**AUDEE: Automated Testing for Deep Learning Frameworks**

**Qianyu Guo**
College of Intelligence and Computing, Tianjin University, China

**Xiaofei Xie**
SCSE, Nanyang Technological University, Singapore

**Yang Liu**
SCSE, Nanyang Technological University, Singapore

**Yi Li**
SCSE, Nanyang Technological University, Singapore

**Xiaohong Li**
College of Intelligence and Computing, Tianjin University, China

**Chao Shen**
School of Cyber Science and Engineering, Xi’an Jiaotong University, China

**ABSTRACT**

Deep learning (DL) has been applied widely, and the quality of DL system becomes crucial, especially for safety-critical applications. Existing work mainly focuses on the quality analysis of DL models, but lacks attention to the underlying frameworks on which all DL models depend. In this work, we propose AUDEE, a novel approach for testing DL frameworks and localizing bugs. AUDEE adopts a search-based approach and implements three different mutation strategies to generate diverse test cases by exploring combinations of model structures, parameters, weights and inputs. AUDEE is able to detect three types of bugs: logical bugs, crashes and Not-a-Number (NaN) errors. In particular, for logical bugs, AUDEE adopts a cross-reference check to detect behavioural inconsistencies across multiple frameworks (e.g., TensorFlow and PyTorch), which may indicate potential bugs in their implementations. For NaN errors, AUDEE adopts a heuristic-based approach to generate DNNs that tend to output outliers (i.e., too large or small values), and these values are likely to produce NaN. Furthermore, AUDEE leverages a causal-testing based technique to localize layers as well as parameters that cause inconsistencies or bugs. To evaluate the effectiveness of our approach, we applied AUDEE on testing four DL frameworks, i.e., TensorFlow, PyTorch, CNTK, and Theano. We generate a large number of DNNs which cover 25 widely-used APIs in the four frameworks. The results demonstrate that AUDEE is effective in detecting inconsistencies, crashes and NaN errors. In total, 26 unique unknown bugs were discovered, and 7 of them have already been confirmed or fixed by the developers.

**CCS CONCEPTS**

- Software and its engineering → Software testing and debugging.

**KEYWORDS**

Deep learning frameworks, Deep learning testing, Bug detection

ACM Reference Format:

1 INTRODUCTION

In recent years, deep learning (DL) has achieved tremendous success in many domains, such as face recognition [41], speech recognition [16] and natural language processing [54]. However, it has been demonstrated that DL system is vulnerable and may cause serious consequences, e.g., the Tesla/Uber accidents [1, 5]. The quality assurance of DL system is becoming increasingly important, especially when it is applied in safety- and security-critical applications such as autonomous driving [8] and healthcare [26].

However, the quality assurance of DL systems is very complex due to its unique programming paradigm. The DL system usually involves three levels [34]: the application (e.g., DNN design), the DL framework (e.g., basic DL functionality support) and the hardware support (e.g., CUDA, CPU, and GPU). Developers first collect training data and design deep neural networks (DNNs) with the Application Programming Interfaces (APIs) of DL frameworks (e.g., TensorFlow [6] and PyTorch [32]). Then the DNNs are trained and inferred on the frameworks, with the support of underlying CUDA, GPU or CPU. Thus, the quality and robustness of a DL system is dependent on the quality of the three levels.

Recently, there is a lot of research focusing on the quality assurance of DL systems. However, most of them work on the application...
level, including the evaluation of DNN robustness [7, 15, 33, 50, 55] and the robustness enhancement [20, 31, 51]. Except for the quality assurance, some progress has been made on studying the bugs in DL systems, such as the application programming bugs [58], the DL framework bugs [34] and the deployment issues [17]. Nevertheless, there still lacks the techniques on automatically testing the DL frameworks and hardware (e.g., CUDA). In this paper, we mainly focus on the problem of DL framework testing.

There are several challenges in DL framework testing. 1) The test cases are different from those used by traditional software that usually takes concrete values of the inputs or files (e.g., AFL [28], EvoSuite [14]) as inputs. For DL framework testing, the test case is a DNN together with its inputs. The state-of-the-art approach CRADLE [34] selects some popular DNNs and train/test dataset to test DL frameworks. However, these DNNs and dataset may not be sufficient to cover all framework behaviours and may miss some bugs. 2) It lacks oracles for identifying logical bugs as we do not know what the ground truth is. Such logical bugs can be caused by various reasons, such as the real framework bugs or different implementation logics. Differential testing provides a promising solution to detect logical bugs by cross-checking inconsistent behaviours among different DL frameworks [34]. Despite some logical bugs have been found, it is non-trivial to understand their root causes.

After the inconsistencies are detected on a DNN, a source localization process is needed to identify where the inconsistency occurs for further root cause analysis and bug confirmation. However, there are two challenges in this stage: 1) Since a DNN may contain multiple layers and each layer may contain multiple parameters, it is challenging to identify which layers and parameters cause the inconsistency. 2) As the layers are connected, the inconsistent outputs caused by one layer will affect the subsequent layers, which may have no bugs. CRADLE [34] proposes a metric to measure the inconsistency degree for buggy layer localization. However, as aforementioned, the inconsistency degree may be larger or smaller due to the impacts of previous layers. Thus, it is non-trivial to set one threshold that is applicable for all layers. If the threshold is too large, some buggy layers with smaller inconsistency degree may be missed. If the threshold is too small, there will be many false positive cases, which increase the complexity of bug analysis.

To address these challenges, in this paper, we propose Audee, a novel approach for detecting and localizing bugs in DL frameworks. Specifically, Audee adopts a search-based testing method to generate test cases. To test more functionalities in DL frameworks, we first prepare diverse seed DNNs consisting of multiple hidden works. Specifically, Audee adopts a search-based testing method to precisely narrow the localization scope to certain layers and the parameters, by which the bugs and inconsistencies are caused. Thus a further manual debugging analysis can be conducted more efficiently to investigate and confirm the root causes.

