How Do Injected Bugs Affect Deep Learning?

Li Jia*, Hao Zhong*, Xiaoyin Wang†, Linpeng Huang*, Zexuan Li*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†Department of Computer Science, University of Texas at San Antonio, USA
insanelung@sjtu.edu.cn, zhonghao@sjtu.edu.cn, Xiaoyin.Wang@utsa.edu, huang-lp@cs.sjtu.edu.cn, lizx_17@sjtu.edu.cn

***Abstract*—In recent years, deep learning obtains amazing achievements in various fields, and has been used in safety-critical scenarios. In such scenarios, bugs in deep learning software can introduce disastrous consequences. To deepen the understanding on bugs in deep learning software, researchers have conducted several empirical studies on their bug characteristics. In the prior studies, researchers analyzed the source code, bug reports, pull requests, and fixes of deep learning bugs. Although these studies provide meaningful findings, to the best of our knowledge, no prior studies have explored the runtime behaviors of deep learning bugs, because it is rather expensive to collect runtime impacts of deep learning bugs. As a result, some fundamental questions along with deep learning bugs are still open. For example, do most such bugs introduce significant impacts on prediction accuracy? The answers to these open questions are useful to a wide range of audience. In this paper, we conducted the first empirical study to analyze the runtime impacts of deep learning bugs. Our basic idea is to inject deliberately designed bugs into a typical deep learning application and its libraries with a mutation tool, and to compare the runtime differences between clean and buggy versions. In this way, we constructed 1,832 buggy versions, and compared their execution results with corresponding clean versions. Based on our comparison, we summarize 9 findings, and present our answers to 3 research questions. For example, we find that more than half of buggy versions do not lead to any observable errors, and most of them introduce only insignificant differences on the accuracy of their trained models. We interpret the significance of our findings from the perspectives of application programmers, API developers, and researchers. For example, based on our findings, better results alone are insufficient to prove better parameters nor better treatments, and researchers shall build strong theories to explain their improvements.**

I. Introduction

As a broader family of machine learning techniques, deep learning has been used in various fields (*e.g.*, computer vision [48] and natural language processing [17], [73]). In software engineering, researchers also have used deep learning to handle domain-specific problems (*e.g.*, generating code comments [35] and locating faults [79]). In recent years, deep learning has found its way to more safe-critical scenarios (*e.g.*, self-driving cars [23]), but a bug in such scenarios can lead to disastrous consequences [85]. Meanwhile, a recent study [39] reports that deep learning is not fully tested.

To deepen the knowledge on deep learning bugs, researchers have conducted various empirical studies on bugs in deep learning clients [38], [84] and libraries [40]. In these studies, to understand bugs and their fixes, researchers have analyzed many bug reports [40], [84], pull requests [40], and threads on StackOverflow [38], [84]. As it is too expensive to recreate and execute real bugs, these studies did not analyze the runtime behaviors of deep learning bugs. As a result, many questions along with deep learning bugs are still open. For example, do such bugs produce observable error messages, and do they introduce significant differences in the accuracy of trained models? The answers to these questions are useful to improve the quality of deep learning software.

It is challenging to conduct an empirical study to answer the above questions.

Challenge 1. It is expensive to recreate and execute deep learning bugs. To execute a bug, researchers shall locate the exact buggy commits and their testing inputs. As this process is often expensive, in the prior studies researchers [38], [40], [84] never execute all deep learning bugs in their studies.

Challenge 2. It is challenging to analyze many bugs. As it needs much expertise to understand deep learning bugs, in all the prior studies [38], [40], [84], researchers analyzed only hundreds of bugs. As a result, their findings can be not general.

Challenge 3. It is challenging to ensure the reliability of the analysis. Pham *et al.* [69] trained six popular neural network models on three datasets, and they find that identical training runs have around 10% random accuracy differences.

Researchers like to analyze real bugs in empirical studies, but analyzing real bugs can introduce a bias to our study (see the end of this section for more discussions). Mutation testing is widely used to inject bugs. In this study, we use such bugs to attack the above three challenges. In particular, as bugs are automatically injected, we bypass the first challenge. For the second challenge, in total, we used a mutation tool, called mutmut [9], to create 1,832 buggy versions. To attack the third challenge, following the guideline of Arcuri and Briand [15], we execute both mutants and clean versions multiple times, and use statistic tests to compare their results.

Although mutmut is designed to inject bugs to traditional software, it is sufficient for our study, since researchers report that bugs in deep learning libraries [40] and their applications [38], [84] are both quite similar to those in the traditional software. As it is too expensive to recreate real bugs, we have to substitute them with mutants. Researchers have intensively compared mutants with real faults, but their studies focus on whether mutation scores are correlated with the ability of a test suite to detect real faults [43], [66]. The above comparisons are irrelevant to our empirical study, since we do not calculate mutation scores. Instead, we care about whether mutants can produce similar buggy behaviors as real faults do. A recent survey [65] claims that this problem is rarely studied and is

still an open question. Gopinath *et al.* [31] implemented a tool to extract bug fixes, and compared the changed tokens between mutants and bug fixes. Even with advanced machine learning techniques, a recent work [58] achieved only around 70% fscore to identify bug fixes. As a result, their conclusions can be polluted by other types of code changes. Their study does not touch the runtime behaviors of bugs either. Meanwhile, an early study [24] reports that 85% of corrupted states from mutants are the same with those that are produced by real faults. Due to the above considerations, we believe that it is reasonable to inject bugs with our mutation tool.

Our study explores the following research questions:

- **RQ1. What are the symptoms of injected bugs?**
  Motivation. For programmers, the answers are useful to debug deep learning bugs, and for researchers, the answers are useful to explore better debug tools.
  Answer. More than half of buggy versions do not lead to observable errors (Finding 1). Although these bugs typically reduce the accuracy and increase the training time (Findings 2 and 3), most of them introduce insignificant differences and are difficult to be identified (Finding 4).
- **RQ2. What are the impacts of mutation operators?**
  Motivation. The answers are useful to understand the impacts of different bugs. Here, we put a bug into a category based on the bug's type of code modification during mutating, and analyze impacts by bug categories.
  Answer. We find that the impacts of bugs can be quite different. For example, the bugs caused by modifications on `if`-statements introduce more crashes (Finding 5), and some bugs can have more impact on the accuracy (*e.g.*, modifications on assignments, Findings 6 and 7) or the training time (*e.g.*, wrong comparisons, Finding 6).
- **RQ3. What are the impacts on deep learning phases?**
  Motivation. A deep learning program typically has three phases: preprocessing, constructing, and learning. The answers are useful to debug bugs in different phases.
  Answer. We find that normal outputs and crashes are equally distributed in the three phases (Finding 9). For those normal outputs, the bugs of the constructing and learning phases introduce more visible impacts on both accuracy values and training time than those of the preprocessing phase (Finding 8).

After we conduct this study, we realize that analyzing real bugs introduces a survival bias. According to our results, many injected bugs do not introduce observable differences in trained models, but it is difficult to detect such bugs, even if they are quite similar to real bugs. Analyzing real bugs thus ignores bugs that are similar to our injected bugs. Indeed, this bias can motivate in-depth discussions. For example, why can buggy code still train models whose accuracy is close to those models trained from clean code? How to detect such bugs?

