The Unit Test Quality of Deep Learning Libraries: A Mutation Analysis

Li Jia, Hao Zhong and Linpeng Huang
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
insanelung@sjtu.edu.cn, zhonghao@sjtu.edu.cn, huang-lp@cs.sjtu.edu.cn

***Abstract*—In recent years, with the flourish of deep learning techniques, deep learning libraries have been used by many smart applications. As smart applications are used in critical scenarios, their bugs become a concern, and bugs in deep learning libraries have far-reaching impacts on their built-on applications. Although programmers write many test cases for deep learning libraries, to the best of our knowledge, no prior study has ever explored to what degree such test cases are sufficient. As a result, some fundamental questions about these test cases are still open. For example, to what degree can existing test cases detect bugs in deep libraries? How to improve such test cases? To help programmers improve their test cases and to shed light on the detection techniques of deep learning bugs, there is a strong need for a study on the test quality of deep learning libraries.**

To meet the strong need, in this paper, we conduct the first empirical study on this issue. Our basic idea is to inject bugs into deep learning libraries, and to check to what degree existing test cases can detect our injected bugs. With a mutation tool, we constructed 1,545 buggy versions (*i.e.*, mutants). By comparing the testing results between clean and buggy versions, our study leads to 11 findings, and we summarize them into the answers to three research questions. For example, we find that although existing test cases detected 60% of our injected bugs, only 30% of such bugs were detected by the assertions of these test cases. As another example, we find that some exceptions were thrown only in specific learning phases. Furthermore, we interpret our results from the perspectives of researchers, library developers, and application programmers.

I. Introduction

In recent years, deep learning techniques have already been applied in many fields [14], [20], [30], [43]. To reduce the development effort, various libraries (*e.g.*, TensorFlow [11]) are widely used [19], [47]. As a result, bugs inside libraries can have far-reaching impacts on many deep learning applications. Like many other machine learning systems, deep learning software inherently has variance in its results [40]. To check outputs with variance, Nejadgholi and Yang [34] report that 5% to 24% test cases in deep learning libraries check outputs against specific ranges. As it is challenging to determine the proper range of a test case, they [34] found that programmers frequently modify assertions: using a different assertion (23%), tightening or loosening thresholds (18%), and switching code under test/test oracles (58%).

The variance of deep learning systems and their frequently changed test units leave some open and fundamental questions. For example, How are deep learning libraries tested? Which test cases are more effective? To shed lights on the related research, there is a strong and timely need for an empirical study on this issue.

In this paper, we conduct the first empirical study to analyze the unit test quality of deep learning libraries. Our study is built on mutation testing, an intensively studied technique to assess the quality of a test suite [25]. Given a program with a test suite, mutation testing mutates the program with predefined mutation operators, and generates many mutants, *i.e.*, buggy versions. If a test case detects a mutant, this mutant is considered as killed. In mutation testing, a test suite is considered to be better, if it kills more mutants than others. In this study, we selected three popular deep libraries including TensorFlow [11], Keras [3], and Theano [15], since the three libraries are among the top ten of the most popular deep learning libraries [19], [47]. With a mutation testing tool [4], we constructed 1,545 mutants; recorded their execution results with test cases; and compared them with those of clean versions. Based on our results, we summarize 11 findings, and present our answers to the following three research questions.

- **RQ1. To what degree can mutants be killed, if they are injected to the system under test (SUT)?**
  Motivation. The answers are useful to estimate the unit test quality of deep learning libraries. Besides the overall results and symptoms, we classify mutants by their types to explore the impact of bug types.
  Answer. Table III shows that around 60% mutants in SUTs are killed. Existing test cases are more effective to detect bugs caused by wrong logic flows than those caused by wrong values (Finding 1), but they fail to detect mutants on decorator sentences (Finding 2). Among the killed mutants, more than 60% mutants were not detected by assertions in test cases, but were revealed by exceptions (Finding 3).
- **RQ2. To what degree can mutants be killed, if they are injected to dependent-on components (DOC)?**
  Motivation. For a test case, its DOC includes the files that are not directly tested but called by its SUT. Section II-B presents an example of SUT and DOC. As unit test cases are limited, many source files of deep learning libraries are DOCs. The answers are useful to estimate how such files are tested.
  Answer. Finding 4 shows that survived mutants increases from around 30% of SUTs to more than 50% of DOCs.

TABLE I: Our subject deep libraries

Library	Ver.	LoC	File	Method	Assertion	Unit
Theano	1.05	49,844	147	2,170	5,488	unittest
TensorFlow	1.15	309,942	986	4,611	37,648	unittest
Keras	2.31	12,170	45	748	1,608	pytest

- **RQ3. Which test cases are effective to kill mutants?** **Motivation.** The answers are useful to improve unit test cases. To achieve this research goal, we classify test cases by their assertions, thrown exceptions, and their learning phases. After that, we analyze which types of test cases are more effective to kill mutants.
  Answer. Finding 5 shows that Theano may not fully use the assertions of unit frameworks. Assertions in test cases are more effective to kill mutants than those in other files (Finding 6). When mutants are revealed by thrown exceptions, most of them throw `TypeError`, `ValueError`, and `IndexError` (Finding 7). Compared to the exceptions defined by Python, customized exceptions are rarely used, and mutants are seldom revealed by customized exceptions (Finding 8). For Theano and TensorFlow, mutants in the preprocessing phase are more challenging to be killed than the other phases (Finding 10), and several customized exceptions are associated with specific deep learning phases (Finding 11).

II. Preliminaries

In this section, we introduce mutation testing (Section II-A) and the test cases of deep learning libraries (Section II-B).

A. Mutation Testing

Mutation testing [25] is an intensively studied technique to assess the quality of test cases. To achieve this goal, mutation testing defines mutation operators (see Table II for examples), and applies operators to produce faulty versions of a software project. In mutation testing, each produced faulty version is called a mutant. After mutants are generated, test cases are executed on all mutants. If a test case fails on a mutant, the mutant is marked as "killed", which means that the test case has the capability of revealing the mutant. If no test case fails on a mutant, the mutant is marked as "survived", which means that no test case has the capability of revealing the mutant. Given two test suites, if a test suite kills more mutants, the test suite shall have better quality than the other one. Facebook reported that a mutation testing tool successfully improved its test suites [13].

Motivated by mutation testing, some recent approaches [21], [33] mutate trained deep learning models to produce buggy models. These approaches are unable to support our research purpose, because no unit test in deep learning libraries is designed to test trained models. As our subjected deep libraries in Section III-A all provide Python APIs, we selected a mutation tool for Python called mutmut [4] to inject bugs. The injected faults are different from real faults, and we further discuss this issue in Section IV-D.

