A Study of Oracle Approximations in Testing Deep Learning Libraries

Mahdi Nejadgholi, Jinqiu Yang
Department of Computer Science and Software Engineering
Concordia University
{m_nejadg, jinqiuy}@encs.concordia.ca

***Abstract*—Due to the increasing popularity of deep learning (DL) applications, testing DL libraries is becoming more and more important. Different from testing general software, for which output is often asserted definitely (e.g., an output is compared with an oracle for equality), testing deep learning libraries often requires to perform oracle approximations, i.e., the output is allowed to be within a restricted range of the oracle. However, oracle approximation practices have not been studied in prior empirical work that focuses on traditional testing practices. The prevalence, common practices, maintenance and evolution challenges of oracle approximations remain unknown in literature.**

In this work, we study oracle approximation assertions implemented to test four popular DL libraries. Our study shows that there exists a non-negligible portion of assertions that leverage oracle approximation in testing DL libraries. Also, we identify the common sources of oracles on which oracle approximations are being performed through a comprehensive manual study. Moreover, we find that developers frequently modify code related to oracle approximations, i.e., using a different approximation API, modifying the oracle or the output from the code under test, and using a different approximation threshold. Last, we performed an in-depth study to understand the reasons behind the evolution of oracle approximation assertions. Our findings reveal important maintenance challenges that developers may face when maintaining oracle approximation practices as code evolves in DL libraries.

***Index Terms*—Software Quality Assurance, Software Testing, Testing Deep Learning Libraries, Test Oracle**

I. Introduction

Deep Learning (DL) techniques are widely applied to solve important real-world problems, such as image and voice recognition, and autonomous driving cars. Due to the complexity and wide application of deep learning techniques, practitioners build DL libraries to make DL techniques more accessible to application developers. Modern DL applications heavily depend on popular DL libraries, such as TensorFlow [1], PyTorch [2], Theano [3] and Keras [4]. The quality assurance practice of DL libraries is critical as it affects millions of applications that are built on top of them.

However, there has been little attention to study the quality assurance practice in DL libraries (i.e., the test cases written by the developers of DL libraries). Such test cases guarantee the correctness of the implementations in DL libraries. Insufficient or low-quality test cases affect the quality of DL libraries. Furthermore, the generated models from DL libraries can be negatively impacted as well [5]. DL libraries, especially the core algorithm module, heavily encode mathematical formulas and arithmetic operations. The corresponding test cases in DL libraries have unique properties, which are not studied by prior empirical studies on test cases [6], [7].

Testing DL libraries often requires oracle approximations (OA for short) instead of definitely asserting the equality between the output from code under test (CUT) and the oracle. Oracle approximations are needed in DL libraries for various reasons. The output of the implementations in DL libraries, may slightly vary in each test run (e.g., due to randomness). For the cases where test oracles are floating numbers, it may be hard for testers to precisely define the oracle in code. Moreover, the output from CUT is allowed to be slightly different from a computed oracle, such as when oracle is other DL implementations (i.e., differential testing). Oracle approximation is an essential testing practice adopted by DL libraries in implementing assertions. Below is an example of *OA* assertion from Keras. In the code snippet, *p* is the defined oracle (line 2). The *OA* assertion (line 4) compares whether the mean of a sample from a binomial distribution (i.e., *rand*, an output from a Keras function) is close enough to the defined oracle *p*. The value that defines the accepted range is named as approximation threshold, i.e., *0.015* in the code snippet below.

```
1 def test_random_binomial(self):
2     p = 0.5
3     ...#rand gets the output from CUT, which is a Keras
          function that generates a binomial distribution
4     assert np.abs(np.mean(rand) - p) < 0.015
```

Oracle approximation opens new challenges to developers. First, developers need to select *proper* sources of oracles for *OA* assertions. Improper test oracles may introduce unstable test results, i.e., flaky tests. During code evolution, some sources of test oracles may be more fragile than the others and require more frequent updates (i.e., modifying test oracles or the accepted range of oracles). Second, oracle approximations require developers to *properly* decide the range of accepted oracles. If the range is too loose, the test case would fail to check the correctness of the implementation. If the range is too tight, the test case would constantly fail and introduce false warnings of failing tests to developers. Last, developers may encounter challenges in the maintenance and evolution of *OA* assertions. For instance, when testing the recurrent neural network (*rnn*) component in TensorFlow, developers use the output from Keras as an oracle, and implement the comparison using oracle approximation. However, in accommodating a

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00078

recent Keras update, TensorFlow developers had to modify the approximation threshold (i.e., loosening the range) to avoid constant test failures.

Hence, to better address the challenges, it is important to understand the current *OA* practices in DL libraries. In this work, we take an important first step to study *OA* practices in DL libraries. In particular, we study the practice and evolution of oracle approximations in four popular DL libraries. Our study provides both quantitative and qualitative findings to conclude common practices and highlight maintenance challenges that can inspire future research to provide better tooling support to developers in adopting and maintaining *OA* practices and better utilize oracle approximations to guarantee the quality of DL libraries.

We answer the following four research questions (RQs) by studying four popular DL libraries (i.e., TensorFlow, PyTorch, Theano and Keras).

RQ1: *How many oracle approximation (OA) assertions are implemented in testing deep learning libraries?*

We studied the prevalence of *OA* assertions in testing DL libraries. We investigated the commonly-used APIs that developers use when implementing *OA* assertions. We find that there exists a non-negligible portion of OA assertions (i.e., 5% to 24%) in the four studied DL libraries.

RQ2: *What are the common types of test oracles and thresholds used in* OA *assertions?*

We performed a manual study to identify the common types of test oracles used in *OA* assertions. We derived a systematic labeling system to guide this categorization process. Our study shows that a diverse set of oracle types are used in *OA* assertions. Developers often use computation-based oracles (27%-72%), such as output from other DL libraries, or output from similar functions in the same DL library. Moreover, through analyzing the thresholds used, we find that developers often need to specify thresholds used instead of using the default ones provided by *OA*-assertion APIs.

RQ3: *What are code changes developers perform on* OA *assertions?*

Our study shows that developers frequently modify *OA* assertions because of code evolution. On average, 23% of the modifications are about using a different *OA*-assertion API, 18.0% are about tightening or loosening the thresholds, and 58.9% are about modifying CUT or test oracles.

RQ4: *Why developers perform changes on* OA *assertions?*

We performed an in-depth analysis on all the *OA*-related commits of one test module in TensorFlow, i.e., under *kernel_test* directory. In total, we analyzed 71 commits on *OA* assertions in-depth, including analyzing the code changes in the commits, analyzing other relevant code, reading the commit messages and executing test cases before and after the commits. Our findings reveal maintenance challenges developers may face when managing *OA* assertions.