To evaluate the effectiveness of Audee, we select 7 diverse seed DNNs on 3 popular dataset (MNIST, CIFAR-10, and IMDB) and apply Audee in testing 4 DL frameworks, i.e., TensorFlow, PyTorch, CNTK, and Theano. The results show that Audee is effective in detecting inconsistencies and bugs. Specifically, Audee detects 151 unique inconsistencies among the frameworks. Compared to CRADLE [34], Audee can further perform a more fine-grained localization, which identifies not only buggy layers but also buggy parameters. In addition, Audee is more effective and efficient in detecting NaNs than the state-of-the-art tool TensorFuzz [30]. On average, Audee detects NaNs in 77.0% DNNs while 52.67% in TensorFuzz. To further understand the root causes of the inconsistencies and bugs, with an over 224 person-hour effort, we conduct a manual analysis on 5 inconsistencies, 8 NaNs and 13 crashes.

To summarize, this paper makes the following contributions:

- We propose an automated testing approach to identify logical bugs, NaN errors and crashes of DL frameworks based on diverse mutation strategies.
- We propose a fine-grained localization method to identify the buggy layer as well as buggy parameters, by which the inconsistencies or bugs are caused.
- We conduct an extensive evaluation to demonstrate the effectiveness of our techniques on four DL frameworks, where 26 unknown bugs are identified, and 7 of them have been confirmed or fixed by the developers.
- We conduct an empirical study on 151 unique inconsistencies and 26 real bugs to summarize their root causes, which can provide useful guidance for the evolution of DL frameworks.

2 APPROACH

In this section, we first define the problems in DL framework testing and then detail the bug detection and localization algorithms.

2.1 Problem Definition

Definition 1 (Deep Neural Network). A Deep Neural Network (DNN) $f$ consists of a sequence of layers $(L_0, L_1, \ldots, L_m)$, where $L_0$ is the input layer, $L_m$ is the output layer, and $L_1, \ldots, L_{m-1}$ are the hidden layers. Each layer is configured with some layer parameters $L^p$ and assigned with neural weights $L^\theta$. Given a d-dimensional input vector $x \in \mathbb{R}^d$, $f$ calculates the outputs layer-by-layer:

$$x_{L_i} = \begin{cases} \phi_{L^0}(x), & i = 0 \\ \phi_{L^i}(x_{L_{i-1}}), & i > 0 \end{cases},$$

where $x_{L_{i-1}}$ is the output of the previous layer $L_{i-1}$, $\phi$ is the underlying DL framework, and $\phi_{L^i}$ is the layer algorithm which depends on the configured parameters and learned weights. Given an input $x$, the output of $f$ with framework $\phi$ can be represented as $f(x) = (x_{L_0}, \ldots, x_{L_m})$. Particularly, if $f$ is a m-classification task, the label
output on $\phi$ can be calculated as $f^\phi\left(x\right) = \arg\max_{i \in \{0, \ldots, m-1\}} \left(x_{Lm}[i]\right)$, where $x_{Lm}$ is a $m$-dimensional probability vector.

The inconsistency between two frameworks is defined as follows:

**Definition 2 (Inconsistency).** Given a classifier $f$, an input $x$ and two different DL framework implementations (i.e., $\phi_1$ and $\phi_2$), we consider an inconsistency occurs between $\phi_1$ and $\phi_2$ if $f^\phi_1\left(x\right) \neq f^\phi_2\left(x\right)$.

The inconsistencies are usually caused by the different code logic between $\phi_1$ and $\phi_2$. The root causes of such differences can be: 1) at least one of the two frameworks is buggy; or 2) neither has a bug, but the algorithms are implemented quite differently.

We define a test case for DL framework testing as follows:

**Definition 3 (Test Case).** A test case for testing DL frameworks is a tuple $(f, x)$, where $f$ stands for a DNN consisting of multiple layers and $x$ represents an input to $f$.

In DL framework testing, we aim to detect program crashes, NaN errors, and result inconsistencies, by generating diverse test cases to cover more behaviours of DL frameworks. As a DNN may contain multiple layers and each layer may contain multiple parameters, we need to localize a minimal root cause, so that further debugging can be conducted more effectively. The process of source localization is to identify both buggy layers and parameters, by which the bugs or inconsistencies are generated.

### 2.2 Overview

Figure 1 shows the overview of Audee which includes three major steps: the testing routines for DL frameworks, the source localization, and the empirical study on inconsistencies and bugs.

As defined earlier, each test case consists of a DNN $f$ and an input $x$ (Definition 3). The DNN $f$ contains multiple layers, where each layer $L$ contains a number of parameters $L^p$ and weights $L^\theta$. Therefore, the diversity of test cases could be measured in four aspects: 1) layer diversity in $f$, 2) parameter diversity in a layer, 3) input diversity (i.e., different $x$), and 4) weight diversity (i.e., different $L^\theta$). Audee is designed to generate test cases aiming at achieving diversity from above four aspects. Specifically, considering Figure 1a, we first summarize the widely used APIs (e.g., convolution and pooling) and their configurable parameters (Section 2.3). Then, a set of seed DNNs which cover these APIs are selected (for (1)). Next, Audee randomly generates multiple DNNs by mutating the values of parameters in each layer based on the API configurations (for (2)). With a set of seed inputs, the generated DNNs are used to test the DL frameworks (i.e., TensorFlow, PyTorch, CNTK, and Theano) by passing the inputs to these DNNs.