B. Deep Learning Libraries and their Unit Test Cases

Their unit test cases are built upon Python test frameworks (*e.g.*, `pytest` [6]). In `pytest`, the name of a test method must start with "test_" or end with "_test". For example, in Keras, a test file, `losses_test.py` is implemented to test the loss funtions, and a test function is as follows:

```
1 def test_unweighted(self):
2   mse_obj = losses.MeanSquaredError()
3   y_true = K.constant([1, 9, 2, -5, -2, 6],shape=(2, 3))
4   y_pred = K.constant([4, 8, 12, 8, 1, 3],shape=(2, 3))
5   loss = mse_obj(y_true, y_pred)
6   assert np.isclose(K.eval(loss), 49.5, atol=1e-3)
```

In this function, `y_true` and `y_pred` (defined in Line 3 and 4) are prepared as the input of the loss function, `mse_obj`. As this loss function is implemented in the `losses.py` file, we consider this file as the SUT of this test method. Beside this SUT, the `mse_obj` method calls the functions of `tensorflow_backend.py` and `generic_utils.py`, and we consider the two files as the DOCs of `test_unweighted`.

The links between a test file and its SUT can be identified by the matching their file names. For example, under many directories of TensorFlow source code, we can find files named by `x_test`, where `x` denotes the SUT of a test file. For example, the SUT of the file `session_test.py` is `session.py`. Like other test frameworks, `pytest` has `assert` to check test outputs. In Line 6 of this example, the assertion compares the output value of `K.eval(loss)` with an expected value 49.5. If the difference is within an acceptable range (`1e-3`), the test method is passing.

We record traces with a coverage tool, and a file is considered as touched, if it appear in the traces (see Section III-B for details). Both SUT and DOC must appear in traces, and we injected bugs only to the executed lines of SUT and DOC.

Unit test cases are designed to test individual modules and functionalities, and integration test cases are designed to test software as a whole system [18]. However, when we conducted our study, the three libraries provide very few integration test cases with assertions. Without assertions, test cases serve as confidence testing [16], but it is difficult to determine whether a mutant can be killed. As a result, we ignore such test code. We notice that programmers start to release the problem. For example, Keras have gradually added integration test cases to its repository [10].

TensorFlow is implemented in multiple languages, its kernel is implemented by C++. Jia *et al.* [24] report that modifications in C++ files can affect in test cases in Python. If another tool injects mutants to a source file in C++ files, it is more difficult to detect this bug, since all unit test cases are written in Python and are typically not designed to detect cross-language bugs. As we inject only Python code, we can overestimate the quality of test cases. Meanwhile, mutants can be semantically equivalent to a clean program [29]. As it is infeasible to kill equivalent mutants, we can underestimate the quality of test cases. As equivalent mutants shall be rare, we can still overestimate the test quality. Additionally, TensorFlow provides testcases in different languages, but we only consider

TABLE II: Mutation operators

Operator	Description
ArOR	$op1 \leftrightarrow op2 \in \{+ \leftrightarrow -, * \leftrightarrow /, \% \leftrightarrow /, // \leftrightarrow /\}$
BitOR	$op1 \leftrightarrow op2 \in \{\& \leftrightarrow \vert, \wedge \leftrightarrow \&, << \leftrightarrow >>\}$
ComOR	$op1 \leftrightarrow op2 \in \{> \leftrightarrow >=, < \leftrightarrow <=, == \leftrightarrow !=, is \leftrightarrow is\ not\}$
LogOR	$op1 \leftrightarrow op2 \in \{and \leftrightarrow or, not \leftrightarrow\}$
AsOR	$op1 \leftrightarrow op2 \in \{+= \leftrightarrow -=, += \leftrightarrow =, -= \leftrightarrow =, *= \leftrightarrow /=, *= \leftrightarrow =, /= \leftrightarrow =\}$
MemOR	$op1 \leftrightarrow op2 \in \{in \leftrightarrow not\ in\}$
BVR	$b1 \leftrightarrow b2 \in \{True \leftrightarrow False\}$
NVR	$original\ value \leftrightarrow original\ value + 1$
SVR	$original\ string \leftrightarrow XX + orinial\ string + XX$
KVR	$break \leftrightarrow continue;\ copy \leftrightarrow deepcopy$
DecSR	$remove\ decorator\ sentence$
LmER	$remove\ lambda\ expression$
AsVR	$original\ assignment \leftrightarrow none$

ArOR: arithmetic operator replacement; BitOR: bitwise operator replacement; ComOR: comparison operator replacement; LogOR: logical operator replacement; AsOR: assignment operator replacement; MemOR: member operator replacement; BVR: boolean value replacement; NVR: numeric value replacement; SVR: string value replacement; KVR: keyword value replacement; DecSR: decorator sentence removal; LmER: lambda expression removal; AsVR: assignment value replacement;

test cases written in Python. As a result, the assessment of TensorFlow test case quality can be partial.

III. Methodology

In this section, we introduce our dataset (Section III-A) and our general protocol for RQs 1, 2, and 3 (Section III-B).

A. Dataset

To ensure the representativeness of our study, we select three popular deep learning libraries such as TensorFlow [11], Theano [15] and Keras [3]. We select these libraries, since they are among the top ten of the most popular deep learning libraries [19], [47]. Column "Ver." lists the versions of selected libraries. Column "LoC" lists the lines of test code. Column "File" shows the number of test files. Column "Method" shows the number of test methods. A test file typically has more than one test method, and a bug can affect one or more test methods. Column "Assertion" shows the number of assertions in their test files. Each test method can have one or more assertions, and an assertion checks whether a return result is as expected. Column "Unit" shows the unit test frameworks upon which they build the test cases. Here, `unittest` [7] is the official unit test framework of Python.

B. General Protocol

To determine whether a mutant is killed, our general protocol has the following three steps:

Step 1. Collecting execution results of original test cases. First, we executed the original test cases of each clean version for 20 times. We introduce this step for three purposes: (1) we recorded the numbers of passing, failing and skipped test methods, and use them as the baseline to determine killed mutants; (2) based on the executions, we located the SUTs of all test files; and (3) we used a coverage analysis tool [2] to collect executed files and code lines. Based on the results, we inject bugs to only executed lines, so that our injected bugs are all tested. To ensure the reliability of our results, we execute the test files for multiple times.

Step 2. Generating mutants of deep learning libraries. We classified the mutation operators of mutmut [4] into the 13 categories in Table II. For example, the first row lists an operator named arithmetic operator replacement. Here, the original arithmetic operator is denoted by $op1$, its target operator is denoted by $op2$, and the relation between the two operators is denoted by the mapping constructed by them. In such operators, buggy versions are generated by the replacement of original operators and their alternatives. A sample code snippet is shown as below:

```
imshp_logical = (imshp[0],) + imshp_logical[1:]
```

When the arithmetic operator replacement is applied to it, the original operation "+" will be replaced by the target operation "–", as a result, the mutated code becomes as follow, which may introduce a bug:

```
imshp_logical = (imshp[0],) - imshp_logical[1:]
```

This mutation tool uses a Python parser called parso [5] to build Abstract Syntax Trees (ASTs) from source files. For each covered line of a source file, we checked whether our mutation tool can mutate its AST nodes. If it does, we used the mutation tool to mutate this line, and generated a buggy version library. For files containing more than one mutable node, we generated multiple buggy versions. To make the relationship between bug symptom and cause of in a buggy version clearer, we modified only one node in each objective file. As several files are much larger than others, if we mutate all nodes, our analysis are limited to the several files, and our results can be biased. To reduce the bias, from each file, we mutated no more than 50 nodes.

