SCIENCE CHINA
Information Sciences

• PERSPECTIVE •

July 2023, Vol. 66 176101:1–176101:3
https://doi.org/10.1007/s11432-022-3580-5

Toward actionable testing of deep learning models

Yingfei XIONG[1,2*], Yongqiang TIAN[3,4*], Yepang LIU[5,6*] & Shing-Chi CHEUNG[3*]

[1]*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing* 100871, *China;*
[2]*School of Computer Science, Peking University, Beijing* 100871, *China;*
[3]*Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong* 999077, *China;*
[4]*School of Computer Science, University of Waterloo, Waterloo* ON N2L 3G1, *Canada;*
[5]*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen* 518055, *China;*
[6]*Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen* 518055, *China*

Received 8 February 2022/Revised 12 June 2022/Accepted 30 August 2022/Published online 12 June 2023

Citation Xiong Y F, Tin Y Q, Liu Y P, et al. Toward actionable testing of deep learning models. Sci China Inf Sci, 2023, 66(7): 176101, https://doi.org/10.1007/s11432-022-3580-5

Deep learning has become an important computational paradigm in our daily lives with a wide range of applications, from authentication using facial recognition to autonomous driving in smart vehicles. The quality of the deep learning models, i.e., neural architectures with parameters trained over a dataset, is crucial to our daily living and economy.

To ensure the quality of deep learning models, many testing approaches have been proposed to assess various properties of the models [1, 2], such as correctness, safety, and robustness. Here, we use the term "testing" in a broad sense to refer to any approach that can detect violations of the desirable properties (called bugs), regardless of whether the bugs are detected through static analysis, dynamic analysis, or comparing the output with an oracle. Most existing approaches treat the trained deep learning models as programs and aim to find inputs that trigger incorrect outputs, also known as adversarial samples. Success in detecting thousands of adversarial inputs on widely used deep learning models [3, 4] has been reported.

However, these approaches have received limited industrial adoption, unlike their counterparts for conventional software. A possible explanation is that they are not actionable. For conventional software, a failing test typically depicts a buggy control flow and the set of variables that may have received inappropriate value assignment. In contrast, existing deep learning testing approaches do not provide similar information that helps diagnose and fix the glitches. Although many adversarial samples can be found, they provide no clue for the developers to make a modification that eliminates the incorrect predictions on certain inputs while preserving the correctness of other inputs. A straightforward approach is to add these discovered adversarial samples to augment the training set, known as adversarial training, but adding these samples will change the distribution of the training set, which may negatively affect the performance of the model [5]. Thus, most existing testing approaches may only be used to assess the quality of a model. For example, a deep learning model may be considered not robust if many adversarial samples are found. However, this assessment has limited usefulness. One cannot even compare the quality of two models based on the number of adversarial samples found because some adversarial samples may be rare or unreal in practice, and finding more such samples does not necessarily indicate less robustness.

To address this limitation, in this study, we argue that we should also consider actionable testing of deep learning models. A testing approach is actionable if it provides (actionable) clues together with the bugs detected. An (actionable) clue is a hint such that, based on the current body of human knowledge and the clue, an average developer can modify the source of the deep learning model, such as the implementation code and the training set, to remove or mitigate the impact of the bug. A clue can be a concrete modification to be conducted on the deep learning model or an explanation of the bug hinting at the changes that are needed.

Providing actionable testing of deep learning models is challenging. A major obstacle is the lack of understanding of deep learning. Given an input that leads to an incorrect output, neither the deep learning developers nor the testing researchers know how to fix the bug. Because a breakthrough in deep learning theory is unlikely in the short term, in this study, we propose a pragmatic strategy toward actionable deep learning testing: instead of trying to find a generic actionable testing approach for any deep learning defect, we characterize and classify the defects and design specific testing approaches for the popular defect types whose fixing solutions we know of. For example, when a "divided-by-zero" error occurs during neural network computation, the

* Corresponding author (email: xiongyf@pku.edu.cn, ytianas@connect.ust.hk, liuyp1@sustech.edu.cn, scc@cse.ust.hk)

computed values become INF, and the model will probably produce an incorrect output. Similar to the situations in conventional programs, divided-by-zero errors in deep learning programs are usually repairable. If the testing system could indicate which divisor becomes zero during the computation, the developers could probably repair the bug by changing the implementation code of the neural model.

Note that the idea of actionable testing is to complement non-actionable testing with guidance to improve the system. By relaxing the requirement of being actionable, non-actionable testing could possibly cover more bug types and could be useful in, for example, quality assessment and acceptance testing.

In the following, we describe a high-level conceptual framework of actionable deep learning model testing, highlight a few testing approaches that can provide actionable results, and discuss a roadmap for future research on actionable deep learning testing.

Framework. Let s and e denote a deep learning model under test and a bug detected in the model, respectively. For example, a "divided-by-zero" bug can be represented by $e = (i, o)$, where divided-by-zero occurs at division operation o given the test input i.