Apart from test cases, test oracles are also needed in DL framework testing. In this paper, we mainly focus on three types of issues: crashes, NaN errors (e.g., square root of negative values), and logical bugs. The first two types can be caught easily with corresponding instruments. For logical bugs, we aim to identify inconsistencies by leveraging multiple DL frameworks as cross-checking oracles. With the help of model converters (e.g., MMdnn [27], ONNX [13], and Keras [4]), the same DNN can be run on different frameworks. If Audee cannot identify bugs or inconsistencies with the generated DNNs, then the fine-grained mutations are performed to generate more diverse test cases, i.e., mutating the seed inputs, the DNN weights, or both of them (for (3) and (4)). Audee repeats this process until an inconsistency is triggered or it exceeds a given time budget.

After the inconsistencies or bugs are detected in $f$, we adopt a layer-based causal testing to localize the buggy layers as well as buggy parameters (Figure 1b). Audee identifies the most inconsistent-prone layer. Then it minimally modifies the value of each parameter one by one and observe their impacts to the inconsistency meanwhile. Intuitively, if the inconsistent behaviour disappears after we change the value of certain parameter, then the parameter is likely to be the root cause. Finally, to further understand the root causes of bugs and inconsistencies, we conduct an empirical study on the bugs and inconsistencies discovered by Audee (Figure 1c).

### 2.3 API Configuration and DNN Generation

As aforementioned, the DNN is a stack of multiple layers, and each layer is configured with multiple parameters to realize its DL-specific functionality. In the framework implementation, a layer is generally represented by a certain API. Namely, testing a layer can be equally considered as testing the corresponding API. Overall, we manually investigate 25 commonly used APIs of four frameworks (i.e., TensorFlow, PyTorch, CNTK, and Theano) and summarize the configurable parameter values for each API, which can be seen on our website [3]. Then we randomly generate a set of DNN variants by combining different parameter configurations. Particularly, to better evaluate these APIs, we also introduce some crafty invalid values for certain parameters. This can expand our testing space to go through more code paths in the framework implementation. Examples of such invalid values are the $\text{None}$ for string-type parameters, and the negative numbers for some numerical parameters. Figure 2 shows an example of parameter configuration for the API Conv2D. The left side is a simplified configuration file of Conv2D including six parameters, where $\text{int}$ and $\text{str}$ indicate the value types of the parameter. For the value type $\text{int}$, “1” and “2” represent the required dimensions of the values. For the value type $\text{str}$, we list all of possible values (e.g., “relu” and “linear”) for the optional parameter. Based on the left configuration file, Audee generates a series of parameter sets for Conv2D by randomly combining all values of all involved parameters. The right side shows some generated parameter sets, including one valid example and one invalid example (marked by red dash box), respectively. Obviously, the kernel size requires to be a positive number, and $\text{kernel\_size} = 0$ is an invalid parameter, which finally raises a crash under TensorFlow [48].

### 2.4 Search-based DL Framework Testing

By random parameter mutation (Section 2.3), we can perform the testing with initial seed inputs and fixed weights. To cover more framework behaviours, we need more representative inputs and weights. However, it is impossible to traverse all cases, due to the high dimensions of the inputs and weights. Then, the challenge is how to generate optimal inputs and weights that are more likely to trigger the abnormal framework behaviours. Audee adopts a Genetic Algorithm (GA) [52] based testing, a popular heuristic technique, to find optimal solution through multiple evolutions.
Algorithm 1: DL Framework Bug Detection

Input: \( f \): A DNN, \( \theta \): Weights of the DNN
\( x \): An input to the DNN
\( \Phi \): The targeted DNN frameworks
\( r_1 \): Crossover rate, \( r_2 \): Mutation rate
\( \text{max} \text{fit} \): Maximum iterations, \( n \): Size of population

Output: \( F \): A set of failed test cases

Const: \( m \): The number of selected parents

1. \( \text{iter} := 0; \)
2. \( F := \emptyset; \)
3. \( P := \text{initPopulation}(x, \theta, n); \)
4. while \( \text{iter} < \text{max} \text{fit} \) do
5. \( \text{iter} := \text{iter} + 1; \)
6. \( \text{Fit} := \text{computeFitness}(P, \Phi); \)
7. \( P' := \text{select}(P, m, \text{Fit}); \)
8. \( P'' := P'; \)
9. while \( \text{size}\left( P'' \right) < n \) do
10. \( x_1, x_2 := \text{selectParents}(P''); \)
11. \( x' := \text{crossover}(x_1, x_2, r_1); \)
12. \( r := U(0,1); \)
13. if \( r < r_2 \) then
14. \( x'' := \text{mutate}(x'); \)
15. \( P'' := P'' \cup \{x''\}; \)
16. \( X := \text{checkFailed}(P''); \)
17. if \( X \neq \emptyset \) then
18. \( F := F \cup X; \)
19. return \( F; \)

There are failed cases in current population (Line 16). If yes, they are added into the failed pool \( F \) (Lines 17 to 18). Note that, the failed cases refer to various anomalous behaviours on DL frameworks, categorized by our testing oracles in Section 2.4.1.

2.4.1 Oracles. As mentioned earlier, we focus on three different types of errors as the testing oracles in this paper:

1. Crashes. We monitor the processes of DNN loading and inference to check whether the DL framework exits abnormally.
2. NaN errors. We check whether there are some not-a-number values in outputs of each layer.
3. Logical bugs. For the inconsistencies that do not raise crashes or NaNs, we conduct a cross-checking between DL frameworks, which should have the same functionality. Intuitively, given the
we construct a population of chromosomes, which define genes that
inconsistency degree through evolutions, until it is significant enough
to inconsistent inference results (see Figure 5a for example).