Step 3. Collecting killed mutants. For each buggy version, we executed all the test files under the identical settings of Step 1. As there were many buggy versions, due to time limit, we executed each buggy version only once. For executed buggy versions, we recorded the number of passing and failing tests, and compared them with the baselines on clean versions. For failing tests, we further recorded their error messages. If an execution hanged, we manually stopped it and recorded its error message. We then classified the results by their outputs.

Step 4. Identifying test cases that kill mutants. If a test method failed on a mutant but it passed on the clean version, we consider that this mutant is killed. Here, a mutant can be killed for three reasons: (1) an assertion is violated; (2) a test method crashes; and (3) a test method hangs. We collected both the test cases that killed mutants and the test that did not kill mutants for our latter analysis.

IV. Empirical Result

This section presents the results of our study. More details are listed on our project website:
https://github.com/fordataupload/testcase

A. RQ1. Killed SUT Mutants

1) Protocol: We followed the protocol presented in Section III-B, but in this research question, we inject bugs to only

TABLE III: The killed SUT mutants

Operator	Theano			TensorFlow			Keras		
	killed	survived	total	killed	survived	total	killed	survived	total
ArOR	**21 (95.5%)** (%)	1 (4.5%)	22	**19 (95.0%)**	1 (5.0%)	20	**15 (60.0%)**	10 (40.0%)	25
BitOR	0	**1 (100%)**	1	**4 (57.1%)**	3 (42.9%)	7	0	0	0
ComOR	**66 (72.5%)**	25 (27.5%)	91	**46 (83.6%)**	9 (16.4%)	55	**69 (71.1%)**	28 (28.9%)	97
LogOR	**85 (73.3%)**	31 (26.7%)	116	**69 (87.3%)**	10 (12.7%)	79	**54 (88.5%)**	7 (11.5%)	61
AsOR	2 (50%)	2 (50%)	4	**5 (83.3%)**	1 (16.7%)	6	0	0	0
MemOR	**10 (90.9%)**	1 (9.1%)	11	**11 (91.7%)**	1 (8.3%)	12	**17 (89.5%)**	2 (10.5%)	19
BVR	18 (32.1%)	**38 (67.9%)**	56	9 (40.9%)	**13 (59.1%)**	22	6 (30.0%)	**14 (70.0%)**	20
NVR	**39 (62.9%)**	23 (37.1%)	62	**27 (77.1%)**	8 (22.9%)	35	42 (41.2%)	**60 (58.5%)**	102
SVR	26 (43.3%)	**34 (56.7%)**	60	24 (35.3%)	**44 (64.7%)**	68	**33 (62.3%)**	20 (37.7%)	53
KVR	0	0	0	1 (50%)	1 (50%)	2	0	**1 (100%)**	1
DecSR	0 (%)	0 (%)	0	5 (6.8%)	**68 (93.2%)**	73	1 (3.0%)	**32 (97.0%)**	33
LmER	**9 (100%)**	0 (%)	9	**1 (100%)**	0	1	0	0	0
AsVR	**64 (64.6%)**	35 (35.4%)	99	**125 (86.2%)**	20 (13.8%)	145	**117 (78.5%)**	32 (21.5%)	149
Total	**340 (64.0%)**	191 (36.0%)	531	**346 (65.9%)**	179 (34.1%)	525	**354 (63.2%)**	206 (36.8%)	560

```
1  def softmax(x, axis=-1):
2      ndim = K.ndim(x)
3  -   if ndim == 2:
4  +   if ndim != 2:
5          return K.softmax(x)
6      elif ndim > 2:
7          e = K.exp(x-K.max(x, axis=axis, keepdims=True))
8          s = K.sum(e, axis=axis, keepdims=True)
9          return e / s
10     else: ...
```

Fig. 1: A mutation on the `softmax` method

SUTs. In this way, we analyze to what degree can mutants be killed, if their buggy lines appear in SUTs.

2) *Result:* Table III shows the results. Column "Operator" lists the mutation operators that are used to generate mutants. Subcolumn "killed", "survived" and "total" list the numbers of killed, survived and total mutants, respectively. Due to the different of code structures, the numbers of the mutants vary across operators. For example, as we do not find bits in the SUTs of Keras, we cannot use BitOR to generate its mutants.

In Table III, for an operator, if its mutants of two libraries have more killed ones than survived ones, we highlight it in italic. ArOR, ComOR, LogOR, MemOR, AsVR, NVR, and LmER fall into this category. Except AsVR and NVR, all the other operators change the logic flows of clean versions. We notice that if logic flows are changed, a bug is likely to introduce visible impacts. For example, a mutant changed the `softmax` method as shown in Figure 1. In particular, ComOR changed "`==`" in Line 3 to "`!=`". A test case prepares a two-dimension tensor as its input. For this test case, the changed Line 3 leads to a wrong branch to handle the tensor, and produces a result that is quite different from the expect one.

In Table III, for an operator, if its mutants of two libraries have more survived ones than killed ones, we highlight it in bold. BVR, SVR, and DecSR belong to this category. Except DecSR, all the other operators change the values. We notice that these operators may introduce minor or local impacts. For example, a boolean variable is mutated as follows:

```
1  def bilinear_kernel_1D(ratio, normalize=True):...
```

BVR replaced the default value of `normalize` in Line 1 with `False`. In test methods, the default value is never used. For example, the following test method assign it to `True`:

```
1  kernel_ten_norm = bilinear_kernel_1D(ratio=rat,
       normalize=True)
```

As an exception, NVR often generates mutants that are easier to be killed. For example, another mutant is generated from the `softmax` in Figure 1: an NVR replaced the number `2` in Line 3 with `3`. As the dimension of test tensor is 3, the mutant went into the first condition branch instead of the second, and the mutant was killed. The above observations lead to our finding:

Finding 1. Existing unit test cases are more effective to detect bugs that are caused by wrong logic flows than those caused by wrong values.

Although the decorator is a widely used Python feature [37], as shown in the DecSR row of in Table III, most mutants (more than 90%) generated by removing decorator sentences survived. The main reason is that such sentences are used for wrapping functions without changing theirs contents. As most of the test methods manipulate original functions but do not use wrappers, removing decorator sentences does not affect the result of tests. A sample is shown as follow,

```
1  @contextlib.contextmanager
2  @tf_export("xla.experimental.jit_scope")
3  def experimental_jit_scope(compile_ops=True,
       separate_compiled_gradients=False):
4  ...
```

In TensorFlow, the `@tf_export` exposes a function or a class with a different name. In the above code, Line 2 exposes `experimental_jit_scope` with an alias, `xla.experimental.jit_scope`. As this alias is never tested, existing test cases fail to reveal any differences after removing this decorator, as a result, the mutant survives.