II. Methodology

A. Our Selected Deep Learning Application

We select a sentiment classifier [45] as our subject application due to the following consideration: (1) sentiment classification is a classical natural language processing (NLP) task, and has been used in various applications [19], [21]; (2) convolutional neural network (CNN) [49] is widely used in various research fields [25], [63], and is often used with other deep learning models like recurrent neural network (RNN) [29]; (3) The problems of many deep learning applications [16], [22], [30] can be reduced to the classical classification problem; (4) our classifier has three implementations that are built on TensorFlow [14], Keras [8], and Theano [18], which are among the top ten of the most popular deep learning libraries [33], [80]; and (5) the classifier is published in a reputable NLP conference (EMNLP 2014) with more than 8,000 citations.

In this work, although we select only an NLP application, we analyzed thousands of its buggy versions, which are much more than the inspected bugs of the prior studies [38], [40], [84]. Indeed, in a recent empirical study, Pham *et al.* [69] also analyzed only a single application, *i.e.*, image classification. For this application, Pham *et al.* [69] chose three networks: LeNet [51], ResNet [34], and WideResNet [82]. As a comparison, we chose three implementations of our selected application to explore the impacts on deep learning libraries.

B. The Characteristic of Our Application

Table I shows our subject. Column "Implementation" shows the three implementations of our selected application. Column "Library" introduces their underlying deep learning libraries. As shown in Subcolumn "Name", the application is built on the three popular libraries such as TensorFlow, Theano, and Keras. Although MILA stopped developing Theano, we still selected its Theano implementation, since the github project of Theano [13] and a forked project called Aesara [12] are still under active maintenance. In our study, we inject bugs to both the applications and the libraries. We chose library versions newer than the minimum requirement and compatible with our selected application. The sizes of libraries are much larger than those of the implementations.

C. Dataset

Kim [45] used movie reviews to train the model. The dataset was released by Pang and Lee [64], and it is available on their website [1]. It contains 5,331 positive reviews and 5,331 negative reviews. In a movie review, a comment is associated with a subjective rating (*e.g.*, one star). The rating is used to label the sentiment of the comment. For example, *"one of the greatest family-oriented, fantasy-adventure movies ever"* is regarded as a positive review, but *"the movie is a mess from start to finish"* is regarded as a negative review.

D. Injected Bugs

In software testing, mutation testing [41] is a classical technique to inject bugs, and it is widely used in various research fields such as software testing [26], network protocols [78], web service [81], and deep learning system testing [55]. It implements various mutation operators, and each operator generates a type of buggy versions. After bugs are injected, researchers count how many injected bugs are

TABLE I: Our selected deep learning applications

Implementation (sentence classification)						Library						
URL	Language	File	LoC	Covered	%	Name	File	Tagged	Version	LoC	Covered	%
[3]	Python 2.7	4	369	287	77.8%	Theano	352	389	1.04	98,250	15,610	15.9%
[4]	Python 3.6	3	188	187	99.5%	TensorFlow	1,614	15,363	1.15	289,509	80,524	27.8%
[5]	Python 3.6	3	187	147	78.6%	Keras	91	7,210	2.31	18,299	4,806	26.3%

File: the number of source files; LoC: the line of code; Covered: the number of executed lines, when we trained the applications with the default setting and dataset (Section II-C); %: $\frac{Covered}{LoC}$; Tagged: GitHub projects with the library tags.

detected by a test suite to determine the quality of this test suite. The generated programs with defects are called mutants. To generate a mutant, researchers carefully design mutation operators, which change code in a specific way.

For deep learning, researchers have proposed various approaches to generate data inputs. For example, adversarial attacks [83] has been a hot research topic, and the approaches in this line (*e.g.*, [74]) mutate data to generate more challenging testing inputs, and their purpose is to measure the robustness of trained models. Besides data inputs, researchers (*e.g.*, [55], [72]) adapt mutation testing techniques to mutate trained models, and their purpose is to measure the quality of testing data. According to the IEEE glossary [37], a software bug or fault is an incorrect step or data definition in a program. The above approaches do not mutate code, and thus they do not inject bugs to deep learning systems. As our study focuses on the impacts of deep learning bugs, we do not choose them in our study, but use a classic mutation testing tool [41]. In particular, as our applications in Table I are all written in Python, we selected mutmut [9] to inject bugs, and we the eight mutation operators as listed in Table II, since they are shared by other mutation tools [2], [44], [56]. In Table II, the first column shows our selected eight operators. For example, the first row shows the arithmetic operator replacement. An arithmetic operator is replaced by an alternative, when the tool generates a buggy version. Here, $op1$ denotes an original operator, $op2$ denotes its target operator, and they construct a tuple to denote the mapping between two operators. For example, a code snippet is shown as below:

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

If 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 to introduce a bug:

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

A modification can introduce syntax errors. We ignored such modifications and did not consider them as bugs, because syntax errors lead to compilation errors and will be fixed before they become real bugs in deep learning code.

E. General Protocol

We collected the impact of a bug with the three steps:

Step 1. Collecting the execution results of original applications. To collect the results, we trained models with the three original implementations. During the training process, we used the dataset in Section II-C. Table III shows the default settings

TABLE II: Mutation operators

Operator	Description
ArOR	$op1 \leftrightarrow op2 \in \{+ \leftrightarrow -, * \leftrightarrow /, \% \leftrightarrow /, // \leftrightarrow /\}$
BitOR	$op1 \leftrightarrow op2 \in \{\& \leftrightarrow \mid, \wedge \leftrightarrow \&, <<\leftrightarrow>>\}$
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	$original\ value \leftrightarrow original\ value + 1$

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

of the three applications, and their default settings have minor differences. Column "loss function" shows the loss functions. A loss function defines how to minimize the distance between predictions and ground truth labels. By default, all three implementations use cross-entropy [7] as their loss functions. Here, cross-entropy is a measurement to quantify the distance between two probability distributions. Column "embedding dimension" shows the dimension of word embedding. Word embedding is a language modeling technique that translates the vocabulary into vectors of real numbers [59]. This column defines the dimensions of the vector for each word. Column "embedding initialization" shows the way to initialize word embedding. The TensorFlow implementation initializes word vectors with random values [11], but the other two implementations use a pre-trained model called word2vec [59]. Word2vec uses a shallow neural network model that is pre-trained on a large corpus. Compared to random values, its built vectors can reveal the similarity between words. Column "filter size" defines the parameters of filters. In a convolution operation, a filter is a matrix that defines a region to select the local feature of an input tensor. In a textual convolution, this matrix is a vector because word embedding encode words into one-dimension vectors. Column "batch size" shows the number of samples in a batch. To reduce memory cost, the training process on a dataset is divided into multiple passes. The batch size defines the number of samples that are propagated through the built network in each pass. Column "epoch number" shows the number of epochs in the whole training process. An epoch is a full training cycle on the whole training set, and a training process can contain more than one epoch. In our study, we have many buggy versions, and it is infeasible to execute them in an acceptable time limit. We notice that if we reduced the epochs, the accuracy values of trained models have only minor changes, but the training time can be significantly reduced: Theano from 8,000 seconds to 1,600 seconds and TensorFlow

TABLE III: The default settings of the three implementations

	loss function	embedding dimension	embedding initialization	filter size	batch size	epoch number
Theano	cross-entropy	300	word2vec	[3,4,5]	50	25
TensorFlow		128	random uniform distribution	[3,4,5]	64	200
Keras		50	word2vec	[3,8]	64	10

from 7,200 seconds to 600 seconds. As a result, we reduced the epoch number of the Theano implementation and the TensorFlow implementation to 5 and 20, respectively, to save time. Please note that for the Theano implementation and the TensorFlow implementations, we reduce the epoch numbers of both clean and buggy versions. As a result, the setting shall not introduce bias to our study. The average training time of the Keras implementation was about 60 seconds, so we did not change its default epoch number.