Based on these notations, an actionable deep learning testing approach comprises three components.

• Buggy condition. A predicate $P(s, e)$ that evaluates to true when e is a bug in the deep learning model s.

• Bug detector. A function $d : S \to 2^E$, where S is the set of deep learning models supported by the testing approach, and 2^E is the power set of all possible bugs that can be detected. This component detects a set of bugs from the deep learning model. It is sound if all detected bugs are true positives, i.e., $\forall e \in d(s), P(s, e)$. In practice, depending on the detection algorithm, false positives may occur.

• Clue generator. A function $g : S \times E \to 2^A$, where S is the set of deep learning models, E is the set of all possible bugs, and 2^A is the power set of all possible clues. This component generates clues for a bug to guide the developers in fixing the bug in the source of the model.

Take a testing approach for divided-by-zero as an example. The buggy condition is a predicate $P(s, (i, o))$ that evaluates to true when input i triggers a division with a zero divisor at the operation o. The bug detector detects as many pairs of problematic input and division operators as possible. The clue generator directly reports the bugs ((i, o) pairs) to the user. This clue is actionable because the bug is divided-by-zero, and the developers can analyze the network to understand why the bug occurs and derive modifications to the network architecture to repair the bugs.

Example approaches. We notice that some existing deep learning testing approaches are already actionable. Here, we highlight a few examples to demonstrate what clues are actionable.

DEBAR. The first example detects bugs that are similar to traditional programs. Similar to the above divided-by-zero example, these bugs are directly understandable by the developers and can be directly used as clues. DEBAR [6] is an approach for detecting numeric bugs in neural models. Similar to their counterparts in traditional programs, numeric bugs cause invalid numeric computations, such as divided-by-zero, resulting in crashes or meaningless results (e.g., INF or NaN) during neural network computation. The three components of DEBAR are described as follows.

• Buggy condition. In DEBAR, a numeric bug is defined by a numeric operator in the implementation code and the possible ranges of its operands such that the ranges contain values that would cause numeric errors through this operator. To formulate the buggy condition, DEBAR predefines the conditions where the various types of numeric operators would produce an error. For example, a division operator produces an error when the divisor is zero, and a logarithmic operator produces an error when the argument is not positive. The buggy condition holds when there exists an input such that an operator may produce a numeric error according to the predefined conditions. Because invalid numeric computation may occur at the training stage, DEBAR can also be used to test the model before training by treating the parameters as input.

• Bug detector. DEBAR employs a static analysis algorithm that includes two novel abstract domains to precisely analyze the range of values that may appear during execution for each variable.

• Clue generator. DEBAR directly returns the detected bugs as clues. Similar to numeric bugs in traditional programs, developers can debug the implementation code of the neural network based on the clues.

Object relevancy. The second example demonstrates that repair actions can be derived from the properties of the application domain. Image classification identifies the object contained in an image and is a key application of deep learning techniques. Image classification has been widely used in video surveillance, search engines, criminal investigations, etc., and thus the quality of image classification systems must be ensured. However, developing actionable testing for image classification is not easy, as we still lack an understanding of the intrinsic inference logic of neural image classifiers.

Tian et al. [7] proposed a desirable property of image classifiers, object relevancy, and a testing approach to detect bugs violating object relevancy. Object relevancy refers to whether the inferences are based on the target objects of image classifiers. For example, if in a dataset cats are always in a house while tigers are always in a forest, an image classifier may learn to distinguish cats and tigers based on their backgrounds and thus violate object relevancy, as the inference is based on the background and not the target object. Such bugs can be repaired by adding more images with diverse backgrounds to the training set. The three components of the testing approach are as follows.

• Buggy condition. Based on intuition, Tian et al. proposed two metamorphic relations (MRs) that an image classifier should satisfy. The first MR expects that modifying the target object in an image will cause the model to produce a different classification result or the same result with less certainty. The second MR expects that modifying the background will not affect the classification result. Then, a bug is defined as an image and its modified version that violate any of the MRs.

• Bug detector. The system generates random modifications to the images and checks if a violation of the MRs can be found.

• Clue generator. A detected bug implies insufficient background diversity for the target object in the training set, which provides a clue leading to repair actions. The developer can add more images of the target object with a more diverse background to the training set. In contrast to adversarial training, this clue reveals an understandable reason for the bug, and the developer can add standard images without distorting the distribution.

TransRepair and CAT. The third example demon-

strates that repair actions can be derived by wrapping the model with external runtime facilities. Modern machine translation systems, powered by deep learning models, are used by millions of users daily. The quality of such systems is critical to avoiding misunderstanding.

Multiple testing approaches [8–11] for machine translation systems have been proposed. Their basic idea is that replacing a word in a sentence with a similar word should not induce substantial changes in the structure of the sentence's translation. For example, replacing "Lily went to school" with "Lucy went to school" should only cause a change from "Lily" to "Lucy" in the translated sentence. If the translated sentence undergoes a large structural change, it is probably a mistranslation. However, similar to most testing approaches, although this approach helps detect bugs, it provides little information for bug fixing.