Besides aliases, DecSR can modify other annotations. In the above sample, Line 1 introduces `contextmanager` to manage resources. To achieve this goal, `contextmanager` uses a `with` statement to call the `__enter__` and `__exit__` method of a user defined class [1]. In a test method, a `with` statement is used as following (Line 3):

TABLE IV: The symptoms of killed SUT mutants

Operator	Theano				TensorFlow				Keras			
	assertion	crash	hang	total	assertion	crash	hang	total	assertion	crash	hang	total
ArOR	0 (%)	**21 (100%)**	0	21	**10 (52.6%)**	9 (47.4%)	0	19	**13 (86.7%)**	2 (13.3%)	0	15
BitOR	0	0	0	0	1 (25.0%)	**3 (75.0%)**	0	4	0	0	0	0
ComOR	**36 (54.5%)**	29 (43.9%)	1 (1.5%)	66	**28 (60.9%)**	17 (37.0%)	1 (2.1%)	46	20 (29.0%)	**49 (71.0%)**	0	69
LogOR	36 (42.4%)	**42 (49.4%)**	7 (8.2%)	85	25 (36.2%)	**43 (62.3%)**	1 (1.5%)	69	7 (13.0%)	**47 (87.0%)**	0	54
AsOR	0	1 (50.0%)	1 (50.0%)	2	2 (40.0%)	**3 (60.0%)**	0	5	0	0	0	0
MemOR	5 (50.0%)	5 (50.0%)	0	10	**7 (63.6%)**	3 (27.3%)	1 (9.1%)	11	4 (23.5%)	**12 (70.6%)**	1 (5.9%)	17
BVR	6 (33.3%)	**10 (55.6%)**	2 (11.1%)	18	**5 (55.6%)**	3 (33.3%)	1 (11.1%)	9	**3 (50.0%)**	2 (33.3%)	1 (16.7%)	6
NVR	12 (30.8%)	**27 (69.2%)**	0	39	13 (48.1%)	**14 (51.9%)**	0	27	21 (50.0%)	21 (50.0%)	0	42
SVR	3 (11.5%)	**23 (88.5%)**	0	26	7 (29.2%)	**17 (70.8%)**	0	24	8 (24.2%)	**25 (75.8%)**	0	33
KVR	0	0	0	0	**1 (100%)**	0	0	1	0	0	0	0
DecSR	0	0	0	0	0	**5 (100%)**	0	5	0	**1 (100%)**	0	1
LmER	1 (11.1%)	**8 (88.9)**	0	9	0	**1 (100%)**	0	1	0	0	0	0
AsVR	13 (20.3%)	**47 (73.4%)**	4 (6.3%)	64	12 (9.6%)	**109 (87.2%)**	4 (3.2%)	125	12 (10.3%)	**103 (88.0%)**	2 (1.7%)	117
Total	112 (32.9%)	**213 (62.7%)**	15 (4.4%)	340	111 (32.1%)	**227 (65.6%)**	8 (2.3%)	346	88 (24.9%)	**262 (74.0%)**	4 (1.1%)	354

TABLE V: SUT mutants killed by assertions

Unit	Statement	Library								
		Theano			TensorFlow			Keras		
		test case	other file	total	test case	other file	total number	test case	other file	total
Numpy	assert_allclose	0	0	0	16 (99)	0	16	0	0	0
unittest	assertEqual	11 (10)	0	11	56 (981)	0	56	0	0	0
	assertNotEqual	0	0	0	0 (7)	0	0	0	0	0
	assertListEqual	0	0	0	6 (11)	0	6	0	0	0
	assertItemsEqual	0	0	0	0 (41)	0	0	0	0	0
	assertMultiLineEqual	0	0	0	1 (40)	0	1	0	0	0
	assertTrue	4 (86)	0	4	2 (63)	0	2	0	0	0
	assertFalse	0 (16)	0	0	1 (38)	0	1	0	0	0
	assertIn	0	0	0	0 (13)	0	0	0	0	0
	assertIs	0	0	0	0 (86)	0	0	0	0	0
	assertIsNot	0	0	0	0 (0)	0	0	0	0	0
	assertIsNone	0	0	0	0 (0)	0	0	0	0	0
	assertRaises	0 (42)	0	0	0 (74)	0	0	0	0	0
	assertRaisesRegexp	0	0	0	20 (87)	0	20	0	0	0
	assertGreater	0	0	0	0 (7)	0	0	0	0	0
	assertLess	0	0	0	0 (3)	0	0	0	0	0
nose	assert_raise	1 (22)	0	1	0	0	0	0	0	0
	assert_true	0 (3)	0	0	0	0	0	0	0	0
Python	assert	56 (397)	19	75	0 (4)	9	9	76 (447)	12	88
	raise	4 (109)	17	21	0 (4)	0	0	0 (88)	0	0
Total		76	36	112	102	9	111	76	12	88

```
def compute(self, use_jit, compute_fn):
...
with jit.experimental_jit_scope(use_jit):
r = compute_fn()
```

After the decorator sentence is removed, the test fails, since the `__enter__` method is never called.

Finding 2. Existing unit test cases are ineffective to detect the bugs on decorator sentences, and in total more than 90% such mutants survived.

We further inspected the killed mutants by their symptoms, and Table IV shows the results. Column "Operator" lists the mutation operators. Subcolumn "assertion", "crashes" and "hang" list symptoms. In particular, the assertion denotes that a killed mutant triggers `assert` errors; and the crash denotes that a mutant is killed by crashes. Here, if a mutant throws an exception by `raise` statement, we count it in crash. If test cases hang on a mutant, as the program does not respond, we do not know whether its assertions would be triggered. As a result, we count them in hangs.

Nejadgholi and Yang [34] report that in the test cases of classical software, execution results are often compared with constants, but due to the random nature of deep learning, in test cases of deep learning applications, execution results are often compared with ranges. Table V shows mutants killed by assertions. Column "Statement" lists the assertion statements, and their names indicate a return value is checked. For instance, `assertIn`, `assertGreater`, and `assertLess` are designed to check ranges, and programmers can check ranges in other assert statements (details are introduced in Section IV-C). In sum, Table IV shows that assertions killed fewer mutants than crashes. Indeed, the distribution leads to our finding:

Finding 3. Instead of carefully designed assertions in unit test cases, most mutants (62.7%, 66.1%, and 81.9%) are killed by crashes.

Typically, when programmers write test cases, they rely on assertions to detect bugs. However, we find that most mutants were killed by crashes other than assertions. Even if a mutant is killed by a crash, programmers can still take much effort to locate its faulty locations.

B. RQ2. Killed DOC Mutants

1) Protocol: In this research question, we explore survived mutants in DOCs. To achieve this goal, for each test case,

TABLE VI: The killed DOC mutants

Operator	Theano			TensorFlow			Keras		
	killed	survived	total	killed	survived	total	killed	survived	total
ArOR	**6 (100%)**	0	6	**2 (100%)**	0	2	4 (30.8%)	**9 (69.2%)**	13
BitOR	0	0	0	0	0	0	0	0	0
ComOR	**24 (51.1%)**	23 (48.9%)	47	**17 (73.9%)**	6 (26.1%)	23	**18 (60.0%)**	12 (40.0%)	30
LogOR	**30 (55.6%)** (%)	24 (44.4%)	54	**40 (75.5%)**	13 (24.5%)	53	**31 (60.8%)**	20 (39.2%)	51
AsOR	**1 (100%)**	0	1	**2 (100%)**	0	2	2 (33.3%)	**4 (66.7%)**	6
MemOR	**4 (66.7%)**	2 (33.3%)	6	**4 (100%)**	0	4	**3 (75.0%)**	1 (25.0%)	4
BVR	7 (13.2%)	**46 (86.8%)**	53	2 (9.5%)	**19 (90.5%)**	21	5 (41.7%)	**7 (58.3%)**	12
NVR	7 (18.4%)	**31 (81.6%)**	38	1 (4.3%)	**22 (95.7%)**	23	8 (12.1%)	**58 (87.9%)**	66
SVR	5 (14.7%)	**29 (85.3%)**	34	6 (15.4%)	**33 (84.6%)**	39	8 (32.0%)	**17 (68.0%)**	25
KVR	0	0	0	0	0	0	0	0	0
DecSR	4 (19.0%)	**17 (81.0%)**	21	0	**32 (100%)**	32	2 (8.3%)	**22 (91.7%)**	24
LmER	0	0	0	0	0	0	0	0	0
AsVR	**31 (47.7%)**	34 (52.3%)	65	**36 (64.3%)**	20 (35.7%)	56	**55 (63.2%)**	32 (36.8%)	87
Total	119 (36.6%)	**206 (63.4%)**	325	110 (43.1%)	**145 (56.9%)**	255	136 (42.8%)	**182 (57.2%)**	318