Some API calls (*e.g.*, `atomicAdd`) are nondeterministic on a GPU. To rule out their impacts, as Pham *et al.* [69] did, we trained all models on a single CPU thread. To rule out the impacts of settings, we use their original settings.

Step 2. Collecting the execution results of buggy versions. To obtain the results of the process with bugs, we used mutation tools described in Section II-D to generate buggy versions (i.e., mutants). This mutation tool is built upon a Python parser called parso [10], and it can build the AST of source code. During the executions of clean versions, we used a coverage analysis tool [6] to record all touched files and their covered lines. We executed all the implementations for 50 times. We found that their coverage remains unchanged. As a result, we believed that more executions may not lead to more covered code lines. For each line that was covered in the standard application, we analyzed whether its corresponding AST node can be mutated by a mutation operator. When a mutation operator was found, we used mutation tool to mutate this line, and generated a buggy version. If more than one mutation operator were found, we generated multiple buggy versions by iterating the found operators. As large files can have much more mutation nodes than small files, if we mutate all nodes, most mutants are the buggy versions of several large files, which introduces a bias to our study. To reduce this bias, in each buggy version, we limit the number of mutated nodes for each file. For Keras, we limited the nodes to 50, and for Theano and TensorFlow, we limited the nodes to 30, since the latter two libraries have more source files. As we did not compare the results from different libraries, this setting does not affect our results.

For each buggy version of an implementation, we trained the model on the same dataset, and we used the identical default settings. As there were many buggy versions, due to time limit, we executed each buggy version only once. For executed buggy versions, we recorded their outputs such as accuracy and training time. For versions which could not be executed, if the program stopped automatically, we recorded its error message; if an execution hanged, we manually stopped it and recorded error message. We then classified the results by their outputs.

Step 3. Comparing execution results. We conducted ten-fold cross validations for both clean and buggy versions. All three implementations report their classification accuracy, so we compared the accuracy values to analyze the impacts of bugs. Here, the accuracy is calculated by Equation 1, where N_c denotes the number of correct classification results and N_i denotes the number of incorrect classification results.

$$Accuracy = \frac{N_c}{N_c + N_i} \tag{1}$$

To compare the results of the clean versions and our buggy versions, we introduce two-tailed t-test [53] instead of one-way ANOVA, since this is a comparison between two sets of data. The null hypothesis is that the difference between a buggy version and a clean version is insignificant. If the hypothesis is rejected at the significance level $\alpha = 0.05$, we consider that the difference is significant. We further compared their means to determine whether the results were significantly increased or reduced. Please note that even if a difference is statistically insignificant, if it occurs in a safe critical application such as self-driving cars, it can lead to disastrous consequences.

TABLE IV: The overall distribution of symptoms

	Normal	Crash	Hang	Total
Theano	297 (57.45%)	217 (41.97%)	3 (0.58%)	517
TensorFlow	266 (57.70%)	195 (42.29%)	0	461
Keras	538 (62.99%)	315 (36.89%)	1 (0.12%)	854

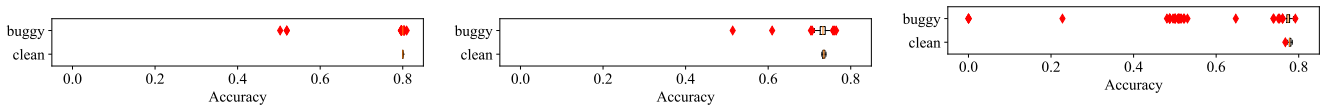
III. Empirical Result

This section presents the results of our study. More details are listed on our project website:

https://github.com/bugdataupload/deeplearningbugs

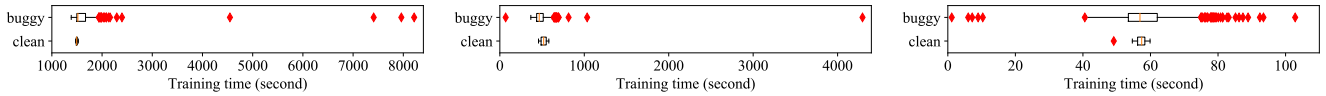
A. RQ1. Runtime Symptoms

1) Protocol: As introduced in Section II-E, we trained a model on every buggy version, and stored its output to a log. After that, we implemented a tool to check the log. If it produced no crashes or errors, we extracted its accuracy and training time. In addition, when mutmut generated buggy versions, our tool recorded the link between each buggy version and its mutation information (*i.e.*, the mutation operator, the mutated file, the modified line number, and the execution log). We analyzed the results of each implementation with four steps. First, based on the logs of buggy versions, we classified their outputs into categories. Second, we manually inspected some buggy versions to understand their impacts. Third, for those buggy versions that produce normal outputs, we drew box plots to show the distributions of their accuracy values and training times. Finally, we compared the accuracy values and training times between clean versions and buggy versions.



(a) Theano (b) TensorFlow (c) Keras

Fig. 1: The accuracy distribution



(a) Theano (b) TensorFlow (c) Keras

Fig. 2: The distribution of training time

2) Results: Table IV shows three types of symptoms:

1. Normal outputs. Column "Normal" lists the number of buggy versions with normal outputs. These buggy versions terminate with outputs whose formats are consistent with the outputs of clean versions. Besides their values, we cannot find any traces of the injected bug. However, several outliers can produce abnormal values. For example, such a outlier occurs in the `result` method of the `Reduce` class:

```
def result(self):
  if self.reduction == metrics_utils.Reduction.SUM:
    return self.total
  elif self.reduction in [
    metrics_utils.Reduction.WEIGHTED_MEAN,
    metrics_utils.Reduction.SUM_OVER_BATCH_SIZE]:
  return self.total / self.count
```

Given a value whose type is `Reduce`, this method calculates a value based on the value of `self.reduction`. The return value is used to calculate the overall accuracy of a trained model. In Line 7, our mutation tool replaces the arithmetic operator from "/" to "*". The `Reduce` class calculates output metrics, and the above modification causes a wrong calculation of the accuracy. In this example, we obtain a wrong accuracy, 884,543.00. Although the value is illegal (greater than one), we put it into the normal category, because it produces no error messages.