Paper Organization. Section II discusses the background of oracle approximations. Section III describes the approaches we follow to answer the four RQs. Section IV presents the answers to the four RQs. Section V lists both internal and external threats. Section VI surveys the related work. Finally, Section VII concludes the paper.

II. Background on Oracle Approximations

In this section, we describe the background on oracle approximations, especially the reasons that *OA* assertions are necessary in DL libraries. There exists a non-negligible difference between the true value of a mathematical function and the value that is calculated by the code implementation of the mathematical function. We call such non-negligible difference a *numerical error*. The implementation of DL algorithms consists of encoding mathematical formulas and arithmetic operations in programming languages. Therefore, DL libraries inevitably contain such numerical errors. Such numerical errors will be represented by oracle approximations in test cases.

Atkinson and Han (*Elementary numerical analysis* [8]) categorize five common numerical errors in arithmetic computations. Below we describe the five types of numerical errors in detail and their relevance to DL libraries: the first three types of errors contribute to the analyzed oracle approximations in this work and the last two types do not.

- Machine representation error. When representing floating numbers in computer, there exists a difference between the true value and the value represented by computers. Due to the limited representation and use of rounding techniques, this type of error widely exists in software that contains floating-number computation. DL libraries are no exception. Some of the *OA* assertions studied in this paper are related to this type of error.
- Mathematical approximation error. Errors occur because computers use estimation and approximation algorithms in the process of calculating certain functions, such as differential equations and trigonometric functions. Such estimations introduce numerical errors. An example of such approximation function in DL libraries is gradient function over tensors. Our study includes the *OA* assertions that are caused by this type of error.
- Defects in implementations. This source of error is related to developer error in the process of calculation. Developers may introduce implementation defects in DL libraries, which may lead to severe numerical errors in any generated DL models. The *OA* assertions in our study could be related to this type of error.
- Modeling error. This type of error occurs when there are uncertainties in the modeled mathematical relation of a phenomenon. For instance, in DL libraries, there might be some unspecified implementation details in the distribution definition of the learning algorithms. This type of error does not contribute to our studied *OA* assertions.
- Physical measurement error. There might exist a difference between the physical reality of a phenomenon and the measured value. Such errors are typically introduced in data collection process and do not exist in DL libraries.

Therefore, the analyzed oracle approximations in this work are not related to this type of error.

III. Methodology

In this section, we provide details on the studied four DL libraries and describe the methodology we used to conduct the study on oracle approximations to answer the following four research questions (**RQs**):

RQ1: How many oracle approximation (*OA*) assertions are implemented in testing deep learning libraries?

RQ2: What are the common types of test oracles and thresholds used in *OA* assertions?

RQ3: What are code changes developers perform on *OA* assertions?

RQ4: Why developers perform changes on *OA* assertions?

A. Studied Systems

TABLE I
STATISTICS ON THE STUDIED DEEP LEARNING LIBRARIES. KERAS IS A PYTHON LIBRARY. THERE IS CODE OF OTHER LANGUAGES (E.G., C++) IN TENSORFLOW, THEANO AND PYTORCH.

System	Version	Release Date	KLOC (Python)	Studied Dev. Period
TensorFlow	1.12.0	09/2018	2,003	11/2015 – 09/2018
Theano	1.0.3	09/2018	667	01/2008 – 09/2018
PyTorch	1.0.0	12/2018	217	01/2012 – 12/2018
Keras	2.2.4	10/2018	65	03/2015 – 10/2018

In total, we include four deep learning libraries in the study, i.e., TensorFlow [1], Theano [3], PyTorch [2] and Keras [4]. The selected four systems are commonly cited as the most popular DL libraries, e.g., the most starred and forked DL projects on GitHub and most cited framework on ArXiv and Medium.[1] Table I shows the statistics of the studied DL libraries. For RQ1 and RQ2, we analyzed a recent stable version of each deep learning library (see "Version" in Table I). For RQ3 and RQ4 (i.e., two RQs about code evolution), we analyzed a development period for each library (see "Studied Dev. Period") in Table I). The analyzed development history starts from the first commit of each studied DL library.

Among the four analyzed systems, Keras is the only project that is in pure Python, because there is no computational core in Keras structure. However, TensorFlow, PyTorch and Theano have a C/C++ core and Python API as their default interface [2]. In these three frameworks, there is a non-negligible portion of non-python core code, i.e., 1,290 KLOC C++ code in TensorFlow and 423 KLOC C++ code in PyTorch and 27 KLOC C code in Theano. Our study focuses on Python test cases (i.e., test cases written in Python) and does not include C/C++ test cases. However our study actually includes a comprehensive set of test cases that test non-python code. Based on TensorFlow[3] and PyTorch documentation[4], Python has been mentioned as the preferable API for development and testing. In TensorFlow and PyTorch, it remains as a common practice to use Python wrappers to wrap C++ code (e.g., the computation core in TensorFlow [9]), and further test the Python wrappers using Python test cases. Note that wrapping C/C++ code in Python, there may or may not be new functionalities (e.g., computation, data processing) being added in the Python code.

In short, although we focus on test cases written in Python, our study actually includes test code for testing both Python and C++ code in TensorFlow and PyTorch since their C++ code is tested by Python test cases. In the future, we plan to expand the study of oracle approximations to other programming languages (e.g., Java and C++) by analyzing systems such as Microsoft CNTK, Deeplearning4j and Caffe.

B. Identifying OA Assertions (RQ1 and RQ2)

First, we identify the assertion methods that developers use in the studied libraries to express *OA* assertions. Then we extract *OA* assertions by finding the usages of the identified *OA*-assertion methods. In particular, we parse every test method in Python into an abstract syntax tree (AST), iterate the AST and identify the assertion statements that use one of the identified *OA*-assertion methods.

In Python, assertions are expressed in two ways: 1) *assert* keyword, which is then followed by a boolean expression; and 2) using customized assertion APIs, e.g., internally defined by each Python project, the Python *unittest* built-in functions, and assertion APIs provided by NumPy, i.e., a commonly-used library in Python that supports computation on arrays and matrices. Inherited from the above-mentioned two ways, oracle approximation assertions can be expressed using 1) *assert* keyword, which is followed by a boolean expression that performs relational operations, particularly $<$ and $<=$, and 2) assertion functions that allow the expression of oracle approximations, i.e., *OA* assertion APIs. Note that for 1), we do not include notations such as $>$ and $>=$ because we find that these greater notations are often about about inequality instead of approximations. Differently we find that $<$ and $<=$ are often used to express oracle approximations such as *absolute(oracle - output)* $<$ *0.05*.

Our procedure of identifying *OA*-assertion APIs for each studied library combines reading the official documentation of the libraries (i.e., TensorFlow, PyTorch, Theano and Keras) to initiate regular expression queries such as *assert*close*, *assert*almost*, searching the initiated queries in each codebase, and expanding the list of *OA*-assertion APIs by examining the search results. The steps in our procedure is performed

[1]For more information on Metrics and results of Deep Learning framework popularity analysis please read https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

[2]For TensorFlow there are other API in other languages such as Java and Go but we do not cover them in this study.