we checked its coverage obtained in III-B to determine its DOC. After that, we injected bugs to DOCs, and followed the protocol in Section III-B to analyze survived mutants.

2) Result: Table VI shows the results. Its columns have the same meanings as Table III. Finding 1 shows that wrong logic flows are easier to be killed than wrong values, but NVR is an exception. As a comparison, on DOCs, Table VI shows that even NVR produces more survived mutants than killed ones. Besides wrong values, even wrong logic flows are more difficult to be killed on DOCs. For example, in `test_training` of Keras, the `losses` component is imported to implement loss function. A code snippet is shown as following:

```
1 def mean_squared_error(y_true, y_pred):
2   if not K.is_tensor(y_pred):
3                   y_pred = K.constant(y_pred)
4                   y_true = K.cast(y_true, y_pred.dtype)
5   return K.mean(K.square(y_pred - y_true), axis=-1)
```

In the above code, a mutant replaced the "–" with "+" in Line 5, and it modified the loss function `mean_squared_error`. This mutant is survived, because test methods check only the type and the shape of an output tensor but do not check their values. As test cases focus more on SUTs, by comparing Tables III and VI, we come to another finding:

Finding 4. The mutants on DOCs are more challenging to be killed than the mutants on SUTs, and survived mutants in DOCs increase from around 30% to more than 50%.

C. RQ3. Effective Test Case

1) Protocol: In this research question, we explore the effectiveness of test cases. To achieve this research goal, we classify test cases by different criteria.

1. Classifying test cases by their assertion statements. In a test case, typically, programmers write assertions to check whether return values are as expected. As shown in Column "Unit" of Table V, assertion statements are implemented by Numpy [9], unittest, nose [8], and Python. Column "Statement" lists the assertion statements. In particular, `assert_allclose` checks whether the differences between two arrays are fewer than a predefined threshold; `assertListEqual` checks whether two lists are equal; `assertMultiLineEqual` checks whether two string values are equal when ignoring their line breaks; `assertRaises` and `assert_raise` raise an exception when a condition is satisfied; and `assertRaisesRegex` raises an exception if a string value matches a predefined regular expression.

Besides the above assertion statements, programmers can define their own checks. For example, TensorFlow implements an assertion statement, `assertTransformedResult`, to check whether a transformed result is as expected:

```
def assertTransformedResult(self, test_fn, inputs,
    expected, symbols=None):
...
with self.converted(test_fn, control_flow, symbols,
(constant_op.constant,)) as result:
            self.assertAllEqual(self.evaluate(result.test_fn
                (*inputs)), expected)
```

As shown in the above code, the statement is built upon the `assertAllEqual` statement of unittest, we classified this assertion into the category of `assertAllEqual`. To understand the test capability of both sources, we classify them separately.

2. Classifying test cases by their exception types. As shown in Tables III and VI, more than half of mutants are killed, because they crash test cases. When they crash, they typically throw exceptions. For these test cases, we classify them by their thrown exceptions.

3. Classifying test cases by their learning phases. Each test case has its SUT, and a SUT can implement functionality of different deep learning phases such as preprocessing, constructing, and learning [42]. In particular, the preprocessing phase transforms raw data (*e.g.*, texts, images, and videos) into the input formats of deep learning libraries; the constructing phase builds the structure of a deep model (*e.g.*, CNN and RNN); and the learning phase trains and tests deep learning models. We classify test cases according to their phases. If a SUT is used in more than one phase, we counted its test cases in each phase. For example, we find that a SUT stores the neural network parameters and it is used in both constructing and learning phases, we count its test cases twice in both the constructing and learning phases.

2) Results: Based on our three criteria, we classify our test cases, and our results are as follows:

a) ***Test cases classified by their assertion statements:*** Table V shows the results. Subcolumn "test case" denotes

TABLE VII: The exceptions thrown by killed SUT mutants

Source	Exception	Description	Library		
			Theano	TensorFlow	Keras
Python	AtrributeError	An attribute reference or assignment fails	620	561	176
	TypeError	The type of an object is not as expected	5,206	1,207	516
	ValueError	A value is incorrect	3,130	615	196
	KeyError	A mapping key is not found	35	23	12
	UnboundlocalError	No value has been bound to a local variable in a method	0	34	90
	IndexError	A sequence subscript is out of range	1,877	41	14
	FileExistsError	Trying to create a file or directory which already exists	0	0	1
	FileNotFoundError	A file or directory does not exist	0	0	1
	NameError	A local or global name is not found	0	23	20
	NotImplementError	A functionality is unimplemented	29	0	0
	RuntimeError	Undefined errors	0	9	45
Numpy	AxisError	The value of parameter `axis` is out of range	60	0	2
Theano	UnusedInputError	A symbolic input passed to function is not needed	67	0	0
	MissingInputError	A symbolic input needed to compute the output is missing	19	0	0
	GradientError	A gradient is incorrectly calculated	15	0	0
	CachedConstantError	A graph that has a cached constant is passed to FunctionGraph	2	0	0
	InconsistencyError	A graph that has invalid state is passed to FunctionGraph	1	0	0
TensorFlow	OperatorNotAllowedInGraphError	An operator is not allowed in graph execution	0	64	0
	InvalidArgumentError	An operation receives an invalid argument	0	40	2
	StagingError	During staging of converted code	0	21	0
	InaccessibleTensorError	A tensor is not accessible	0	1	0
Total			11,061	2,639	1,073

mutants that are killed by assertions in test cases. The number in brackets denotes the total numbers of assertions in test cases, and the number outside brackets denotes the numbers of assertions that killed mutants. As an assertion can kill more than one mutant, the outside numbers can be larger than the inside numbers. Besides test cases, mutants can be killed by assertions in other files (*e.g.*, deep learning libraries). In Table V, subcolumn "other file" denotes mutants that are killed by such assertions. The assertion statements are implemented by Numpy [9], unittest, nose [8], and Python. The three libraries use assertion statements in different ways.

1. Theano mainly use two types of assertion statements such as asserts (`assertTrue`, `assertFalse`, `assert_true` and `assert`) and raises (`assertRaise`, `assert_raise` and `raise`). In Python, `assert` checks whether a condition is true, and `raise` throws an exception. Although Theano uses other unit testing libraries, its programmers use only the two types of checks. Indeed, we tried to replace `assert_raise` with `raise`, and the testing results are not changed.