2.4.2 Generation of Population. With the initial inputs and weights, we construct a population of chromosomes, which define genes that GA is trying to optimize. We design three levels of chromosomes:

(1) Input Level. The chromosome is an input of the DNN such as image, audio or embedded values of natural language. The original input is flattened as a vector \((x_0, \ldots, x_n)\), where \(x_i\) is a gene in the chromosome.

(2) Weight Level. The chromosome is the weights of the DNN.

The weights of all layers are concatenated into one vector \((\theta_0, \ldots, \theta_m)\), where \(\theta_i\) is a floating point number serving as the gene of the chromosome.

(3) Input & Weight Level. The chromosome is the concatenation of input and the weights represented as: \((x_0, \ldots, x_n; \theta_0, \ldots, \theta_m)\).

Given an input \(x\) and the weights \(\theta\), we construct a set of populations by randomly adding Cauchy noises [53]. For input level population, the initial weights \(\theta\) remain unchanged during the evolutions. We only mutate the inputs so that bugs can be triggered on the DNN \(f\) with new inputs. Similarly, for weight level population, the inputs keep unchanged. The weights of the DNN are evolved such that bugs can be triggered on the new weights. For input and weight level, we mutate both of the inputs and weights simultaneously. The optimization space of input-and-weight level is much wider than that applying only input level or only weight level.

2.4.3 Inconsistency Fitness Function. An inconsistency is usually a cumulative effect produced by the calculation difference of hidden layers within the DNN [17], which is finally represented as a classification difference in the inference results (Definition 2). We call the impact of each hidden layer on inconsistency as the inconsistency degree of this layer. To measure the inconsistency degree between two DL frameworks, we define the layer distance as follows:

**Definition 4 (Layer Distance).** Given a DNN \(f = (L_0, \ldots, L_n)\) with an input \(x\), the outputs of \(f\) on two DL frameworks \(\phi\) and \(\phi'\) are \(f_\phi(x) = \{x_\phi^1, \ldots, x_\phi^n\}\) and \(f_{\phi'}(x) = \{x_{\phi'}^1, \ldots, x_{\phi'}^n\}\), respectively. We define the output distance of layer \(L_i\) between two frameworks as:

\[
\delta_{\phi_i, \phi'_i} = \frac{1}{m} \sum_{j=1}^{m} \left| x_\phi^j - x_{\phi'}^j \right|
\]

where \(m\) is the dimension of the output of \(L_i\).

The layer distance characterizes the output difference between two frameworks on this layer. Intuitively, a large layer distance will inevitably affect subsequent calculations and may ultimately leads to inconsistent inference results (see Figure 5a for example). AUDDE adopts the layer distance as fitness so that GA could amplify the inconsistency degree through evolutions, until it is significant enough to trigger an inconsistency. A question is which layers should be selected to compute the distance. The most straightforward solution is using the average distance of all hidden layers. However, it is too computationally expensive in practice, especially when a DNN contains too many layers. Inspired by the state-of-the-art adversarial attack techniques [7, 15], we select the logits layer as the target layer, because it is the output layer of the DNN, which can reflect the overall inconsistency degree of all previous layers.

\[
2.4.4 NaN Fitness Function.\] Not-a-Number (NaN) is a numerical data type used to represent any value that is undefined or unrepresentable. NaN could raise dangerous behaviours in important systems [30]. Since there exist massive high-precision computing operations in DL tasks, it is more likely to generate NaN outputs.

NaN errors are commonly reported in some cases including invalid operations (e.g., the square root of a negative number) and unsafe operations that take overflow/underflow values as inputs [59]. Through tracing the NaN cases by layer, we observe that NaN is usually caused by the outlier values, especially when some too-large or too-small vector elements are involved in calculation. For example, softmax is a common operation in DL tasks to generate the final inference probabilities. In TensorFlow, it is implemented as \(\text{softmax}(x) = \frac{\text{exp}(x)}{\text{sum}(\text{exp}(x))}\), where \(x\) is a numerical vector. If there is an element that is too large in \(x\), \(\text{exp}\) may output an inf value, which can further lead to NaN when the inf involves in some arithmetics [43]. Another example is the square root operation which is also common in the implementation of the DL framework. It generates a NaN output if the input contains negative elements (see the NaN bug [47]). Both cases can be attributed to the unbalanced numerical distribution when the inputs or weights participate in calculation.

Based on above observations, we propose a heuristic-based fitness function for NaN detection. Given a DNN \(f\) with an input \(x\) and a target DL framework \(\phi\), the NaN fitness is calculated as:

\[
\max(x^\phi_L) - \min(x^\phi_L),
\]

where \(L_i\) is one layer in \(f\), \(x^\phi_L\) is an \(n\)-dimensional vector. This fitness is designed to characterize the unbalanced data distribution on certain layer. Then GA is applied to amplify the unbalanced distribution within the hidden layers until a NaN is triggered. Intuitively, the generated unevenly distributed output vector, with very large or very small elements included, are more likely to produce NaNs. Similarly, we also select the logits layer to calculate the NaN fitness. Note that, unlike inconsistency detection, it does not rely on the multi-framework cross-checking to calculate the NaN fitness.

2.5 Source Localization

After an inconsistency is detected, we need to localize the source (i.e., caused by which layers and which parameters) so that the bug analysis could be easily performed. Here, we propose a causal-testing based approach using the layer change rate as metric, which is to measure the output change of certain layer compared to the adjacent previous layers [34]. Specifically, given an input \(x\) on two frameworks \(\phi\) and \(\phi'\), the change rate of layer \(L_i\) can be calculated as follows:

\[
R_{L_i}^{\phi, \phi'}(x) = \frac{\delta x_{\phi_i, \phi'_i}}{\delta x_{\phi_i} + \epsilon},
\]

where \(\delta x_{\phi_i} = \max_{x\in\text{pre}(L_i)}(\delta x_{\phi_i, \phi'_i})\) and \(\text{pre}(L)\) stands for the previous layers whose outputs are the inputs of layer \(L\). Notably, the \(\epsilon = 10^{-7}\) is set to avoid the division-by-zero.