[3]In documentation of TensorFlow, it has been mentioned that for testing new customized operation, "We usually do this [operation testing] in Python for convenience". Therefore, testing the computational core, which is in C++, is actually implemented in Python test cases. https://www.tensorflow.org/guide/extend/op

[4]In PyTorch's README it has been explicitly mentioned that "PyTorch is not a Python binding into a monolithic C++ framework. It is built to be deeply integrated into Python".

iteratively until we are not able to find any new *OA*-assertion APIs in one studied library. The two authors of this work confirmed that the resulting list of APIs indeed expresses the oracle approximations. Our manual analysis in RQ2 confirmed that the approach and heuristics we adapt to identify *OA* assertions are accurate: more than 99% of the identified *OA* assertions are *true OA* assertions.

C. Identifying Oracle Type in Oracle Approximation Assertions (RQ2)

Test oracles are defined differently, e.g., a defined oracle, or output from an external library using differential testing [10]. Understanding the commonly used oracle types in oracle approximation assertions would inspire future research to design better tooling support when oracle approximations are needed in assertions. For example, differential testing may use oracle approximations to allow a slight difference between one function from TensorFlow and a similar function from Keras. Developers would also need to specify the maximum value of such slight difference. If the maximum allowance is set too large, then the test case is weak in detecting regressions. If the maximum allowance is set too small, then it may require frequent changes as code evolves.

We manually examined the oracle type for every oracle approximation assertion in a statistically significant sample from each of the studied system. In particular, we analyzed 348 *OA* assertions from TensorFlow, 219 from PyTorch, 281 from Theano, and 180 from Keras. All the samples are taken with a confidence level of 95% and a confidence interval of 5%.

To derive a systematic labeling, two of the authors independently perform analysis on a set of 100 *OA* assertions from TensorFlow and take notes about oracle types. The two authors then discussed and agreed on a systematic labeling that consists of five main categories: manually defined (`D`), differential testing (`DT`), a naive implementation (`NI`), a relevant internal function (`RI`), and the same code under test (`SCUT`). Examples of the five categories will be provided when we present the results in Section IV-B.

We use the following approach to perform the systematic labeling process. Given one oracle approximation assertion, the following steps are used to label the oracle type:

1) ***Identify the argument that represents a test oracle.*** For all the approximation APIs in Table II except for the category of *assert* and relational operations, developers are expected to place the output from the code under test (*CUT*) as the first argument and the test oracle as the second argument. However in practice, developers may switch the placement of the two arguments. Therefore, we need to decide which argument is representing the test oracle by identifying the argument represents the output from CUT by understanding the test code.
2) ***Decide if the oracle is defined or through computation.*** If the oracle is defined or assigned with numerical values (e.g., *np.abs(np.mean(rand)- 0.5) < 0.015* where 0.5 is the defined oracle), we labeled the oracle type as a defined oracle (`D`). Developers may perform computation on the defined numerical values (e.g., calculating the mean of a data set) and use the result as a test oracle instead of using the raw values. For such cases, we still labeled them as `D` oracle type.
3) ***Given an computation oracle, decide whether the computation code is internal or external.*** If the oracle is output from an external library (e.g., using NumPy's result when testing TensorFlow), we labeled the oracle type as differential testing (`DT`).
4) ***Classify the oracle type into one of the three types (i.e., `NI, RI, SCUT`) when the computation code is internal.*** Developers may provide a naive implementation (i.e. often in the test code) and use the result from the naive implementation as a test oracle. The naive implementation (`NI`) is similar to the code under test in terms of functionality but may lack of complexity and provide limited accuracy. Oracles of relevant internal code (`RI`) refer to the oracles that are computed by a relevant function in the same codebase (e.g. comparing the output from two methods in TensorFlow). We also find assertions in which developers use the output of the same code under test (`SCUT`) as the test oracle. Among the `SCUT` cases, when used as test oracles, the same code under test may or may not be invoked with the exactly same inputs and configurations. We also label `SCUT` cases with a sub-category label based on this distinction.

The systematic labeling is comprehensive as it covers whether the oracle is defined (`D`) or through computation (`NI`, `DT`, `RI`, and `SCUT`). Among the oracles from computation, the labeling covers all the cases where the computation is from: internal (`NI`, `RI`, `SCUT`) and external (`DT`).

Upon agreeing on the systematic labeling, the two authors *independently* label the oracle types for the randomly sampled *OA* assertions and resolved the disagreements through discussions. The two authors reached a Cohen's kappa of 0.77 in this systematic labeling process, which is a substantial-level of agreement.

D. Categorizing the modifications on oracle approximation assertions (RQ3)

In RQ3, we investigate what are the changes developers perform on *OA* assertions and the distribution of such code changes. We developed an automated solution to answer RQ3. The tool firstly extracts all the changes of *OA* from commit history. Each OA modification consists a pair of deleted and added lines of code, both of which are on *OA* assertions. After extracting the modifications, the tool determines the category of each modification.

To extract OA modifications (i.e., a pair of added and deleted lines), given one commit, we utilized the Abstract Syntax Trees (ASTs) that are generated from statements that differ in pre- and post-commit versions. Tokens in *OA* assertions are aligned for equality comparison and the top-matched *OA* statements are extracted as one OA modification.

In the second step, the tool decides the category of OA modification by first deciding whether cosmetic or non-cosmetic modifications, then further dividing the non-cosmetic modifications into three categories, i.e., API modification, oracle/CUT modification, and threshold modification. Cosmetic modifications are changes that *only* contain tabs, spaces and newlines. We identify cosmetic changes by asserting whether one added and one deleted lines are identical after removing spaces, tabs and newlines. For non-cosmetic modifications, the tool uses the syntactic position of the modified token in AST to decide whether the modification is to change API, oracle/CUT, or threshold. For *API modification*, the arguments of the added and deleted lines are the same but the assertion API has been changed. For *oracle/CUT modification*, the first two arguments of an *OA* assertion are modified. Due to the fact that developers might misplace oracle and CUT, we are not able to precisely distinguish the oracle change from CUT change automatically. For *threshold modification*, developers may change the threshold used to control the comparison between the oracle and the output from CUT, i.e., tightening the comparison by using a smaller threshold, or loosening the comparison by using a larger threshold.

E. Understanding the reasons behind OA *assertions modifications through manual analysis (RQ4)*

Understanding the context and reasons that developers perform modifications on *OA* assertions will provide insights to provide better tooling support to developers in maintenance activities.