2. TensorFlow use all types of assertion statements. The programmers of TensorFlow seldom use `assert` and `raise` of Python in test cases. Their test cases are built on `unittest`, and they use all types of assertion statements of this unit testing library. To make it easier to compare two arrays, they even use `assert_allclose` of Numpy.

3. Keras uses only the `assert` and `raise` statements of Python. The test cases of Keras are built upon `pytest`, and this unit framework does not implement customized assertions. As a result, the test cases of Keras use only the `assert` and `raise` statements of Python.

Kawrykow and Robillard [28] find that programmers may not effectively call API libraries. Although unit frameworks like `unittest` implement various assertion statements, we find that programmers may not effectively use them either. Its impacts on killed mutants are minor, since Table III shows that the test cases of all the libraries killed around 60% mutants. However, Table IV shows that the assertions of Keras killed fewer mutants than others (around 32% vs 24.9%), while Table V shows that Keras uses fewer types of assertion statements. The observations lead to a finding:

Finding 5. Leveraging assertion functions provided by specific testing frameworks can improve the effectiveness of unit test cases.

Besides test cases, programmers can write assertions in their code, but only few mutants were killed by such assertions. As shown in Table V, the assertions in test code killed 67.9%, 91.9%, and 86.4% mutants of Theano, Tensorflow, and Keras. This observation leads to another finding:

Finding 6. The assertions in test cases are more effective to kill mutants than the assertions in library code.

*b) **Test cases classified by their exception types:*** As shown in Table VII, Column "Source" shows sources where the exceptions are implemented. Column "Exception" shows the types of exceptions. Column "Description" shows the scenarios when these exceptions are thrown. From the results in Table VII, we have the following observations:

1. Most killed mutants throw `TypeError`, `ValueError`, and `AtrributeError`. `TypeError` indicates that the type of value is not as expected. Previous studies [24], [49] report that the type confusion is a common reason of bugs in deep learning software. We notice that library code has many checks for such errors. For example, in a test method `testRepr` of TensorFlow:

```
def testRepr(self):...
op = ops.Operation(ops._NodeDef("None", "op1"), ops.
    Graph(), [], [dtypes.float32])
```

In the method `_update_input` of `Operation`, the input to this operation is updated, and the type of input `tensor` should be checked at first, if it is not a `Tensor`, a `TypeError` is raised.

TABLE VIII: The SUT mutants killed by assertions of different phases

Phase	Library								
	Theano			TensorFlow			Keras		
	test case	other file	total	test case	other file	total	test case	other file	total
Preprocessing	3 (188)	3	6	5 (689)	0	5	23 (126)	0	23
Constructing	42 (605)	14	56	53 (1,024)	8	61	24 (214)	3	27
Learning	50 (434)	21	71	44 (1,349)	1	45	27 (334)	26	53

```
def _update_input(self, index, tensor):
if not isinstance(tensor, Tensor):
raise TypeError("tensor␣must␣be␣a␣Tensor:␣%s" % tensor)
```

`ValueError` indicates that the value of a variable is incorret. The prior studies [24], [49] report that deep learning software has dimension mismatches, and such bugs can lead to `ValueError`. For example, the test method, `test_cce_one_hot`, of Keras calls the `sparse_categorical_crossentropy` method:

```
def test_cce_one_hot(self):
objective_output = losses.
      sparse_categorical_crossentropy(y_a, y_b)
```

As shown in the above code, this method calls the `sparse_categorical_crossentropy` method:

```
def sparse_categorical_crossentropy(y_true, y_pred,
      from_logits=False, axis=-1):
return K.sparse_categorical_crossentropy(
y_true, y_pred, from_logits=from_logits, axis=axis)
```

A mutant changed the default value of `axis` from `-1` to `0`, and the following backend method checked this value:

```
def sparse_softmax_cross_entropy_with_logits(...):...
 if (static_shapes_fully_defined and
labels_static_shape != logits.get_shape()[:-1]):
raise ValueError("Shape␣mismatch:...")
```

As a result, this mutant was killed.

`AttributeError` indicates that a class has no required attributes. As Python supports both class attributes and object attributes, even if two objects are initiated from the same class, they can have different attributes. Two objects with different attributes do not trigger `TypeError`, if their type is the same. However, when an object does not have an expected attribute, `AttributeError` will be thrown. For example, in a test method, `testNoVariables`, is as follows:

```
1 def testNoVariables(self):...
2 sgd_op = gradient_descent.GradientDescentOptimizer(3.0)
3 with self.assertRaisesRegexp(ValueError, '...'):
4 sgd_op.minimize(loss)
```

In Line 4 of the above code, the `minimize` method calls the `compute_gradients` method:

```
1 def compute_gradients(...):
2 if var_list is None:
3 var_list = tape.watched_variables()...
4 grads = tape.gradient(loss_value, var_list, grad_loss)
```

In Line 2 of the above code, a mutant replaced `is` is replaced with `is not`. As a result, the attributes of `var_list` are not correctly constructed. Line 4 of the above code calls the `tape.gradient` method:

```
1  def gradient(self,target...):
2   for t in nest.flatten(target):
3         if not t.dtype.is_floating:
```

Line 3 of the above code checks the type of the attribute. As the attribute is not defined, it raises `AttributeError`.

2. Many killed mutants of Theano throw `IndexError`. `IndexError` indicates that the index of a list/array is invalid, *e.g.*, out of bounds. Theano programmers may not check the indexes of their lists and arrays sufficiently, so many killed mutants throw `IndexError`. For example, the `test_argtopk_1d` test method of Theano calls the `argtopk` method:

```
def test_argtopk_1d(self, size, k, dtype, sorted,
    idx_dtype):...
y = argtopk(x, k, sorted=sorted, idx_dtype=idx_dtype)
```

The `argtopk` method calls the `_topk_py_impl` method:

```
1 def _topk_py_impl(op, x, k, axis, idx_dtype):...
2 elif k > x.shape[axis]:
```

As shown in the above code, Line 2 compares an array without checking its index. A mutant produced a wrong value of `axis`, and it threw `IndexError`.

Meanwhile, we find that other libraries check the ranges of indexes more carefully. For example, the `_validate_keep_input` method of TensorFlow is as follows:

```
1 def _validate_keep_input(keep_input, enqueue_many):
2   keep_input = ops.convert_to_tensor(keep_input)
3   if keep_input.shape.ndims is None:
4     raise ValueError(
5         "keep_input␣dimensions␣must␣be␣known␣at␣graph␣
               construction.")
6   ...
7   if keep_input.shape.ndims > 1:
8     raise ValueError("keep_input␣must␣be␣0␣or␣1␣
           dimensions.")
```

Line 7 of the above code checks the shape of `keep_input`. If an out-of-bound index is passed, the above code avoids `IndexError`, and throws `ValueError`.

Finding 7. In total, 46.9%, 26.7%, and 9.2% of killed mutants throw `TypeError`, `ValueError`, and `AtrributeError`, respectively. In Theano, `IndexError` is also common, and which are thrown from 17.0% of killed mutants.