2. Crashes. Here, we consider a raised exception as a crash. For example, the following method throws an exception:

```
if cost is None:
  if known_grads is None:
    raise AssertionError("...can't␣both␣be␣None.")
```

In Line 1, the mutation tool changes `None` to `not None`. As the `cost` is not `None`, the mutated code throws exceptions.

3. Hangs. For example, the `_add_unique_metric_name` method of the `Model` class has a `while` statement as follows:

```
while metric_name in self.metrics_names:
  metric_name = '%s_%d' % (base_metric_name, j)
  j += 1
```

In Line 1, `in` is replaced with `not in`. As Line 2 does not modify `metric_name`, the loop becomes infinite.

Finding 1. More than half of the buggy versions do not produce observable errors (57.45% to 62.99%); fewer than half of the buggy versions crash (36.89% to 42.29%); and several buggy versions hang (0% to 0.58%).

For the buggy versions that produce normal outputs, we drew box plots to show the distributions of their accuracy values. Figure 1 shows the results. To make the figures more readable, in this figure, we removed five data points, because their accuracy values were far outstripped the legal range. For example, as introduced in the beginning of this section, the accuracy of a buggy version is 884,543.00, and this data point was removed from this figure. Please note that we still included these data points, when we compared the accuracy values between clean and buggy versions.

Figure 1 shows that most buggy versions only slightly change accuracy. Although we injected bugs to the executed lines, some bugs did not change the values of executions. For example, the `Reduce` class has the following code lines:

```
if self.reduction == metrics_utils.Reduction.
    SUM_OVER_BATCH_SIZE:
  num_values = K.cast(K.size(values), self.dtype)
elif self.reduction == metrics_utils.Reduction.
    WEIGHTED_MEAN:
  if sample_weight is None:
      num_values = K.cast(K.size(values), self.dtype)
  else:
      num_values = K.sum(sample_weight)
```

In Line 1, the mutation tool replaces "==" by "!=". When the mutated line was executed, the value of `self.reduction` was `WEIGHTED_MEAN`, so Line 2 is incorrectly executed. Although Line 5 is not executed which should be in original program, the final results keep unchanged because Line 2 and 5 are the same. However, when `self.reduction` is set to `SUM_OVER_BATCH_SIZE` under other settings, Line 2 should be executed but not, the above bug may lead to visible differences.

In a box plot, the minimum denotes the lowest data point excluding outliers, and the maximum denotes the largest data point excluding outliers. Figure 1 shows that most outliers are below the minimum. In particular, 31 out of 32 Keras outliers, 3 out of 7 Tensorflow outliers, and 16 out of 21 Theano outliers are below their minimums. After inspection, we notice that bugs in core algorithms can have more impact. For example, the `call` method of the `Dense` class implements an activation operation of a neural network. This operation is a non-linear transformation function, and belongs to the core algorithms. Its code lines are as follows:

```
def call(self, inputs): ...
      if self.activation is not None:
          output = self.activation(output)
      return output
```

TABLE V: The symptoms classified by bug types

	Theano				TensorFlow				Keras			
	Normal	Crash	Hang	Total	Normal	Crash	Hang	Total	Normal	Crash	Hang	Total
ArOR	9 (33.3%)	**18 (66.7%)**	0	27	**36 (72.0%)**	14 (28.0%)	0	50	39 (48.1%)	**42 (51.9%)**	0	81
BitOR	0	0	0	0	0	0	0	0	0	0	0	0
ComOR	41(36.9%)	**70 (63.1%)**	0	111	45 (48.9%)	**47 (51.1%)**	0	92	94 (49.0%)	**98 (51.0%)**	0	192
LogOR	55 (45.1%)	**65 (53.3%)**	2 (1.6%)	122	61 (47.3 %)	**68 (52.7 %)**	0	129	**88 (51.2%)**	84 (48.8%)	0	172
AsOR	**2 (66.7%)**	0	1 (33.3%)	3	**8 (80.0%)**	2 (20.0 %)	0	10	**10 (71.4%)**	4 (28.6%)	0	14
MemOR	5 (17.9%)	**23 (82.1%)**	0	28	6 (16.2%)	**31 (83.8%)**	0	37	16 (40.0%)	**23 (57.5%)**	1 (2.5%)	40
BVR	**78 (86.7%)**	12 (13.3%)	0	90	**51 (79.7%)**	13 (20.3%)	0	64	**67 (83.8%)**	13 (16.2%)	0	80
NVR	**107 (78.7%)**	29 (21.3%)	0	136	**59 (74.7%)**	20 (25.3%)	0	79	**224 (81.5%)**	51 (18.5%)	0	275

In Line 2, our buggy version replaces `is not` with `is`. Although `self.activation` is not `None`, this activation function is disabled, and its output value is passed to follow-up layers directly. As a result, the overall accuracy values are reduced.

Figure 1 shows that 1 Keras outliers and 3 TensorFlow outliers are beyond the maximum. Although some buggy versions produce higher accuracy, we find that even the results of such outliers are mostly accidental. For example, the code of the `compute_mask` method is as follows:

```
1 if cache_key in self._output_mask_cache:
2  return self._output_mask_cache[cache_key]
3 else:
4  _, output_masks,_=self.run_internal_graph(inputs,masks)
5  return output_masks
```

In Line 1, the buggy version replaces `in` with `not in`. Although the clean version and our buggy version go to different branches, we observe that both Lines 2 and 5 return `none`. We executed each buggy version for only once. Due to various random issues, the accuracy of this buggy version is higher than most clean versions. To fully understand this buggy version, we executed it for additional 10 times, and compared the results with the accuracy of clean versions. Our t-test comparison shows that the accuracy between the clean versions and this buggy version are not significantly different.

Finding 2. For the accuracy, 50 outliers are below the minimum, and only 10 outliers are beyond the maximum.

Figure 2 shows the distributions of training times. In total, 19 out of 19 Theano buggy outliers, 16 out of 17 TensorFlow buggy outliers and 31 out of 37 Keras buggy outliers are beyond the maximum. After inspection, we notice that some bugs can change the settings of the training process. For example, the `_validate_or_infer_batch_size` method sets `batch_size` to 32, if both `batch_size` and `steps` are `None`:

```
1 if batch_size is None and steps is None:
2             batch_size = 32
```

The buggy version changes the condition to set the value:

```
1 if batch_size is not None and steps is None:
2             batch_size = 32
```

In our setting, the batch size is 64 and variable `steps` is `None`. After the modification, in the buggy version, `batch_size` is modified to 32. As we introduce in Section III-A1, reducing batch size increases the number of training passes, and more passes increases the training time from about 60 seconds to about 100 seconds.

Figure 2 shows that 1 out of 17 TensorFlow buggy outliers and 6 out of 37 Keras buggy outliers are below the minimum. We notice that some bugs can break the training process. For example, the `fit_loop` method has a code line:

```
1 callbacks.model.stop_training = False
```

The mutation tool replaces `False` with `True`. When this variable is `True`, the training loop will break:

```
1 for epoch in range(initial_epoch, epochs):...
2         if callback_model.stop_training: break
```

As a result, the training program does not loop enough times as the predetermined epoch number, the training time is thus reduced from about 60 seconds to 6 seconds.