To answer this RQ, we performed an in-depth analysis on all the commits on one computation component in TensorFlow (i.e., the *kernel_tests* directory). In particular, we categorized the commits based on the effects of code changes and reasons behind. The kernel_tests directory in TensorFlow contains all the test cases that test tensor operation functions [5]. Focusing on one directory allows us to conduct an in-depth analysis since such analysis requires extensive understanding and knowledge on the test suites and source code in the study. We leveraged the labels of RQ3 in the first manual study of RQ4. The tool we developed for answering RQ3 can label a commit as a cosmetic or a non-cosmetic change. We examined the cosmetic changes one-by-one to make sure all are indeed cosmetic changes. For non-cosmetic changes, different from RQ3 labels, we further analyze the effect and possible reasons behind the change. We combined reading commit messages, source code and test file changes to understand the nature and root causes of the commits. Furthermore, we reproduced some commits of each category (i.e., executing test cases in the versions of pre- and post-commit) to better understand the rationale of code changes performed by developers.

IV. Results

This section presents the answers to the four research questions.

[5]More information about operations in TensorFlow can be found at https://www.tensorflow.org/api_docs/python/tf/Operation.

A. RQ1: How many oracle approximation (OA) assertions are implemented in testing deep learning libraries?

We followed the procedure that is described in Section III-B to identify the oracle approximation assertion APIs used in each of the studied DL library (see Table II). Then, we expanded the API list to include all the assertion APIs used in each DL library. Last, we counted the number of all assertions and the number of *OA* assertions for each DL library.

Table II presents the *OA*-assertion APIs used in each studied DL library, the number of each API usage and also the source of the API (i.e., in which library the API is defined). We categorize the *OA*-assertion APIs into four main categories. The APIs of "absolute and relative tolerance" use both absolute and relative thresholds to express the accepted range of oracles. Whether the assertions using these APIs fail or not depends on the result of `assert abs(CUT - oracle) < atol + rtol * abs(oracle)`, in which `CUT` is the output from code under test, `atol` is absolute tolerance, and `rtol` is relative tolerance. The second category of OA APIs uses absolute tolerance only, for which the following condition is used to determine the result `assert abs(CUT - oracle) < atol`. The third category of *OA*-assertion APIs use rounding to assert the closeness between CUT and oracle based on `assert abs(oracle - CUT) < 1.5 * 10**(-decimal)`. The argument `decimal` is used to specify the number of significant digits in CUT and test oracle. The last category of APIs (i.e., "error bounding") do not directly compare CUT and oracle, but first calculates the difference between the two, and then asserts whether the difference (i.e., *error*) is smaller than a threshold such as `assert error < threshold`.

Our study shows that the *OA*-assertion APIs from Numpy are commonly used across the studied DL libraries. Moreover, in each DL library, developers may define library-specific APIs to express *OA* assertions (e.g., 80% of all the *OA* assertions in TensorFlow). Both TensorFlow and PyTorch use the Python built-in assertion APIs that are defined in *unittest.TestCase* (16% in TensorFlow and 34% in PyTorch).

Table III shows the results of RQ1. In particular, we show the number of *OA* assertions, the number of all assertions, the number of test cases with at least one oracle approximation assertion, and the number of all test cases. In summary, there exists a non-negligible portion of *OA* assertions in each studied DL library, ranging from 5% to 25%. If considering the test cases, 10% to 67% of all the test cases contain at least one oracle approximation assertion.

We find that there exists a non-negligible portion (5%-25%) of OA *assertions in DL libraries. Assertion APIs from Numpy are commonly used in TensorFlow, PyTorch, Theano and Keras to express* OA *assertions. Developers may heavily use customized functions to express oracle approximation assertion (i.e., 80% in TensorFlow).*

TABLE II
APIs THAT EXPRESS ORACLE APPROXIMATIONS IN EACH OF THE STUDIED DL LIBRARIES.

Category	Source	Methods	TensorFlow	PyTorch	Theano	Keras
Absolute and Relative Tolerence	Numpy	assert_allclose(actual, desired, rtol, atol)	74	119	598	231
	Numpy	assert allclose(actual, desired, rtol, atol)	2	9	358	29
	Numpy	assertTrue(isclose(actual, desired, rtol, atol))	8	54	40	0
	Keras	assert_list_pairwise(A, atol)	14	0	0	24
	TensorFlow	assertAllClose(actual, desired, rtol, atol)	2550	0	0	0
	TensorFlow	assertAllCloseAccordingToType(actual, desired, rtol, atol)	267	0	0	0
Absolute Tolerence	TensorFlow	assertNear(actual, desired, atol)	48	0	0	0
	TensorFlow	assertArrayNear(actual, desired, atol)	32	0	0	0
	TensorFlow	assertNDArrayNear(actual, desired, atol)	1	0	0	0
Rounding Tolerance	unittest	assertAlmostEqual(actual, desired, decimal)	300	88	2	0
	unittest	assertAlmostEquals(actual, desired, decimal)	0	2	0	0
	Numpy	assert_array_almost_equal(actual, desired, decimal)	0	39	16	5
	Numpy	assert_almost_equal(actual, desired, decimal)	1	76	0	5
Error Bounding	Python	assert error $<$ threshold	19	36	33	44
	unittest	assertTrue(A $<$ B)	16	9	0	0
	unittest	assertLess(A,B)	230	35	0	0
	unittest	assertLessEqual(A,B)	24	38	0	0
	Numpy	assert_array_less(A,B)	3	2	0	0
	TensorFlow	assertAllLessEqual(A,B)	0	0	0	0

TABLE III
STATISTICS OF OA ASSERTIONS IN THE STUDIED DL LIBRARIES.

Subject	Assertions	OA Assertions (perc.)	Test Cases	Test Cases w/ OA (perc.)
TensorFlow	21,776	3589 (16%)	2,858	2,018 (70%)
PyTorch	10,049	507 (5%)	3,555	313 (8%)
Theano	5,995	1,047 (17%)	1,972	592 (30%)
Keras	1,351	338 (25%)	582	161 (28%)

B. RQ2: What are the common types of test oracles and thresholds used in OA *assertions?*

We manually analyzed 1,070 *OA* assertions from the studied DL libraries, i.e., TensorFlow, PyTroch, Theano and Keras. The 1,070 instances consist of random samples from all the *OA* assertions in the four studied libraries. Each random sample from each library achieves a confidence level of 95% and a confidence interval of 5%. We labeled the oracle type in each oracle approximation assertion following the systematic labeling that is described in Section III-C. Also, we recorded whether threshold that controls the approximation assertion is either *default*, which means developers do not specify a certain threshold and use the default value in the assertion API, or *customized*, or *depend*, which means the exact value used as the threshold will be decided during runtime (i.e., depending on data types or hardware specifications).

Table IV shows the results from our manual labeling on oracle types and approximation thresholds. The systematic labeling contains five main categories.