TransRepair [12] and CAT [13] employ a novel method for "repairing" such bugs in machine translation systems automatically. The basic assumption is that a well-trained neural network is usually correct, and thus ensemble learning can be used to avoid mistranslations (i.e., bugs). The repair method first replaces words from the original sentence to generate a set of mutated sentences, translates all these sentences, and then chooses one translation that has the smallest difference from all other translations. Finally, the replaced word in the chosen translation is put back to create the final result. In this way, the bug is repaired without actually modifying the neural model, and the testing result becomes actionable with an automatic repair solution.

Under our framework, the three components of TransRepair and CAT are as follows.

• Buggy condition. A bug is represented by two sentences that differ only in one word, and the difference should not affect the structure of the translated sentence; e.g., the two words have the same part-of-speech role and are emotionally similar in the concerned context. The buggy condition is that the two sentences are translated into two sentences with significantly different structures.

• Bug detector. The bug detector randomly generates such pairs of sentences and checks whether the buggy condition is satisfied.

• Clue generator. The clue is a direct repair of the implementation code. The inference of the model is wrapped with an online repair component that automatically generates mutants for each input to detect and repair bugs.

Roadmap. We foresee a roadmap for studying actionable deep learning testing that consists of two steps: (1) designing techniques for specific types of bugs and (2) generalizing these special cases into general actionable testing approaches. We have seen some deep learning testing approaches that are already actionable. However, the bugs covered by these approaches comprise only a small portion of all the bugs that may occur in deep learning models. In fact, all the discussed approaches cover one type of bug, and for some of them, the covered bug occurs only in a particular application of deep learning (e.g., machine translation). Therefore, the next immediate step is to identify more types of deep learning bugs where clues could be provided and develop actionable testing approaches for them. In this way, the developers can use these testing approaches together to detect a large class of bugs. One possible approach is to empirically study how practitioners develop and maintain deep learning models and summarize their bug-fixing strategies.

The second step is to generalize the special techniques to general cases. In the above-discussed actionable testing approaches, the clues provided are diverse: from bug explanations for developer reference to fully automatic bug-fixing actions and from augmenting the training set to wrapping the models with external facilities. At this stage, the commonality of the existing actionable testing approaches is difficult to summarize. This difficulty is due to the studied types of bugs thus far being still limited. With an increasing number of bugs studied, we will be able to answer broader research questions, such as what diagnosis information is needed to help debug deep learning models and form repair actions, and develop more general actionable testing approaches that are applicable to a wide range of deep learning bugs.

Acknowledgements This work was supported by National Key Research and Development Program of China (Grant No. 2019YFE0198100) and Innovation and Technology Commission of HKSAR (Grant No. MHP/055/19).

References

1 Wang Z, Yan M, Liu S, et al. Survey on testing deep learning neural networks (in Chinese). J Software, 2020, 31: 1255–1275
2 Huang X, Kroening D, Ruan W, et al. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. Comput Sci Rev, 2020, 37: 100270
3 Ma L, Juefei-Xu F, Zhang F, et al. DeepGauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of ACM/IEEE International Conference on Automated Software Engineering, 2018. 120–131
4 Pei K, Cao Y, Yang J, et al. DeepXplore: automated white-box testing of deep learning systems. In: Proceedings of the ACM Symposium on Operating Systems Principles, 2017. 1–18
5 Raghunathan A, Xie S M, Yang F, et al. Adversarial training can hurt generalization. In: Proceedings of ICML Deep Phenomena, 2019
6 Zhang Y, Ren L, Chen L, et al. Detecting numerical bugs in neural network architectures. In: Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 826–837
7 Tian Y, Ma S, Wen M, et al. To what extent do DNN-based image classification models make unreliable inferences? Empir Software Eng, 2021, 26: 84
8 Zhou Z Q, Sun L. Metamorphic testing for machine translations: MT4MT. In: Proceedings of Australian Software Engineering Conference, 2018. 96–100
9 He P, Meister C, Su Z. Structure-invariant testing for machine translation. In: Proceedings of International Conference on Software Engineering, 2020. 961–973
10 Gupta S, He P, Meister C, et al. Machine translation testing via pathological invariance. In: Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 863–875
11 He P, Meister C, Su Z. Testing machine translation via referential transparency. In: Proceedings of International Conference on Software Engineering, 2021. 410–422
12 Sun Z, Zhang J M, Harman M, et al. Automatic testing and improvement of machine translation. In: Proceedings of International Conference on Software Engineering, 2020. 974–985
13 Sun Z, Zhang J M, Xiong Y, et al. Improving machine translation systems via isotopic replacement. In: Proceedings of International Conference on Software Engineering, 2020. 1181–1192