Finding 3. For the training time, 7 outliers are below the minimum, and 66 outliers are beyond the maximum.

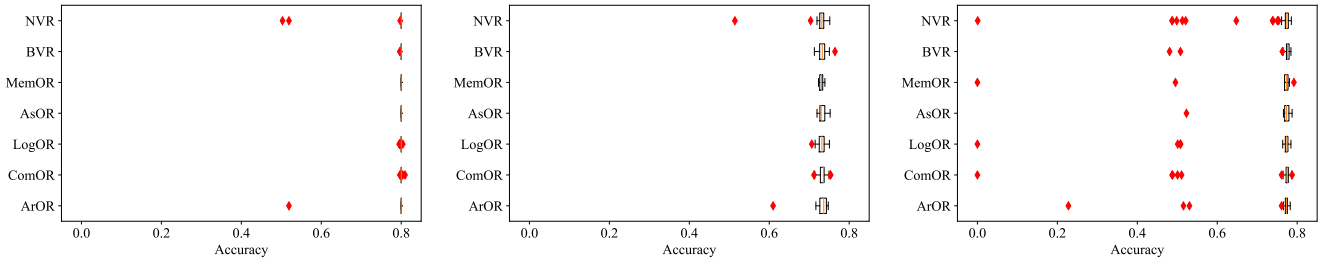
Figures 1 and 2 show that our injected bugs have different impacts on the three implementations. For the accuracy, the impacts on Theano and TensorFlow are both minor, and the impact on Keras is more visible. For the training time, most buggy versions of Theano and TensorFlow increase the time, but the impacts on Keras are more diverse. We compared the prediction accuracy of buggy versions with clean versions. Our results show that for Theano and TensorFlow, the differences are insignificant, but the differences of Keras are significant. As introduced in the first example of this section, in total, 5 buggy versions produced invalid accuracy values (*e.g.*, an accuracy that is greater than one). After we removed the 5 outliers, the differences of Keras are also insignificant. The result leads to a finding.

Finding 4. If buggy versions do not crash or hang, their accuracy medians are close to those of clean versions.

In summary, more than half of buggy versions produce normal outputs without observable errors. Although most of them cause accuracy reduction and increase training times, their differences from clean versions are insignificant.

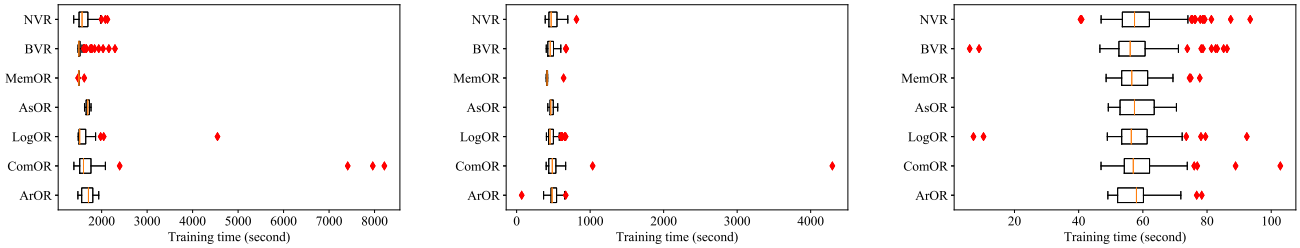
B. RQ2. Impacts of Different Bug Types

1) Protocol: RQ1 explores the overall impacts of bugs. In this research question, we analyze the impacts of different bug types. Table II shows our mutation operators. Each type of operators introduces a type of bugs. We classified our buggy versions by their corresponding mutation operators. For each category of bugs, we re-conducted our analysis in RQ1 to explore the impacts of different bug types. In addition, we manually inspected 35 crashes to explore their causes.



(a) Theano (b) TensorFlow (c) Keras

Fig. 3: The distribution of accuracy classified by bug types



(a) Theano (b) TensorFlow (c) Keras

Fig. 4: The distribution of training time classified by bug types

2) Results: We manually classified 35 crashes as follows:

1. `Raise` and `assert` statements in `if` statements (11, 31.4%). For example, in Section III-A2, we have proposed such an example, when we introduce the crashes of Table IV.

2. Wrong assignments via `if` statements (9, 25.7%). For example, Keras has an `if` statement as follows:

```
1 if shape is not None and not batch_shape:
2         batch_shape = (None,) + tuple(shape)
```

When Line 1 is executed, `batch_shape` is `None`, `shape` is not `None`. As a result, Line 2 is executed and `batch_shape` is assigned to `tuple(shape)`. The buggy version removes the first `not` from Line 1. As the condition of Line 1 is now false, Line 2 of the buggy version is not executed, and `batch_shape` remains `None`. In the following code, `batch_shape` is fed to a method as follows:

```
1 input_layer = InputLayer(batch_input_shape=batch_shape,
      name=name, dtype=dtype, sparse=sparse, input_tensor=
      tensor)
```

The above code throws an exception, because the `__init__` method of the `InputLayer` class has a check and `input_shape` is also `None`:

```
1  if not input_shape and not batch_input_shape:
2                      raise ValueError(...)
```

3. Type errors caused by incorrect operators (8, 22.9%). The `_init_graph_network` method of the `Network` class has a code line:

```
1 mask_cache_key += '_' + object_list_uid(masks)
```

The buggy version replaces "`+`" with "`-`". In Python, the subtraction of two strings are undefined. As a result, the buggy code throws an exception.

4.Dimension mismatches (4, 11.4%). For example, the `Input` method of class `InputLayer` has a code line:

```
1 outputs = input_layer._inbound_nodes[0].output_tensors
```

The buggy version replaces index `0` with `1`, and throws an exception. Its message says `list index out of range`.

5. Wrong assignments (3, 8.6%). For example, the `__init__` method of the `InputLayer` class has a code line:

```
1 def __init__(self,input_shape=None,input_tensor=None
      ,...):...
2         if input_tensor is None:
3                 self.is_placeholder = True
```

In Line 3, the buggy version changes `True` to `False`. This modification affects the following code:

```
1  if K.is_placeholder(v):
2          self._feed_input_names.append(name)...
```

As the above code does not append `name`, the following code raises an exception:

```
1 def standardize_input_data(data,names,shapes=None,...):
2   if not names:
3     raise ValueError('Error when checking model'...)...
```

Table V shows our operators and the symptoms of their corresponding buggy versions. For each operator, we calculated the proportion of its symptoms in the brackets. Based on our manual inspection, more than 50% crashes are caused by `if`-statements. ComOR MemOr and LogOR can often change `if`-statements, but NVR seldom directly modify `if`-statements. The observation is largely consistent with results of Table V. In this table, we also find that ComOR, MemOR, and LogOR lead to more crashes, and BVR and NVR lead to more normals.