Defined oracle (D). Developers often define the oracle (e.g., a floating point number) and assign the defined to a variable that is then used in assertions. In some cases, developers may perform simple calculations on the defined oracles, and such calculations are irrelevant to the code under test. For such cases, we also label them as defined oracle type. We show an example of defined oracle from TensorFlow below. In the example below, p is assigned with an array (i.e., using NumPy, line 3) and is compared with the output from code under test in line 5. Our study shows that 26%-73% of *OA* assertions use defined test oracles.

```
1 @test_util.run_in_graph_and_eager_modes
2 def testLogits(self):
3   p = np.array([0.01, 0.2, 0.5, 0.7, .99], dtype=np.
        float32)
4   ...
5   self.assertAllClose(p, self.evaluate(new_p), rtol=1e-5,
        atol=0.)
```

Naive implementation (NI). The oracles are computed by a naive implementation, which is often defined in test code. The naive implementation provides a similar functionality with the code under test. What differentiates NI from D is that defined oracles may involve computation, but the computation is irrelevant to the code under test. In the code snippet below, the test oracle *kl_expected* is computed through the simple implementation (line 5 to line 9).

```
1 kl = kullback_leibler.kl_divergence(a, b)
2 kl_val = sess.run(kl)
3
4 #provide a naive implementation to get kl_expected
5 a_logits = np.random.randn(batch_size, categories)
6 b_logits = np.random.randn(batch_size, categories)
7 prob_a = np_softmax(a_logits)
8 prob_b = np_softmax(b_logits)
9 kl_expected = np.sum(prob_a * (np.log(prob_a) - np.log(
      prob_b)), axis=-1)
10
11 self.assertAllClose(kl_val, kl_expected)
```

Differential testing (DT). DT is commonly used to test systems when there exist a variety of implementations, such

TABLE IV
ORACLE TYPES AND THRESHOLDS USED IN THE *OA* ASSERTIONS OF THE STUDIED DL LIBRARIES.

Subject	Defined Oracle (D)	Naive Impl. (NI)	Differential Testing (DT)	Relevant Internal Function (RI)	Same Code Under Test (SCUT)		Threshold		
					Same Input	Different Inputs	default	customized	depend
TensorFlow	257 (73%)	23 (7%)	19 (5%)	14 (4%)	16 (5%)	5 (1%)	107 (31%)	226(65%)	15(4%)
PyTorch	93 (42%)	36 (16%)	24 (11%)	18 (8%)	2 (1%)	34 (16%)	42 (19%)	146(67%)	18(8%)
Theano	75 (27%)	44 (16%)	76 (27%)	55 (20%)	6 (2%)	11 (4%)	242 (86%)	18 (6%)	6 (2%)
Keras	72 (40%)	12 (7%)	14 (8%)	21 (12%)	27 (15%)	23 (13%)	68 (38%)	102 (57%)	10 (6%)

as compilers [11]. In the DT cases of testing DL libraries, the oracles are computed from similar code in other DL libraries. The code snippet below shows an example of DT that compares the output from Theano's *tensordot* function (line 1) with *numpy.tensordot* (line 5).

```
1 c = tensordot(avec, bmat, axes)
2 f2 = inplace_func([avec, bmat], c)
3 aval = rand(5)
4 bval = rand(8, 5)
5 utt.assert_allclose(np.tensordot(aval, bval, axes), f2(
      aval, bval))
```

We find that 5%-27% of *OA* assertions utilize differential testing, i.e., using outputs of other libraries as test oracles. Particularly in Theano, there exists a significant portion (27%) of *OA* assertions that use differential testing oracles, and the percentage is much higher than the other DL libraries. The relevancies inferred from differential testing practices can be used to improve test cases for both libraries invovled.

Relevant internal function (RI). Different from DT, RI refers to the cases where the oracles are produced by relevant functions in the same codebase under test. The relevant functions, when invoked with certain arguments, can be used to generate the oracles that are then compared with the code under test. Below is an example of RI category from TensorFlow. In particular, *y1* (line 1) and *y2* (line 2) are computed from two relevant functions in Tensorflow and then be compared for similarity (line 4).

```
1    y1 = nn_ops.atrous_conv2d(y1, f, rate, padding=padding
         )
2    y2 = nn_ops.conv2d(y2, f, strides=[1, 1, 1, 1],
         padding=padding)
3    y2 = array_ops.batch_to_space(y2, crops=pad,
         block_size=rate)
4    self.assertAllClose(y1.eval(), y2.eval(), rtol=1e-2,
         atol=1e-2)
```

In three of the four studied DL libraries, *RI* does not take a significant portion, i.e., 4%–12% of all *OA* assertions. Differently in Theano, *RI* is a much larger portion, i.e., 20%.

Same code under test (SCUT). In some cases, test oracles may be produced by invoking the same code under test, with either the same inputs or different inputs. In DL libraries, the possible reasons include metamorphic testing (i.e., two different inputs may yield the same result from the same code under test), or testing the randomness (i.e., two runs of the same under test with the same input may produce slightly different results). Thus, we derive two sub-categories under the category of SCUT based on whether the two sets of inputs (i.e., arguments) are the same or not. In total, we labeled 6%-27% of the *OA* assertions with the oracle type of SCUT. Among them, 41% use the same input to generate the oracles and 59% use different inputs to generate the oracles. We have found that in the SCUT cases with the same input, the purpose of the developer is to assert whether an additional operation on CUT can cause non-tolerable error or not. Differently, in SCUT cases with different inputs, the concept is analogous to metamorphic testing. Such information potentially can be leveraged to improve automated test generation techniques in the future, e.g., using the inferred metamorphic relations.

```
1 def test_statefulness_GRU(self):
2   model = keras.models.Sequential()
3   ...#right_padded_input is being modified
4   out6 = model.predict(left_padded_input)
5   ...#left_padded_input is being modified
6   out7 = model.predict(right_padded_input)
7   np.testing.assert_allclose(out7, out6, atol=1e-5)
```

Table IV also shows how many of the analyzed cases use *default*, *runtime*, and *customized* thresholds to restrict the approximations. Our study shows that developers often need to specify a proper threshold when using oracle approximations, i.e., ranging from 6% to 67% in the studied systems. For 2% to 8% studied cases, developers need to use different thresholds in the same oracle approximation assertion based on runtime behaviors, such as different data types (e.g., float16, float32 etc.), and whether CPU or GPU is configured in the test environment.

Our study shows that there exists a diverse set of different test oracles and thresholds used in OA *assertions. Our study also shows that there exists a significantly portion (27%-72%) of test oracles are through computation. This indicates that applying automated non-oracle testing techniques (e.g., metamorphic testing, differential testing) should carefully consider and utilize oracle approximation practices.*

C. RQ3: What are code changes developers perform on OA assertions?

We developed an automated solution to answer this RQ in Section III-D. In particular, the approach firstly determines whether a commit is to add OA assertions, to delete the OA assertions, or to modify the OA assertions. Among the modifications, the approach classifies whether modifications is on *OA*-assertion API, the parameters (CUT/oracle), or the threshold.