CRADLE [34] applies this metric to highlight the buggy layers, whose change rates are higher than a preset threshold. However, it may be coarse as a correct layer can also have a high change
rate caused by the previous buggy layers. Audee performs a fine-grained “localize-fix” strategy to troubleshoot the inconsistency layer by layer. The basic idea is as follows: we start from the top layer to identify the first layer that has an outlier change rate. Then we “fix” this inconsistent behaviour of this layer by changing the values of the problematic parameters such that this layer will not affect the results of the following layers. Next we will continue to identify another buggy layer until no one could be found. By this way, we can focus on the influence of current layer for the final result, regardless of the impacts from the previous layers.

Algorithm 2 details the source localization process, with the help of an inconsistency threshold \(t_1\), and a consistency threshold \(t_2\) (\(t_1 \gg t_2\)). The inputs contain a DNN \(f\) and an input \(x\), which causes the inconsistency on two frameworks \(\phi\) and \(\phi'\). In each iteration, we identify the first layer \(L\) whose change rate is greater than \(t_1\) (Lines 3 to 4). \(a_L\) represents the parameters of layer \(L\) (Line 5). For each parameter, we replace the value with other values guided by the summarized API configuration (Line 8), and construct a dummy DNN \(f'\) that only contains the new layer \(L'\) (Line 9). Note that, the parameter replacement may lead to other misbehaviours (i.e., crash or NaN), which are easier to localize. For crash, we can simply retrieve the exception logs. For NaN error, it is obvious to capture. The replaced parameter value is the source of the crash/NaN (Lines 10 to 12). Afterwards, we check whether the previous large distance persists by feeding the same layer input. Algorithm 1 is used to amplify the layer distance between frameworks (Line 14) and \(x_{max}\) is the input that generates the maximum layer distance. Intuitively, if we cannot amplify the layer distance on \(L'\) (i.e., the change rate is always less than \(t_2\)), it means the inconsistent behaviour disappears after the value change. Thus this parameter value is one source of the inconsistency (Line 16). After the inconsistency-triggering parameters \(P\) are identified (Line 17), we “fix” the DNN by replacing the values of these buggy parameters, so that there is no anomalies on the identified layer (Line 18). The new DNN ensures that this layer has no impacts on the change rate of subsequent layers. Then we continue to localize the following layers with the new DNN.

We set conservative values for the thresholds \(t_1\) and \(t_2\) based on our manual study results (see Section 3.3). If the change rate is below \(t_2\), the inconsistency is likely to be a precision issue. If the change rate is higher than \(t_1\), then it is more likely to be caused by a bug or the implementation difference. Note that, Audee simply narrows the localization scope to which layers and which parameters raise such an inconsistency. It still needs further manual source code debugging to judge whether the identified inconsistency is a real bug or just due to the different framework implementations.

### 3 EVALUATION AND RESULTS

We implement Audee using Python on top of Keras [4] and PyTorch [32]. To evaluate the effectiveness of Audee and understand the root causes of inconsistencies and bugs, we design substantial experiments\(^1\) aiming at answering the following research questions:

- **RQ1**: How effective is Audee in detecting inconsistencies?
- **RQ2**: How useful is Audee in localizing layers as well as parameters for the inconsistencies?
- **RQ3**: How effective is Audee in detecting NaNs?

\(^1\)More experimental details can be found on our website [3].

#### 3.1 Dataset and Experimental Setups

To evaluate Audee, we choose the latest stable versions of four widely used DL frameworks as targets (i.e., TensorFlow [6] 2.1.0, PyTorch [32] 1.4.0, CNTK [37] 2.7, and Theano [42] 1.0.4). To capture inconsistencies, the same DNN needs to be run on different frameworks. Thus we conduct the inconsistency detection between four framework pairs, i.e., TensorFlow vs. CNTK, TensorFlow vs. Theano, Tensorflow vs. PyTorch, and CNTK vs. Theano. We use Keras [4] as a high-level wrapper so that a DNN can be seamlessly inferred on TensorFlow, CNTK and Theano. For TensorFlow and PyTorch, we use the state-of-the-art model converter MMDnn [27] to convert the DNNs from TensorFlow to PyTorch. In other words, Keras and MMDnn serve as bridges, ensuring the DNN can be equivalently compared across different frameworks, with the same runtime configurations (e.g., layer parameters and weights).

Overall, seven DNNs are selected in Audee, including four popular CNN models and three RNN models. These models cover 25 popular APIs in total. Three publicly available datasets are selected as the inputs to these models. Specifically, for CNNs, LeNet-5 [23] is inferred on MNIST [24] for hand-written digit recognition; ResNet-20 [18], VGG-16 [39], and MobileNet-V2 [36] are used to detect objects on CIFAR-10 [22]. In addition, we also construct three models with RNN layers involved (i.e., SimpleRNN, LSTM, and GRU). These models are tested on IMDb [2], a dataset of movie reviews that is widely used in the sentiment analysis.