Compared to the exceptions defined by Python, the exceptions defined by Theano and TensorFlow are rarely thrown. For example, `GradientError` is thrown only when calculating gradient descents. This observation leads to a finding:

Finding 8. Compared to the exceptions defined by the Python language, customized exceptions are not widely used, and killed mutants seldom throw such exceptions.

TABLE IX: The exceptions thrown in different phases

Source	Exception	Theano			TensorFlow			Keras		
		P	C	L	P	C	L	P	C	L
Python	AtrributeError	1	334	323	62	219	280	27	15	164
	TypeError	76	1,080	4,584	171	567	469	226	352	390
	ValueError	0	141	2,989	66	121	428	0	83	113
	KeyError	0	32	32	0	0	23	11	12	11
	UnboundlocalError	0	0	0	0	34	0	0	90	0
	IndexError	0	68	1,863	0	11	30	5	5	14
	FileExistsError	0	0	0	0	0	0	1	0	0
	FileNotFoundError	0	0	0	0	0	0	1	0	0
	NameError	0	0	0	0	23	0	0	0	20
	NotImplementError	0	17	12	0	0	0	0	0	0
	RuntimeError	0	0	0	0	0	0	0	45	0
Numpy	AxisError	0	0	60	0	0	0	0	2	0
Theano	UnusedInputError	0	0	53	0	0	0	0	0	0
	MissingInputError	0	19	16	0	0	0	0	0	0
	GradientError	0	0	15	0	0	0	0	0	0
	CachedConstantError	0	2	0	0	0	0	0	0	0
	InconsistencyError	0	0	1	0	0	0	0	0	0
TensorFlow	OperatorNotAllowedInGraphError	0	0	0	0	28	0	0	0	0
	InvalidArgumentError	0	0	0	0	0	40	0	0	2
	StagingError	0	0	0	0	0	21	0	0	0
	InaccessibleTensorError	0	0	0	0	1	0	0	0	0
Total		77	1,693	9,948	299	1,004	1,291	271	604	714

*c) **Test cases classified by learning phases:*** Table VIII shows killed mutants that are classified by the assertions of different deep learning phases. Column "Phase" shows deep learning phases. Subcolumn "test case" denotes assertions of corresponding phases. The numbers in brackets denote the total assertions, and the numbers outside brackets denote the assertions that killed mutants. As some SUTs are shared by different phases, an assertion can be classified to more than one phase, and we count them in each phase. Subcolumn "other file" denotes assertions in other files that also killed mutants.

According to the results, for Theano and TensorFlow, although their preprocessing phases have hundreds of assertions, only several mutants were killed. Although the assertions are many, they often focus on few behaviors.

Finding 9. Although Theano and TensorFlow have hundreds of assertions in their preprocessing phases, these assertions killed only several mutants.

Table IX shows the distribution of exceptions in different deep learning phases. SubColumn "P", "C", and "L"denote the phases such as preprocessing, constructing, and learning, respectively. According to the results, more mutants throw `ValueError` in learning phase than the other phases, in that the learning phase has more calculations and checks. For example, the learning process of TensoFlow calls the `_hessian_vector_product` method:

```
1 def _hessian_vector_product(ys, xs, v):
2 length = len(xs)
3  if len(v) != length:
4   raise ValueError("... must have the same length.")
```

Line 3 checks the valid range of the index, and if the dimensions of input tensors are not matched, Line 4 raises `ValueError`. Table V shows that Theano checks the valid range of an index insufficiently. As a result, besides `ValueError`, in this phase, its killed mutants throw many `IndexErrors`.

Finding 10. In the learning phase, most mutants throw `ValueErrors`. Beside this exception, in this phase, the mutants of Theano throw many `IndexErrors`.

Additionally, we notice that customized exceptions are mainly used in the constructing and learning phases. For example, in Theano, the bugs of only the learning phase throw `GradientErrors`, and in TensorFlow, the bugs of only the constructing phase throw `OperatorNotAllowedInGraphErrors`.

Finding 11. Several customized exceptions are associated with specific learning phases (*e.g.*, `UnusedInputError` and `InvalidArgumentError`) and the constructing phase (*e.g.*, `OperatorNotAllowedInGraphError`).

Meanwhile, none of customized exceptions is thrown from the preprocessing phase.

D. Threats to Validity

The internal threats include the manual classification of deep learning phases, since we might misclassify some components. We carefully read source files and their test cases to reduce this threat. The internal threats also include the controversial correlation between the number of killed mutants and the ability of a test suite to detect real faults. For this issue, researchers provide both positive [27] and negative [35] evidences. Please note that although Papadakis *et al.* [35] criticize that the correlation is weaker than those reported in other studies, their results show that the correlation is still significant. The external threats to validity include the limited subjects. Although we selected 3 popular deep learning libraries, our selected subjects were limited. Analyzing more deep learning libraries (*e.g.*, PyTorch) and mutation tools can reduce this threat.

V. Interpretation

In this section, we interpret our findings.

1. The research opportunities on testing deep learning software. Existing test cases are less effective to detect bugs caused by wrong values (Finding 1) and bugs in specific statements (*e.g.*, decorators, Finding 2). It is worth exploring how to detect such bugs. Finding 3 shows that even if mutants are killed, around 60% of them were not killed by designed assertions of test cases, but were somewhat accidentally revealed by exceptions as introduced in Findings 9 and 10. Researchers can start from such exceptions to generate more effective test cases. Finding 2 shows that existing test cases are less effective to detect bugs in specific statements (*e.g.*, decorators). Meanwhile, as 100% coverage is unnecessary, Petrovic *et al.* [38] find that test cases do not have to kill all mutants. It is worth analyzing how many mutants in deep learning applications are unnecessary to be killed.

2. The inspirations for the programmers of deep learning libraries. Finding 1 shows that the existing test cases killed only around 60% of our injected bugs. In addition, Finding 3 shows that more than 60% bugs are revealed by crashes, instead of designed assertions in test cases. Finding 5 shows that using appropriate assertions provided by unit testing frameworks can improve the testing quality. When their libraries throw exceptions, Findings 10 and 11 show that some customized exceptions are associated with specific deep learning phases. If a bug throws such exceptions, programmers can inspect corresponding learning phases to locate faults. Findings 4 and 6 shows that in assertions in test case are more effective and bugs in SUT are easier to be detected than those of DOC. Findings 7, 9 and 10 have contents that are specific to libraries. For these findings, programmers can learn from other libraries. For example, according to Finding 7, programmers of TensorFlow and Keras can consider throwing more `IndexError` when the corresponding problems occur.

3. The inspirations for the programmers of deep learning applications. Although deep learning libraries are quite useful, programmers shall be aware of that such libraries are imperfect and their test cases detected only around 60% of our injected bugs as shown in Table III. Finding 1 shows that some specific patterns of bug are easier to be detected by test cases, but programmers shall be careful that their wrong values are less likely to be detected than wrong flow logics. Finding 11 suggests that the location of bugs can be inferred from specific assertion and exception messages. For example, `GradientError` may be raised in files related to gradient descent algorithm in learning phase. Such information is useful to locate related bugs.