Table V shows the distribution of oracle approximation assertion commits on the above-mentioned categories. 7%-20% of all the commits involve changes at least one oracle approximation assertion. The remaining rows show the number

TABLE V
BREAKDOWN OF CODE CHANGES ON *OA* ASSERTIONS

	TensorFlow	PyTorch	Theano	Keras
Total # of commits	6,915	4,427	10,350	1,283
Total # of commit (OA)	1,062 (15%)	344 (8%)	776 (7%)	254 (20%)
The numbers below show the numbers of changes in each category.				
Total # of changes (OA)	3,168	1,022	2,801	1,169
# of OA additions	870 (27%)	765 (75%)	1229 (44%)	526 (45%)
# of OA deletions	503 (16%)	115 (11%)	224 (8%)	178 (15%)
# of cosmetic modi.	863 (27%)	61 (6%)	397 (14%)	179 (15%)
# of non-cosmetic modi.	932 (29%)	80 (8%)	951 (34%)	286 (24%)
... # of API modi.	202	10	248	58
... # of para. modi.	522	55	639	109
... # of thres. modi.	208	15	64	119

of changes in each category. One commit may consist of multiple changes. Among all the changes on OA assertions, 27%-75% are adding new OA assertions, e.g., increasing the test coverage, or modifying non-OA assertions to OA assertions. 8%-16% are deleting existing OA assertions. The remaining changes (14%-56%) are OA modifications. Among the OA modifications, after removing cosmetic modifications (i.e., 6%-27%), the non-cosmetic modifications are further divided into three categories: API modification (13%-26% on all non-cosmetic changes), parameter (CUT/oracle) modifications (38%-69%), and threshold modifications (7%-42%).

Figure 1 shows three examples of the three detailed modification categories. The first code snippet is an example of API modification from PyTorch. In this example, developers modified the *OA*-assertion API, i.e., using *assertLess* instead of *assertLessEqual* while keeping the other arguments unchanged. The second code snippet is from Keras, in which both of the CUT and oracle are modified. In this case, developers changed used a specific property of CUT and oracle to compare for faster test execution. The last code snippet is from TensorFlow, and the thresholds (i.e., both the absolute tolerance and relative tolerance) are modified in order to reduce the flakiness of this assertion. With the smaller thresholds, i.e., before this change, the assertion is more likely to fail. [6]

```
- self.assertLessEqual(fn().data[0], initial_value)
+ self.assertLess(fn().data[0], initial_value)
```

An example of API modification from PyTorch

```
- assert_allclose(out, out2, atol=1e-05)
+ assert_allclose(np.squeeze(out), np.squeeze(out2), atol
    =1e-05)
```

An example of CUT/oracle modification from Keras

```
- self.assertAllClose(sample_mean_, analytic_mean, atol
    =0., rtol=0.06)
+ self.assertAllClose(sample_mean_, analytic_mean, atol
    =0.04, rtol=0.)
```

An example of threshold modification from TensorFlow

Fig. 1. Examples of the three modification categories (API, CUT/oracle, and threshold).

[6]According to the commit message, the assertion is flaky and developers decided to loosen the thresholds to avoid any failures.

We find that developers frequently change OA *assertions, i.e., 7%-20% of all the commits. Among the code changes on* OA *assertions, developers often perform non-cosmetic modifications on* OA *assertions, i.e., up to 34% of all changes. Developers can benefit from future tooling support to help them better manage code evolution on* OA *assertions.*

D. RQ4: Why developers perform changes on OA assertions?

In RQ3, we show that many code changes on *OA* assertions are to modify oracle or threshold. In RQ4, we analyzed all the commits on oracle approximations in one computation module in TensorFlow to understand the reasons behind those changes.

Table VI shows the results of our manual study on *OA*-related commits in *kernel_tests* directory in TensorFlow. Below we describe detailed analysis of each category and provide some examples.

TABLE VI
THE REASONS BEHIND CODE CHANGES OF TEST CASES IN KERNEL_TESTS DIRECTORY OF TENSORFLOW

Reasons of Modifications	Commits	Changes
Refactoring	8	46
Increasing test coverage	5	123
Threshold loosening cases	10	34
Threshold tightening cases	1	1
Data-type related modifications	14	26
Hardware related modifications	3	5
Rollbacks	4	25
Unknown	14	18

Refactoring. Developers may perform refactoring to improve test cases, e.g., reducing the test execution time without sacrificing coverage. We call such cases refactoring as they do not change the structure of test cases significantly. Below is a refactoring example from TensorFlow. The assertion (line 4) is modified following the change on a variable assignment (line 2). In this case, developers change the size of the CUT variable (i.e., from 1024 to 128) to "reduce the cost" of running this test. This refactoring change does not affect the threshold. Therefore, although the oracle is changed, the threshold remains unmodified.

```
1 -   v = 2. * (array_ops.zeros([1024, 1024]) + x)
2 +   v = 2. * (array_ops.zeros([128, 128]) + x)
3 ...# yval is computed from v
4 -   self.assertAllClose(4 * (i - 1) * (i - 1) * 128, yval
      , rtol=1e-4)
5 +   self.assertAllClose(4 * (i - 1) * (i - 1) * 1024,
      yval, rtol=1e-4)
```

Increasing test coverage. In the development of DL libraries, developers may add new code to provide new functionalities or improve the current features. Correspondingly, test cases should be updated to test the new code. Developers may modify existing test cases to test new code by increasing the test coverage instead of adding brand new test cases.

Threshold modification. We describe the categories under "threshold modifications" in detail. We find that There are four reasons that developers need to perform threshold modifications.

- Hardware-related modifications. The output from CUT may produce different results depending on whether GPU or CPU is configured. Often developers implement the two test scenarios in one test method and use different thresholds to differentiate the comparison with the same oracle. For example, as code evolves, developers may add GPU support to certain functions, the corresponding test cases are subject to change as well. In total, we find that in 3 commits, developers change threshold to accommodate testing in GPU environment. Below is an example of hardware-related threshold modification. The commit is from TensorFlow and the commit message is " Implement GPU version of tf.determinant". Developers modified the configuration of the test case to include testing GPU (line 8). Correspondingly, the invoked test method *_compareDeterminantBase* (line 2) would need to modify the threshold to accommodate such change. In this example, developers loosened the threshold (i.e., from the default value 1e-6 to 5e-5).

```
1  Message :  Implement GPU version of tf.determinant.
2  def _compareDeterminantBase(...):
3  -   self.assertAllClose(np_ans, out)
4  +   self.assertAllClose(np_ans, out, atol=5e-5)
5
6  def _compareDeterminant(...):
7  -   with self.test_session():
8  +   with self.test_session(use_gpu=True):
9         self._compareDeterminantBase(matrix_x,
10           linalg_ops.matrix_determinant(matrix_x))
```

