21, pp. 102-109, 2004.
48. R. A. Demillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans.softw.eng*, vol. 17, pp. 900-910, 1991.
49. C. Nebut, F. Fleurey, Y. L. Traon, and J. M. Jezequel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 140-155, 2006.