Finding 5. ComOR, MemOR and LogOR introduce more crashes than other mutation operators, since the three operators often change `if`-conditions and more than half of crashes are caused by such changes.

For those normal outputs, Figure 3 and 4 shows the distribution of accuracy values and training times, for different operators, respectively. As NVR, LogOR and ComOR often change the computation process, Figures 3 shows that they have more impact on prediction accuracy. For example, the `binary_crossentropy` method is as following:

```
1 def binary_crossentropy(y_true, y_pred, from_logits=
      False, label_smoothing=0):...
2     if label_smoothing is not 0:...
3         y_true = K.switch(K.greater(smoothing, 0),
              _smooth_labels, lambda: y_true)
4     return K.categorical_crossentropy(y_true, y_pred,
          from_logits=from_logits)
```

In Line 1, NVR changes the initial value of `label_smoothing` from `0` to `1`. As a result, Line 3 is executed, and `y_true` is calculated in a different way. As `y_true` is the ground truth to calculate prediction accuracy, its new values significantly reduce accuracy.

Figure 4 shows that ComOR has more impact on training times. ComOR often modifies conditions of loop statements, and thus increase the training time. In addition, it can change the initial values that determine the conditions to terminate the training. For example, the `conv2d` method in Theano has the code lines as follows:

```
1 def conv2d(...):...
2         if image_shape is not None:
3                 bsize = image_shape[0]
4                 imshp = image_shape[1:]
5         else:
6                 bsize, imshp = None, None
```

The buggy version removes `not` from Line 2, so `bsize` and `imshp` are assigned to `None`. Theano can select the fastest algorithm to optimize the training if `bsize` is not `None`. After `bsize` is assigned to `none`, the optimization is disabled, and it increases the training time.

Finding 6. NVR, LogOR and ComOR can produce buggy versions whose prediction accuracy values are much lower than others, and ComOR can increase the training time much more than others.

We compare the accuracy values of different operators with the one-way ANOVA [53]. The result shows that 48.76% comparisons in Theano, 91.83% comparisons of Keras, and all the comparisons of TensorFlow are significantly different. As operators produce significant differences in most cases, we further compared buggy versions of each operator with clean versions. We find that NVR and BVR significantly reduce the accuracy values of Theano; AsOR significantly increases the accuracy values of TensorFlow; AsOR and MemOR significantly reduce the accuracy values of Keras.

Finding 7. Mutation operators produce significantly different accuracy values of their corresponding buggy versions, and several operators can produce significate deviations from accuracy values of clean versions.

C. RQ3. Impacts on Deep Learning Phases

1) Protocol: In this research question, we analyze the impacts of bugs on different deep learning phases. A deep learning task typically include three phases such as preprocessing, constructing, and learning [71]. In deep learning applications, raw data are stored in different formats (*e.g.*, texts, images, and videos). As the first phase, before raw data are fed into the neural networks, a preprocessing stage is introduced to transfer them into mathematical representations. For example, word embedding [52] is used to transfer text into vectors in our subject application. As the second phase, the structure of a deep model is constructed. The structure includes the applied network unit like CNN and RNN, the number of network layers and parameters, the auxiliary components (*e.g.*, dropout layers). As the phase of learning, the core function of deep model is applied. The training data are imported to train the model, and a back-propagation algorithm is used to minimize overall loss [50]. After a model is trained, the testing data are fed to the trained model to produce outputs, and its outputs are compared to a golden standard (labels) to judge the effectiveness of the model. In study, we cannot separate training and testing into two phases, because Keras implement its training and testing in a single method, `fit`.

To determine which phases a bug can influence, we separated each application program into a preprocessing phase, a constructing phase, and a learning phase. In Table VI, we show the lines of code for each phase. We executed the applications, and compared its coverage with a buggy location to determine whether the buggy location is touched by a phase. If a buggy location is commonly used, it can affect more than one phase. For example, common APIs like loss function and activation are both necessarily called in building network structure and computing output through network. In this case, we counted the bug twice in both the constructing and learning phases.

2) Result: Before we introduce the result, we introduce some bug samples in different phases:

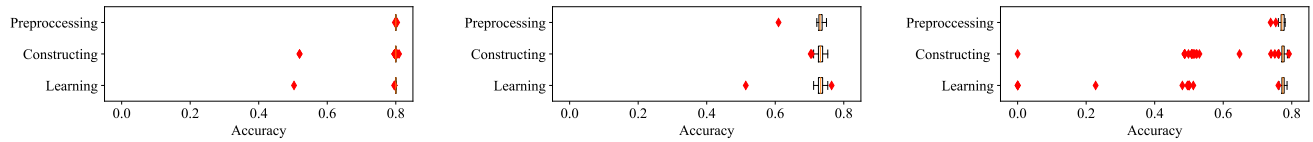
1. A bug sample in the preprocessing phase. Many bugs in the preprocessing phase reside in the operations of vectors, and can cause related problems such as indexes out of ranges. For example, the Keras application has the following code:

```
1 # Data Preparation
2  if sequence_length != x_test.shape[1]:
3     sequence_length = x_test.shape[1]
```

In the above code sample, `sequence_length` is the size of the testing data. In a buggy version, `x_test.shape[1]` is modified to `x_test.shape[2]`, and it leads to `IndexError`.

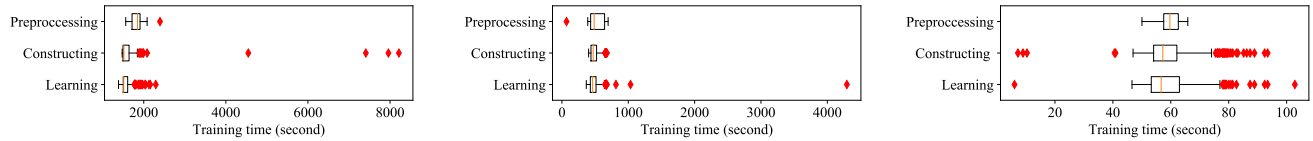
2. A bug sample in the constructing phase. We notice that dimension mismatches [40] often occur in the constructing phase. For example, in the constructing phase of the Keras application has the following code:

```
1 # Convolutional block
2 conv_blocks = []
3 for sz in filter_sizes:
4  conv = Convolution1D(filters=num_filters,kernel_size=sz
       ,padding="valid",activation="relu",strides=1)(z)
5  conv = MaxPooling1D(pool_size=2)(conv)
```



(a) Theano (b) TensorFlow (c) Keras

Fig. 5: The distribution of accuracy classified by bug phases



(a) Theano (b) TensorFlow (c) Keras

Fig. 6: The distribution of training time classified by bug phases

TABLE VI: The impacts on deep learning phases

Name	Phase	Normal	Crash	Hang	Total
Theano	P (82)	**19 (55.9%)**	15 (44.1%)	0	34
	C (166)	**233 (58.4%)**	163 (40.9%)	3 (0.7%)	399
	L (121)	**151 (52.4%)**	134 (46.6%)	3 (1.0%)	288
TensorFlow	P (32)	**18 (85.7%)**	3 (14.3%)	0	20
	C (84)	**232 (55.1%)**	189 (44.9%)	0	421
	L (72)	**214 (53.0%)**	190 (47.0%)	0	404
Keras	P (91)	**16 (59.3%)**	11 (40.7%)	0	27
	C (93)	**373 (63.2%)**	216 (36.6%)	1 (0.2%)	590
	L (3)	**272 (59.3%)**	186 (40.5%)	1 (0.2%)	459