- Data representation related modifications. Computation on floating-numbers suffers from the imprecision of representing floating-numbers in machines. Therefore, the output from CUT in DL libraries may return slightly different values in the same test method because of different data representations, e.g., float16, float32, etc. In the code example below, developers changed the *assertAllClose* to *assertAllCloseAccordingToType* in test *UnsortedSegmentSumTest*. The latter *OA*-assertion API has different default threshold values for different data types. So developers use this OA-assertion API to express different levels of approximation for different data types.

```
1  -    self.assertAllClose(np_ans, tf_ans)
2  +    self.assertAllCloseAccordingToType
3           (np_ans, tf_ans)
```

  We reproduced this commit, logged the value of `tf_ans`, executed the test case 10 times, and concluded that *UnsortedSegmentSum* indeed generates different values for different data types.
- Loosening the threshold. In total we reproduced and analyzed 10 commits in this category. When one *OA*-assertion fails, developers may choose to relax the threshold. The reason was reflected in many commit messages that contains words such as "flaky test" or "flakiness". For instance, the commit message of the code change below is "Increase tolerance in flaky multinomial test, but there is a problem with multinomial distribution test and their solution is to make the function less sensitive to system error". To void such flaky tests, developers may loosen the threshold, i.e., *atol* is changed from 0 to 0.01 in the code snippet below.

```
1  -  self.assertAllClose(sample_mean_, analytic_mean,
       atol=0., rtol=0.01)
2  ...
3  +  self.assertAllClose(sample_mean_, analytic_mean,
       atol=0.01, rtol=0.01)
```

Below we show another example in the category of loosening threshold. Developers modified totally 15 invocations of *assertAllClose* and used loosened thresholds to eliminate the "flakiness" of the tests. The commit message is "Reduce flakiness of tf.distributions tests by tweaking the tolerances.".

```
1  -   self.assertAllClose(sample_mean_,
       analytic_mean, atol=0., rtol=0.06)
2  -   self.assertAllClose(sample_cov_, analytic_cov,
       atol=0., rtol=0.07)
3  -   self.assertAllClose(sample_var_, analytic_var,
       atol=0., rtol=0.07)
4  -   self.assertAllClose(sample_stddev_,
       analytic_stddev, atol=0., rtol=0.02)
5
6  +   self.assertAllClose(sample_mean_,
       analytic_mean, atol=0.04, rtol=0.)
7  +   self.assertAllClose(sample_cov_, analytic_cov,
       atol=0.05, rtol=0.)
8  +   self.assertAllClose(sample_var_, analytic_var,
       atol=0.05, rtol=0.)
9  +   self.assertAllClose(sample_stddev_,
       analytic_stddev, atol=0.02, rtol=0.)
```

We reproduced the commit above and logged relevant values to show the flakiness before the commit, and also how the flakiness is reduced after the commit. There are four *OA* assertions in the code example above. The *OA*-assertion API *assertAllClose* uses both relative and absolute tolerances to formulate the accepted difference between an output of CUT and an oracle. To visualize how close the *OA* assertions are to test failures, we formulated *safe_margin* function as `safe_margin(OA) = (atol + rtol * oracle) - abs(CUT - oracle)`, where *atol* and $rtol$ are respectively absolute and relative tolerance, and CUT and $oracle$ represent the output from the CUT and test oracle. If the $safe\_margin$ value becomes smaller than zero, then there will be an unacceptable difference between *CUT* and *oracle* and therefore, the assertion will fail.

Figure 2 visualizes the histogram of the $safe\_margin$ values in 10 runs of the four invocations of *assertAllClose*. The figure includes two sets of data: one for the pre-commit version (red) and one for the post-commit version (green). The unsafe margins are highlighted in gray, which means that if any of the red or green lines falls into the gray area, the corresponding assertion would fail. Figure 2 shows that after the commit, i.e., loosening the threshold, two properties of the safe margin

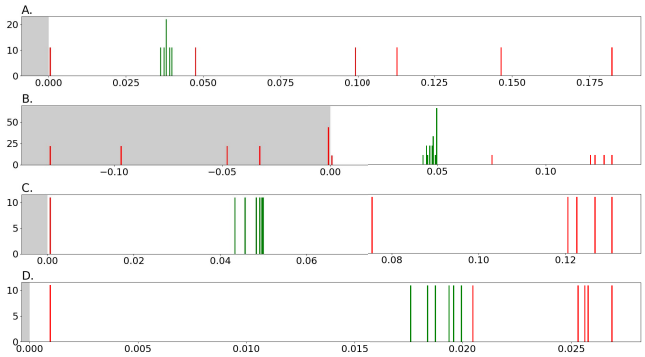

Fig. 2. The safe margins of the four OA assertions in two versions: before (red) and after (green) the threshold modification commit. The gray area is where the test would fail if any sage margin value falls into it.

distributions are changed. First, none of the post-commit values falls into the range of test failures (i.e., the gray block on the left). Second, the variance of post-commit values is lower (i.e., less scattered in the figure), because the developers removed the use of relative tolerance in the commit. Before the commit of loosening the threshold, many safe_margin values fall into the test failure range (i.e., the gray area in the figure). The assertions are flaky and may fail in some runs. Hence, developers manually loosened the threshold to avoid the test failures. After this commit (i.e., the green lines), the assertions become less sensitive to variance and thus less flaky.

We find that OA assertions are often modified as code evolves. Developers modify OA assertions for various reasons, such as to increase test coverage, to accommodate different hardware specifications, to support different data representations, or to avoid test failures. Our study indicates challenges on maintaining OA assertions as there is no systematic support to help developers manage OA assertions, e.g., detecting improper threshold values, and suggesting changes on OA assertions when code evolves.

V. Threats to Validity

Internal Validity. Studying oracle approximation practice requires a list of *OA*-assertion APIs that developers use in each studied library. In this work, we use an iterative approach that combines reading documentation and code search. It is possible that we might miss some *OA*-assertion APIs that are rarely used in the codebases as we do not use an automated method. However, we believe that it should minimal impacts on the findings due to the rarity.

In RQ2, the manual categorization on oracle types is partially subjective in some of the computation oracles, i.e., relevant internal implementation and native implementation. To reduce the subjective bias in this process, we have two people concluding the labels independently and discuss to reach an agreement. In addition, when detecting code changes on *OA* assertions, we rely on the fact that the modifications (i.e. +/- lines) are performed directly on the statments of *OA*-assertions APIs. Hence, we are not able to detect changes if the changes are performed on the variables and the variables are used in *OA* assertions.