---

**Algorithm 2: Inconsistency Source Localization**

<table>
<thead>
<tr>
<th>Input: (f: \langle L_0, \ldots, L_n \rangle): A DNN (x): An input (\phi, \phi':): Two DL frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: (X, Y): two sets of layers as well as parameters</td>
</tr>
<tr>
<td>Const: (t_1): A larger threshold, (t_2): A smaller threshold</td>
</tr>
<tr>
<td>(X, Y := \emptyset)</td>
</tr>
<tr>
<td>(\beta := {R^\phi(x)</td>
</tr>
<tr>
<td>(L := \beta[0])</td>
</tr>
<tr>
<td>Let (a_L := {a_0, \ldots, a_m}) be the parameters of (L)</td>
</tr>
<tr>
<td>(P := \emptyset)</td>
</tr>
<tr>
<td>for (a \in a_L) do</td>
</tr>
<tr>
<td>(L', a' := replace(L, a))</td>
</tr>
<tr>
<td>(f' := \langle L' \rangle)</td>
</tr>
<tr>
<td>(y := checkCrash_NaN(f');)</td>
</tr>
<tr>
<td>if (y \neq \emptyset) then</td>
</tr>
<tr>
<td>(Y := Y \cup {L', a'})</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>(x_{max} := detectInconsistency(f');) Algorithm 1</td>
</tr>
<tr>
<td>if (R^\phi_{a_L}(x_{max}) &lt; t_2) then</td>
</tr>
<tr>
<td>(P := P \cup {a})</td>
</tr>
<tr>
<td>(X := X \cup {L, P})</td>
</tr>
<tr>
<td>(f := fixDNN(f, X, L))</td>
</tr>
<tr>
<td>until (\beta = \emptyset)</td>
</tr>
<tr>
<td>return (X, Y);</td>
</tr>
</tbody>
</table>
3.2 RQ1: Inconsistency Detection

For each seed DNN, we randomly generate 500 different DNNs by changing the values of layer parameters (i.e., network mutation). Then we select 10,000 inputs from the test dataset to check whether some inconsistencies are triggered in the framework pairs. Then the fine-grained input mutation and weight mutation are applied on those non-inconsistency DNNs for further inconsistency detection. Specifically, we randomly select 20 non-inconsistency generated DNNs, each of which is repeatedly tested until an inconsistency is discovered or it reaches a 30-minute timeout. Note that, multiple inconsistencies could be found on a DNN, with different inputs or weights. We conduct an inconsistency reduction based on the following criteria: given a DNN, the inconsistencies caused by different inputs are regarded as the same one, while those caused by different weights are regarded as different ones. This is because the change of weights will modify the decision logic of the DNN, thus we consider the model with mutated weights as a new DNN, which is different from the original DNN.

Figure 3 shows how the input mutation and weight mutation amplify the layer distances within LeNet-5 and thus trigger inconsistencies between TensorFlow and Theano. Given an input image whose label is 2 and the DNN (i.e., LeNet-5), both frameworks infer the image as the label "2", with the initial layer distance being 0.01, marked as the blue arrow. By applying input mutation, AUDEE then generates another new input by adding some noise. Feeding the mutated image to the DNN (see the orange arrow), the layer distance is amplified to 0.89, and triggers an inconsistency (i.e., 2 and 4). By applying weight mutation, AUDEE also generates a new DNN (marked by red), which we call the variant DNN. Finally, as shown by the grey arrow, the inconsistency remains when inferring the mutated image on the variant DNN, with the layer distance further amplified to 1.95.

Table 1 shows the number of inconsistencies detected by AUDEE with different mutation strategies. Columns TF-CN, TF-TH, CN-TH, and TF-PTH represent the framework pairs, where TF, CN, TH, and PTH are TensorFlow, CNTK, Theano, and PyTorch, respectively. Columns NET, IN, WT, and IN-WT show the number of generated DNNs on which the inconsistencies are triggered by network mutation, input-level mutation, weight-level mutation, and input-and-weight mutation, respectively. Column Tot summarizes the total number of inconsistencies of all strategies. Notably, "-" means that MMdnn fails to convert the DNN from TensorFlow to PyTorch.

Answer to RQ1: AUDEE can detect inconsistencies effectively. By random network mutation, AUDEE can detect inconsistencies for some DNNs quickly. Furthermore, the input-level mutation and the weight-level mutation can complement network mutation very well in detecting more inconsistencies. A joint mutation of input and weight shows the best detection performance.
Table 1: The results of inconsistencies generated by different mutation strategies

<table>
<thead>
<tr>
<th>DNNs</th>
<th>TF-CN</th>
<th>TF-TH</th>
<th>CN-TH</th>
<th>TF-PTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tot</td>
<td>NET</td>
<td>IN</td>
<td>WT</td>
</tr>
<tr>
<td>LeNet-5</td>
<td>42</td>
<td>28</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>ResNet-20</td>
<td>252</td>
<td>84</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>VGG-16</td>
<td>25</td>
<td>10</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>MobileNet-V2</td>
<td>34</td>
<td>8</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>SimpleRNN</td>
<td>17</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>LSTM</td>
<td>303</td>
<td>29</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>GRU</td>
<td>53</td>
<td>21</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>726</td>
<td>182</td>
<td>35</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 2: The results of source localization

<table>
<thead>
<tr>
<th>DNNs</th>
<th>#M</th>
<th>#Layer</th>
<th>L_Audee</th>
<th>CRADLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#L(75%)</td>
</tr>
<tr>
<td>LeNet-5</td>
<td>79</td>
<td>9</td>
<td>1.15</td>
<td>2</td>
</tr>
<tr>
<td>VGG-16</td>
<td>22</td>
<td>56</td>
<td>1.5</td>
<td>14</td>
</tr>
<tr>
<td>ResNet-20</td>
<td>84</td>
<td>71</td>
<td>2.18</td>
<td>18</td>
</tr>
<tr>
<td>MobileNet-V2</td>
<td>19</td>
<td>158</td>
<td>3.2</td>
<td>40</td>
</tr>
<tr>
<td>SimpleRNN</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LSTM</td>
<td>29</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GRU</td>
<td>25</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3.3 RQ2: Effectiveness of Localization

To select the thresholds (i.e., $t_1$ and $t_2$) for Algorithm 2, we first conduct a manual study to understand the distribution of change rates of DNNs in terms of two categories, i.e., the inconsistency and consistency, respectively. We focus on the top-1 change rate (Top1-CR) of all layers. Given a DNN $f = \langle L_0, \ldots, L_n \rangle$, its top-1 change rate between two DL frameworks $\phi$ and $\phi'$ on the input $x$ is $\max\{R^\phi_{L_n}(x), \ldots, R^\phi_{L_0}(x)\}$. Specifically, for each of the seven DNN types, we randomly select 150-200 test cases (i.e., DNN and input) for the two categories, respectively. In total, we study 1145 inconsistent cases and 1332 consistent cases to observe the distribution of the top-1 change rate.