P: the preprocessing phase; C: the constructing phase; L: the learning phase. In Column "Phase", the numbers inside the brackets denote the lines of code that appear in the corresponding phases of applications.

```
6   conv = Flatten()(conv)
7   conv_blocks.append(conv)
8 z = Concatenate()(conv_blocks) ...
9 z = Dropout(dropout_prob[1])(z)
10 z = Dense(hidden_dims, activation="relu")(z)
```

The above code builds the structure of a deep learning network. Line 3 iterates each filter size. For each size, a `conv` unit is generated by combining a convolution layer (Line 4) and a pooling layer (Line 5), and a tensor is transformed from multiple dimensions to a block by the `Flatten` operation (Line 6). After that, the block is concatenated to a list (Line 8), and a dropout and dense layer are added to the end of the network. Line 6 calls the `compute_output_shape` method of `Flatten` to generate the output shape:

```
1 def compute_output_shape(self, input_shape):...
2   return (input_shape[0], np.prod(input_shape[1:]))
```

In a mutant, `input_shape[0]` is modified to `input_shape[1]`, and it leads to a dimension mismatch.

3. Bug samples in the learning phase. Our learning phase includes both the training and testing of a deep learning model. Many bugs in core APIs can affect both training and testing. For example, as we introduced in Section III-A2, a bug can disable the activation function, and thus influence both phases. During the training process, the weights of a deep learning network are updated, and bugs on weights affect only the training process. For example, the TensorFlow application has a code line as follows:

```
1 optimizer = tf.train.AdamOptimizer(1e-3)
```

The argument of the above code is the learning rate, and it determines the speed of gradient descent [46]. A buggy version modifies it from `1e-3` to `1.001`. A correct learning rate must be less than 1, so the modified value `1.001` influences the accuracy significantly.

For those buggy versions without crashes, Figure 5 and 6 show the distribution of accuracy values and training time by different deep learning phases, respectively. The outliers in the preprocessing phase are fewer and those of the other two phases. This observation leads to a finding:

Finding 8. The buggy versions in the constructing and learning phases have more impact on both accuracy values and training time than those of the preprocessing phase.

Comparing to the preprocessing phase, the bugs of the other phases can affect the learning process. As shown in our bug sample in the learning phase, some bugs can directly lead to more visible differences.

Table VI shows the impacts on deep learning phases. Although the lines of code in the preprocessing phases do not differ much from those in the learning phases, the learning phases have much more bugs. As we injected bugs to both application code and API code, some few learning-related lines in application code can call many lines of API code. Except the preprocessing phase of TensorFlow version application, the distributions lead to another finding:

Finding 9. In all the phases, the distributions of normal outputs and crashes are largely the same.

In summary, the bugs in the preprocessing phase have less visible impacts than those in the constructing and learning phases, but as for normal outputs and crashes, the bugs in the three phases typically show minor differences.

D. Threats to Validity

The internal threats include equivalent mutants. As such mutants are semantically equivalent to original programs [47], our results can underestimate the derivation of buggy versions. The external threats to validity include our analyzed subjects and our injected source files. Although we selected a popular

deep learning application with three different implementations, our selected subjects were limited to this application and the code structures of our injected files. To reduce the threat, we select a neural model that is widely used and often used with other models, and it could be further reduced by analyzing more types of deep learning models.

IV. Interpretation

The significance of our findings are as follows:

Developing deep learning applications. When applications do not achieve their expected results, researchers often blindly tune their parameters and try different treatments. Our results show that an unexpected low result can also be caused by bugs either in their applications or deep learning libraries. Meanwhile, researches are satisfied when they find that their results are better than the prior ones. However, our results show that *better results alone do not justify better parameters nor better treatments,* in that better results can be caused by bugs. This type of problems can be relieved by more human inspection on potential bugs. However, ultimately, *researchers shall build strong theories to explain their improvements based on better results*. The bugs in the learning phase have more visible impacts than the preprocessing phase, and comparable impacts to the constructing phase (Finding 8), but more than half of bugs do not have observable error messages (Finding 1). Nejadgholi and Yang [60] report that the test cases in deep learning code use more oracle approximations than those of the traditional software. As oracle approximations consider a range of values as legal, such test cases are unlikely to reveal bugs that cause minor different outputs. In critical applications, a minor difference can be significant, and their programmers shall pay more attention to minor differences.

Developing deep learning libraries. We find that more than half of bugs do not have observable error messages (Finding 1) and most of them insignificantly change accuracy (Finding 4). Nejadgholi and Yang [61] report approximations are common in the test cases of deep learning libraries. Such test cases can silently ignore the above bugs. Crashes and their messages explicitly warn library users of bugs, but Finding 9 shows that in all the phases, more than half of bugs do not produce any crashes. The library developers can provide more internal states checking and messages display, so that detecting and localizing bugs can be more efficient. We also find that although changing program structures or logic often cause crashes (Finding 6), if a bug does not change program structures or logics (*e.g.*, NVR and BVR), it is less possible to cause crashes (Finding 5). This finding can be useful in debugging performance issues.

Detecting bugs in deep learning software. Although prior approaches [68], [77] detect bugs in trained models, we find that most of our injected bugs do not lead to observable differences of their accuracy (Finding 1). Although injected bugs do not introduce any observable differences as a whole, we notice that different types of bugs can produce quite different results (Finding 7). It indicates some types of bugs can be detected more easily. In future work, we plan to start from such bugs, and explore bug detection approaches.

V. Related Work

Empirical studies on bugs. Tan *et al.* [75] studied bugs from open source projects such as the Linux kernel and Mozilla. Thung *et al.* [76] analyze the bugs of machine learning systems. Zhang *et al.* [84] analyze the bugs in application code that calls TensorFlow. Islam *et al.* [38] analyze the application bugs of more deep learning libraries. Jia *et al.* [40] analyze the bugs inside deep learning library TensorFlow. Humbatova *et al.* [36] introduce a taxonomy of faults in deep learning systems, and Islam *et al.* [38] analyze their repair patterns. Most prior approaches analyze the static characteristics of bugs, but our study analyzes the dynamic characteristics that are observed during executions.

Detecting deep learning bugs. Pei *et al.* [67] propose a white-box framework to test real-world deep learning systems. Ma *et al.* [54] propose a set of multi-granularity criteria to measure the quality of test cases for deep learning systems. Tian *et al.* [77] and Pham *et al.* [68] introduce differential testing to discover bugs in deep learning software. Our findings are useful for researchers to design better bug detection tools.