VI. Related Work

Empirical study on deep learning development. Researchers analyzed bug characteristics in deep learning software. Zhang *et al.* [49] analyzed the bugs of TensorFlow applications. Islam *et al.* [22] analyzed bugs in the applications of Caffe [26], Keras, Theano, and Torch. Jia *et al.* [24] explored the bugs inside TensorFlow. Jahangirova and Tonella [23] presented an empirical study on the evaluation of mutation operators for deep learning systems. While the previous studies analyze the static characteristics of deep learning bugs, we analyze the runtime behavior of deep learning bugs. Besides its bugs, researchers conducted empirical studies to understand the other perspectives of the deep learning development. Zhang *et al.* [48] conducted a study on the challenges developers commonly face when building deep learning applications. Guo *et al.* [17] provided an overview of the frameworks and platforms influence on deep learning performance. Nejadgholi and Yang [34] conducted an empirical study on oracle approximation assertions in deep learning libraries. We analyze whether test cases of deep learning libraries are sufficient, which is not touched by the above studies.

Empirical studies on software testing. Researchers have conducted empirical studies to understand software testing. Vahabzadeh *et al.* [46] explored the characteristics of test bugs. Pinto *et al.* [41] investigated the evolution patterns of test suites. Barr *et al.* [12] surveyed test oracle problem. Luo *et al.* [31] studied flaky test in open-source projects and Thorve *et al.* [44] conducted study on flaky test in Android Apps. Nejadgholi and Yang [34] conducted an empirical study on oracle approximation assertions in deep learning libraries. In this study, from a different perspective, we analyze the test quality of deep learning libraries, and our results can be useful to improve the testing of deep learning software.

Testing deep learning applications. To improve the testing of deep learning model, researchers have introduced several methods and techniques. Pei *et al.* [36] proposed a white-box framework to test real-world deep learning systems. Ma *et al.* [32] proposed a set of multi-granularity criteria to measure the quality of test cases prepared for deep learning systems. Tian *et al.* [45] and Pham *et al.* [39] introduced differential testing to discover bugs in deep learning software. Our study shows that the existing test cases of deep learning libraries are insufficient to kill many mutants, which leaves adequate space for the improvement of the above approaches.

VII. Conclusion

To understand the test quality of deep learning libraries, with a mutation tool, we generated 1,514 buggy versions of three popular deep learning libraries. After that, we execute the unit test cases of the three libraries to analyze how many buggy versions can be detected. In this way, we explore the quality of their unit test cases. Based on the execution results, we summarized the testing statistics, and also explored the pattern of effective test cases. Our analysis lead to 11 findings. For example, we find that bugs caused by wrong values are more challenging to be detected. As another example, although existing test cases detected around 60% of our injected bugs, we find that only 30% of them were detected by the assertions of these test cases. Furthermore, we interpret these findings from the perspectives of researchers, library developers, and application programmers.

Acknowledgement

We appreciate reviewers for their insightful comments. This work is sponsored by the National Key R&D Program of China No. 2018YFC0830500. Hao Zhong is the corresponding author.

References

[1] Context managers. https://book.pythontips.com/en/latest/context_managers.html?highlight=context%20manager, 2019.

[2] Coverage. https://github.com/nedbat/coveragepy, 2019.

[3] Keras. https://keras.io, 2019.

[4] mutmut - python mutation tester. https://github.com/boxed/mutmut, 2019.

[5] parso - a python parser. https://parso.readthedocs.io/en/latest/, 2019.

[6] The pytest framework. https://docs.pytest.org/en/stable, 2019.

[7] unittest — unit testing framework. https://docs.python.org/3/library/unittest.html, 2019.

[8] nose. https://nose.readthedocs.io/en/latest/index.html, 2020.

[9] Numpy. https://numpy.org/, 2020.

[10] Keras integration test. https://github.com/keras-team/keras/tree/master/keras/integration_test, 2021.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *Proc. OSDI*, pages 265–283, 2016.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[13] M. Beller, C. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer. What it would take to use mutation testing in industry-a study at facebook. *CoRR*, abs/2010.13464, 2020.

[14] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[15] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Wardefarley, I. Goodfellow, and A. Bergeron. Theano: Deep learning on gpus with python. In *Proc. Nips, BigLearning Workshop*, 2011.

[16] G. J. Echternacht. The use of confidence testing in objective tests. *Review of Educational Research*, 42(2):217–236, 1972.

[17] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *Proc. ASE*, pages 810–822, 2019.

[18] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE software*, 8(2):58–65, 1991.

[19] W. G. Hatcher and W. Yu. A survey of deep learning: Platforms, applications and emerging research trends. *IEEE Access*, 6:24411–24432, 2018.

[20] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[21] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *Proc. ASE*, pages 1158–1161, 2019.

[22] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *Pro. ESEC/FSE*, pages 510–520, 2019.

[23] G. Jahangirova and P. Tonella. An empirical evaluation of mutation operators for deep learning systems. In *Proc. ICST*, pages 74–84, 2020.

[24] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu. The symptoms, causes, and repairs of bugs inside a deep learning library. *The Journal of Systems and Software*, page to appear, 2021.

[25] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. Proc. MM, pages 675–678, 2014.

[27] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proc. ESEC/FSE*, pages 654–665, 2014.

[28] D. Kawrykow and M. P. Robillard. Improving api usage through automatic detection of redundant code. In *Proc. ASE*, pages 111–122, 2009.

[29] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. L. Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, 2018.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS*, pages 1106–1114, 2012.

[31] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proc SIGSOFT/FSE*, pages 643–653, 2014.

[32] 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 *Proc. ASE*, pages 120–131, 2018.

[33] 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. In *Proc. ISSRE*.

[34] M. Nejadgholi and J. Yang. A study of oracle approximations in testing deep learning libraries. In *Proc. ASE*, pages 785–796, 2019.

[35] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proc. ICSE*, pages 537–548, 2018.

[36] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proc. SOSP*, pages 1–18, 2017.

[37] Y. Peng, Y. Zhang, and M. Hu. An empirical study for common language features used in python projects. In *Proc. SANER*, pages 24–35, 2021.

[38] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proc. ICST Workshops*, pages 47–53, 2018.

[39] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proc. ICSE*, pages 1027–1038, 2019.

[40] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proc.ASE*, pages 771–783, 2020.

[41] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proc SIGSOFT/FSE*, page 33, 2012.

[42] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[43] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proc. NIPS*, pages 3104–3112, 2014.

[44] S. Thorve, C. Sreshtha, and N. Meng. An empirical study of flaky tests in android apps. In *Proc ICSME*, pages 534–538, 2018.

[45] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proc. ICSE*, pages 303–314, 2018.

[46] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *Proc. ICSME*, pages 101–110, 2015.

[47] Z. Wang, K. Liu, J. Li, Y. Zhu, and Y. Zhang. Various frameworks and libraries of machine learning and deep learning: A survey. *Archives of computational methods in engineering*, pages 1–24, 2019.

[48] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim. An empirical study of common challenges in developing deep learning applications. In *Proc. ISSRE*, pages 104–115, 2019.

[49] Y. Zhang, Y. Chen, S. Cheung, Y. Xiong, and L. Zhang. An empirical study on TensorFlow program bugs. In *Proc. ISSTA*, pages 129–140, 2018.