Figure 4 shows the box plots of Top1-CR for the two categories. Due to the space limit, we only present the results of three DNNs and others can be found in our website [3]. Overall, we find that the DNNs with inconsistencies have much larger Top1-CR, while the Top1-CR on consistent DNNs tends to be very small. Moreover, we also find the Top1-CR depends on the DNN structure. For example, the inconsistency Top1-CR on LSTM is less than that on other two DNNs. Based on this observation, we select two conservative thresholds so that the buggy layers are more likely to be identified. For the inconsistency threshold $t_1$, a smaller value 1000 is selected. In other words, if the layer distance is greater than 1000, it will be regarded as the inconsistency candidate. For the consistency threshold $t_2$, a very small value 2 is selected. Thus, one layer tends to be bug-free only if the change rate is less than 2.

Table 2 shows the results of source localization. Column #M presents the total number of DNNs which have inconsistencies.

Note that, for each DNN, the inconsistency may be caused by multiple inputs and we randomly select one. Column #Layer shows the total number of layers in the DNN. The RNN models contain five layers, of which one layer is the RNN layer. Column L_Audee shows the average number of layer and parameter pairs, which are localized by Audee. We have manually checked all results and found that all of them were true inconsistencies, i.e., caused by bugs or implementation differences. More details on the results of manual analysis will be introduced in Section 3.5.

We also list the results by using the strategy in CRADLE for comparison. CRADLE selects the third quantile of the change rates from all layers as the threshold (Column #L(75%)), leaving the remaining quarter always be identified as buggy layers. As a result, CRADLE will inevitably report some false positive layers (Column FP(75%)) which increases the complexity of inconsistency analysis. Additionally, except for buggy layers, Audee also localizes the buggy parameters which is very useful in the further debugging analysis. Notably, we emphasize it is not absolutely fair to compare Audee with CRADLE, because the threshold in CRADLE could be adjusted. Our purpose is to show that Audee can provide fine-grained localization results, including not only bug-candidate layers like CRADLE, but further bug-prone parameters as well.

Figure 5 shows an example that Audee localizes the source of inconsistency within ResNet-20 between TensorFlow and CNTK. As shown in the red box, there exist some Conv2D layers that exhibit very large change rate (i.e., $2.06e+6 \gg t_1$). Intuitively, such a huge difference will inevitably affect the subsequent calculations and may ultimately leads to inconsistent results on two frameworks. We filter these outlier layers and apply a causal-testing based method to minimally change the values of each parameter (consider Algorithm 2). For example, for the parameter padding, after changing the value from "same" to "valid" while keeping other parameters unchanged (marked by red arrow), we find the change rate decreases dramatically to 3.95e-1. Similarly, after we change the parameter strides from 2 to 1, ensuring other parameters the same as before, the change rate also decreases abruptly to 3.46e-1. We consider the outlier layer distance disappears after the parameter changes, because the change rate is much smaller than the consistency threshold $t_2$ in both cases. Finally, for the inconsistent case of ResNet-20, Audee identifies two candidate sources, i.e., the (Conv2D, padding="same") and the (Conv2D, strides=2). Then we could further narrow the debugging scope to the implementations of these two parameters in Conv2D.
3.4 RQ3: NaN Detection

Similar to the inconsistency detection (RQ1), we first apply the network mutation to generate 500 variants for each seed DNN, checking whether any NaN is triggered on these DNNs. For those DNN variants that do not produce NaN, we then randomly select 20 of them to further mutate the inputs and weights for NaN detection. Following the results of RQ1 that a joint input-and-weight mutation outperforms mutating either alone, we only adopt this mutation strategy to detect NaNs in this experiment. For comparison, we select the state-of-the-art tool TensorFuzz [30], a coverage-guided testing technique as benchmark, which claims to be able to detect NaNs. Since TensorFuzz only supports TensorFlow, we merely compare them on this framework. Note that, this does not mean Audee cannot be applied to other DL frameworks for NaN detection. The testing terminates once a NaN is detected or it reaches a preset 30-minute timeout. To reduce the randomness, we test each DNN five times and calculate the average results.

Table 3 shows the average results of NaN detection. Column NET stands for the number of DNNs with NaN outputs, which are generated by the network mutation. Column #NaN shows the average number of NaNs detected by TensorFuzz and Audee on the 20 randomly selected DNNs. Column #Time(s) represents the average time used to capture the first NaN. Note that, neither of TensorFuzz and Audee can detect NaNs on some DNNs (e.g., the RNN models) and frameworks (e.g., PyTorch), thus we ignore them in Table 3. The results show that, from a total of 1500 generated DNNs, network mutation identifies 26 NaN-producing DNNs. These NaNs are triggered in multiple layers on all Keras backends (i.e., TensorFlow, CNTK, and Theano). We perform a deep investigation and find they are all caused by one parameter (i.e., the exponential activation function), that may output inf value given a large input. More details about this NaN error can be found on our website [3].