Mutation testing. Some early work of mutation testing can be dated back to 1970s [27], [32], [62]. As the programming language developed rapidly, mutation testing techniques have been applied to different languages including Fortran [20], C [70], Java [57], C# [28], and so on. Meanwhile, mutation testing is also widely used in various research fields such as software testing [26], network protocols [78] and web service [81]. These prior works focus on traditional software, but we apply it to deep learning software. As a result, our empirical study is able to report runtime behaviors of deep learning bugs, which are not analyzed in prior studies.

VI. Conclusion and Future Work

To understand the runtime impacts of bugs on deep learning, with a mutation testing tool, we injected 1,832 bugs to 3 deep learning applications. Based on the execution results of these bugs, we classified and analyzed their overview characteristics, and also explored the impacts of bug types and deep learning phases. Our analysis lead to 9 findings, and we interpret these findings from the perspectives of application programmers, API developers, and researchers. Our study presents a comprehensive understanding on the dynamic characteristics of bugs in deep learning software.

In future work, we plan to analyze more deep learning models (*e.g.*, RNN [42]) and the impacts of multiple mutation operators. As many of our injected bugs do not introduce observable differences, we plan to explore more advanced techniques to detect such bugs.

Acknowledgement

We appreciate reviewers for their insightful comments. This work is sponsored by a CCF-Huawei Innovation Research Plan. Hao Zhong is the corresponding author.

REFERENCES

[1] The movie review data in sentiment-analysis experiments. http://www.cs.cornell.edu/people/pabo/movie-review-data/, 2012.

[2] The pit mutation testing tool. http://pitest.org/, 2017.

[3] Convolutional neural networks for sentence classification. https://github.com/yoonkim/CNN_sentence, 2018.

[4] Convolutional neural network for text classification in tensorflow. https://github.com/dennybritz/cnn-text-classification-tf, 2019.

[5] Convolutional neural networks for sentence classification. https://github.com/alexander-rakhlin/CNN-for-Sentence-Classification-in-Keras, 2019.

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

[7] Cross entropy. https://en.wikipedia.org/wiki/Cross_entropy, 2019.

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

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

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

[11] tf.random.uniform. https://tensorflow.google.cn/versions/r1.15/api_docs/python/tf/random/uniform, 2019.

[12] Aesara, a forked theano. https://github.com/pymc-devs/aesara, 2021.

[13] Theano github project. https://github.com/Theano/Theano, 2021.

[14] 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.

[15] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. ICSE*, pages 1–10, 2011.

[16] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixão, F. Mutz, et al. Self-driving cars: A survey. *Expert Systems with Applications*, page 113816, 2020.

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

[18] 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.

[19] E. Boiy and M. Moens. A machine learning approach to sentiment analysis in multilingual web texts. *Inf. Retr.*, 12(5):526–558, 2009.

[20] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. The design of a prototype mutation system for program testing. In *Proc. AFIPS*, pages 623–629, 1978.

[21] L. Chen, C. Liu, and H. Chiu. A neural network based approach for sentiment classification in the blogosphere. *J. Informetrics*, 5(2):313–322, 2011.

[22] Y. Chen, Z. Lin, X. Zhao, G. Wang, and Y. Gu. Deep learning-based classification of hyperspectral data. *IEEE Journal of Selected topics in applied earth observations and remote sensing*, 7(6):2094–2107, 2014.

[23] M. Daily, S. Medasani, R. Behringer, and M. Trivedi. Self-driving cars. *Computer*, 50(12):18–23, 2017.

[24] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes*, 21(3):158–171, 1996.

[25] N. De Rita, A. Aimar, and T. Delbruck. Cnn-based object detection on low precision hardware: Racing car case study. In *Proc. IV*, pages 647–652, 2019.

[26] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Software Eng.*, 27(3):228–247, 2001.

[27] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[28] A. Derezinska. Quality assessment of mutation operators dedicated for c# programs. In *Proc. (QSIC*, pages 227–234, 2006.

[29] Y. Fan, X. Lu, D. Li, and Y. Liu. Video-based emotion recognition using cnn-rnn and c3d hybrid networks. In *Proc. ICMI*, pages 445–450, 2016.

[30] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

[31] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Proc. ISSRE*, pages 189–200, 2014.

[32] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, 1977.

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

[34] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. CVPR*, pages 770–778, 2016.

[35] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, pages 1–39, 2019.

[36] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *Proc. ICSE*, page to appear, 2020.

[37] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[38] 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.

[39] L. Jia, H. Zhong, and L. Huang. The unit test quality of deep learning libraries: A mutation analysis. In *Proc. ICSME*, page to eappear, 2021.

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

[41] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[42] M. I. Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. 1997.

[43] 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.

[44] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In *Proc. ASE*, pages 612–615, 2011.

[45] Y. Kim. Convolutional neural networks for sentence classification. In *Proc. EMNLP*, pages 1746–1751, 2014.

[46] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, 2015.

[47] 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.

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

[49] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[50] Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[51] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[52] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014.

[53] R. Lowry. Concepts and applications of inferential statistics. http://vassarstats.net/textbook/, 2014.

[54] 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.

[55] 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*, pages 100–111, 2018.

[56] Y. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[57] Y. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test. Verification Reliab.*, 15(2):97–133, 2005.

[58] N. Meng, Z. Jiang, and H. Zhong. Classifying code commits with convolutional neural networks. In *Proc. IJCNN*, page to appear, 2021.

[59] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proc. NIPS*, pages 3111–3119, 2013.

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

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

[62] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. *Mutation Testing for the New Century*, 2001.

[63] Z. Ouyang, J. Niu, Y. Liu, and M. Guizani. Deep cnn-based real-time traffic light detector for self-driving vehicles. *IEEE transactions on Mobile Computing*, 19(2):300–313, 2019.

[64] B. Pang and L. Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proc. ACL*, 2005.

[65] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. 2019.

[66] 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.

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

[68] 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.

[69] 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.

[70] H. A. Richard, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the c programming language. *techreport SERC-TR-41-P*, pages 14–27, 1989.

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

[72] W. Shen, Y. Li, Y. Han, L. Chen, D. Wu, Y. Zhou, and B. Xu. Boundary sampling to boost mutation testing for deep learning models. *Information and Software Technology*, 130:106413, 2021.

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

[74] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Proc. ICLR*, 2014.

[75] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[76] F. Thung, S. Wang, D. Lo, and L. Jiang. An empirical study of bugs in machine learning systems. In *Proc. ISSRE*, pages 271–280, 2012.

[77] 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.

[78] G. Vigna, W. K. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proc. CCS*, pages 21–30, 2004.

[79] S. Wang, T. Liu, J. Nam, and L. Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[80] 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.

[81] W. Xu, J. Offutt, and J. Luo. Testing web services by XML perturbation. In *Proc. ISSRE*, pages 257–266, 2005.

[82] S. Zagoruyko and N. Komodakis. Wide residual networks. In *British Machine Vision Conference*, 2016.

[83] W. E. Zhang, Q. Z. Sheng, A. A. F. Alhazmi, and C. Li. Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Trans. Intell. Syst. Technol.*, 11(3):24:1–24:41, 2020.

[84] 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.

[85] C. Ziegler. A google self-driving car caused a crash for the first time. *The Verge*, 2016.