External Validity. First, we focus on studying DL libraries in Python or the Python part in a multi-language library in this work. Our findings may not be generalizable to DL libraries of other programming languages. In the future, we plan to extend the study to include other languages, such as C++ and Java. Second, our findings are based on the four studied DL libraries. In the future, we plan to include more DL implementations, including the ones in different languages. Third, when analyzing the reasons behind the code changes on *OA* assertions, because such analysis would require much manual effort and understanding on the relevant code base, we limit the study to one test module in TensorFlow. In the future, we plan to expand the scope of this RQ significantly to reveal different maintenance challenges developers might have when testing different components in DL libraries.

VI. Related Work

Studies on Testing Scientific Software. There exist some efforts in literature that try to bridge the gap between software engineering and scientific software. Particularly, researchers from both communities start to realize the importance of systematic testing on the developed scientific programs. Software defects cause inaccurate results instead of models and algorithms and may lead to publication retraction [12]. Hannay et al. [13] studied the development process of scientific software and revealed that despite a consensus on the importance of testing, only a small number of scientists have sufficient knowledge on testing. Carver et al. [14] perform case studies on the development of five scientific software and find that developers often find it challenging to perform validation and verification due to difficulty of writing good test cases. However, it remains unknown how developers actually conduct testing in scientific software, i.e., detailed analysis on test cases. Hence, our work takes an important step to understand the testing practice in open-source DL libraries, which have many similarities with traditional scientific software. Moreover, although testing scientific software often requires oracle approximation [15], there is no prior work that focuses on oracle approximation practices in scientific software.

Researchers have proposed various testing techniques to alleviate the oracle problem in testing scientific software, such as metamorphic testing [16], [17] and property testing [18]. Interesting, our study highlights that when using non-oracle testing to test DL libraries, such as metamorphic testing and differential testing, oracle approximations are commonly used. Our findings indicate that future adaption of automated non-oracle testing techniques in DL libraries should consider the prevalence of oracle approximations to avoid flaky test or over-restricted oracle comparison.

Studies on Software Testing. Researchers perform empirical studies on practices of test cases to better understand the challenges of software testing. Vahabzadeh et al. [19] study

the prevalence and reasons of bugs in test code. Barr et al. [10] survey oracle problems in the literature. Zaidman et al. [20] study the co-evolution of source code classes and test cases. Pinto et al. [7] work on understanding common myths on test-suite evolution. Their study find that although most test case changes are about refactoring, deletion and addition of test cases, there are some changes that complex and are hard to automate. Beller et al. [21] work on understanding why developers do not perform test in IDEs. Luo et al. [6] conduct the study to understand why flaky tests occur. Beller et al. [22] empirically study why CI tests fail by analyzing Travis CI and GitHub. Prior studies on software testing focus on general software and do not consider the peculiarity of DL libraries. In this work, we present the first important step to understand how developers perform oracle approximations in DL libraries, which is not studied by prior work.

Testing Deep Learning Models and Deep Learning libraries. In recent years, there have been much research effort paid to improve testing DL models. New coverage metrics targeting at DL models are proposed. Automated testing techniques are proposed to improve the testing of DL models. Pei et al. [23] present the first white-box testing of DL models and utilize a new coverage metric to guide the white-box fuzzing. Tian et al. [24] propose to utilize fuzz testing (i.e., fuzzing images) to generate more test data for autonomous driving cars. More recently, more advanced coverage metrics [25], [26] are proposed to improve testing DL models. Moreover, Ma et al. [27] propose to use mutation testing to evaluate the effectiveness of test cases on DL models. Differently, our work focuses on studying testing practice in DL libraries, and the quality will affect the accuracy of the generated models.

Some recent efforts are spent to improve the quality of DL libraries and applications. Zhang et al. [28] performed an empirical study on 11 open-source DL applications on GitHub and categorized totally 179 bugs. Hung et al. [5] propose a novel technique to detect defects in DL libraries based on inconsistencies among different DL libraries. Their work leverages the differences in the generated DL models, which are high-level outputs, while our work examines tests at all levels from the perspective of oracle approximation practice. It remains as future work to examine how low-level oracle approximations, if not done properly, may affect the accuracy of high-level models.

VII. Conclusions

In this paper, we present an empirical study on oracle approximations in testing DL libraries. Our study is an important first step to understand the current practices of using oracle approximations in compute-intensive software, such as DL libraries. Our work answers four research questions. First, we study the prevalence of oracle approximations in the test cases of DL libraries. We find that up to 25% of all the assertions use oracle approximations. Second, we study and conclude the diversity of test oracles and thresholds used in oracle approximations. In many cases, oracles used in oracle approximations are obtained through computation. Third, we study the common code changes that developers perform on oracle approximation assertions. Last, we conclude the reasons behind the code changes on oracle approximations. Our findings reveal maintenances challenges developers may be faced with in oracle approximations and may inspire future research to provide better tooling support for developers to better manage oracle approximation practices.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[3] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[4] F. Chollet *et al.*, "Keras," 2015.

[5] H. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *Proceedings of the 40th International Conference on Software Engineering*, 2019.

[6] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), pp. 643–653, ACM, 2014.

[7] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, (New York, NY, USA), pp. 33:1–33:11, ACM, 2012.

[8] K. Atkinson and W. Han, "Error and computer arithmetic," in *Elementary Numerical Analysis*, ch. 2, pp. 33–71, WILEY, 3 ed., 2004.

[9] "Tensorflow architecture," 2018.

[10] 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, May 2015.

[11] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 85–99, ACM, 2016.

[12] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, pp. 1219 – 1232, 2014.

[13] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[14] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 550–559, IEEE Computer Society, 2007.

[15] D. Hook and D. Kelly, "Testing for trustworthiness in scientific software," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, (Washington, DC, USA), pp. 59–64, IEEE Computer Society, 2009.

[16] T. Y. Chen, Jianqiang Feng, and T. H. Tse, "Metamorphic testing of programs on partial differential equations: a case study," in *Proceedings 26th Annual International Computer Software and Applications*, pp. 327–333, Aug 2002.

[17] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," no. 1, 2009.

[18] U. Kanewala and J. M. Bieman, "Techniques for testing scientific programs without an oracle," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, SE-CSE '13, (Piscataway, NJ, USA), pp. 48–57, IEEE Press, 2013.

[19] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 101–110, Sep. 2015.

[20] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v. Deursen, "Mining software repositories to study co-evolution of production and test code," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 220–229, April 2008.

[21] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 179–190, ACM, 2015.

[22] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 356–367, May 2017.

[23] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 1–18, ACM, 2017.

[24] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 303–314, ACM, 2018.

[25] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, (New York, NY, USA), pp. 120–131, ACM, 2018.

[26] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), pp. 477–487, ACM, 2019.

[27] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," *CoRR*, vol. abs/1805.05206, 2018.

[28] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), pp. 129–140, ACM, 2018.