Apart from the network mutation, Audee can also effectively detect more NaNs with the input-and-weight mutation. On average, Audee detects NaNs in 75.67% DNNs on TensorFlow and 78.33% DNNs on Theano, respectively. By contrast, TensorFuzz only detects NaNs for 52.67% DNNs on TensorFlow. This indicates that Audee is more effective than TensorFuzz in detecting NaNs. In terms of the time cost, Audee also behaves more efficiently. For each DNN, the average time used to capture the first NaN in Audee is much less than that in TensorFuzz. In addition, the time cost depends on the size of the DNN. The larger DNN requires more inference time, which also leads to consuming more time to detect NaNs. For example, VGG-16 is much larger than other two DNNs and it takes much longer to trigger a NaN.

Answer to RQ3: Audee outperforms the state-of-the-art technique TensorFuzz in two folds: (1) Audee can detect NaNs more effectively and efficiently. (2) Audee supports more DL frameworks for NaN detection, not simply limited to TensorFlow like TensorFuzz.

3.5 RQ4: Empirical Study

Based on above results, we further conduct an empirical study to understand the root causes of inconsistencies and real bugs.

3.5.1 Study on Inconsistencies. In the study, we totally select 171 inconsistencies including: (1) a total of 151 unique inconsistencies (i.e., 151 layer and parameter pairs) from the localization results and (2) 20 randomly selected inconsistencies which are filtered by the
localization process. Then, we conduct a manual analysis on the detailed implementations in these frameworks. Finally, we summarize the root causes of inconsistencies into 3 categories:

1. **Implementation Bug (68/151).** This kind of inconsistencies are caused if there exist bugs in any of the frameworks. Considering the example in Figure 6a, CNTK has a bug in the implementation of the layer DepthwiseConv2D. Specifically, DepthwiseConv2D requires to calculate the convolution operation separately on each channel. However, CNTK only takes the first channel (marked by the red dash box) into calculation, due to a support limitation of the inner MKL library in case of the asymmetric padding. This channel-ignorance bug makes CNTK generate quite different outputs with TensorFlow and Theano, which triggers a significant inconsistency.

2. **Implementation Difference (83/151).** There are some inconsistent cases that, frameworks implement certain operation in different ways, but none of them can be regarded as bugs. Consider the example in Figure 6, the padding operation is implemented differently in TensorFlow and Theano. The blue grids represent the original input area, which is a 3-channel 4 × 4 image. The red grids stand for the extension area for zero-padding. The black grids represent a 2 × 2 convolutional kernel. TensorFlow adopts the asymmetric padding and prefers to arrange more padding area to the right/bottom side given an odd padding number, as shown in Figure 6b. By contrast, Theano strictly conducts symmetric padding that pads the same space on both sides, no matter whether the padding number is odd or even (see Figure 6c). As a result, the starting kernel covers different pixels in Theano compared to TensorFlow, and further leads to output inconsistency. Actually, the different padding implementation has widely affected many layers with padding options involved in the parameter list, such as the convolutional operations (e.g., Conv2D) and pooling operations (e.g., MaxPooling2D and AveragePooling2D).

3. **Precision (20).** We also find 20 inconsistencies filtered by Audee, that are caused by the floating-point precision. As there exist many high-precision computations in DL tasks, it is possible that some slight precision differences may affect the final inference results. We investigate the Top1-CRs of the 20 inconsistencies, and find they are all very small (less than 1.9). However, such tiny precision differences still cause a result change during inference. Audee can filter all of the inconsistencies caused by precision so that manual analysis can be conducted on the meaningful inconsistencies.

It should be noted that, although the implement differences are not identified as bugs, they may still lead to the quality issues when DNNs are migrated between frameworks. In other words, a well-trained DNN may exhibit low performance on other frameworks. The results call for attention for considering and handling the inconsistencies in the migration phase. For example, the current model converter (e.g., MMdnn [27], ONNX [13], Keras [4]) should consider such potential inconsistencies and report warnings for some layers.

3.5.2 **Study on Bugs.** We also take substantial effort to manually understand the root causes of all misbehaviours detected by Audee in the framework source code level, and finally get the bug confirmation. Table 4 shows the unique bugs found by Audee, whose symptoms cover crash, NaN and inconsistency. In total, we discover 26 unique bugs, including 8 TensorFlow bugs, 4 CNTK bugs, 4 Theano bugs, 1 PyTorch bug, and 9 Keras bugs. We have reported the bugs and 7 of them have been confirmed or fixed by the developers [9, 21, 35, 44–49]. More details about the 26 bugs can be found on our website [3]. We summarize the root causes of these bugs into two categories:

1. (Non-robust handling on corner cases. The framework developers may not fully consider the situations that may be encountered when the program is running, thus ignoring some key checks to ensure the code robustness. For example, Fig. 8 shows a typical crash found by Audee, which has already been fixed by TensorFlow [44, 48]. Various convolutional APIs such as DepthwiseConv2D, SeparableConv2D, and Conv2D on TensorFlow take “kernel_size=0” as normal parameter for calculation, and finally lead to crashes. As another example, both TensorFlow and Theano conduct unsafe sqrt operations in BatchNormalization without guaranteeing all inputs be positive, which could further trigger NaN outputs [47]. In this study, we find 16 out of 26 bugs suffer from non-robust implementations. In addition, such cases are not found in PyTorch, indicating PyTorch may be implemented more carefully.

2. Logical implementation errors. 10 out of 26 bugs are caused by incorrect code logic in DL frameworks. The incorrect convolution calculation shown in Figure 6a is a typical example. As another example, Fig. 7 shows a NaN bug occurred on PyTorch, which is caused by the logical error when handling the pooling operation. Specifically, PyTorch has a condition check (Line 4) to ensure the pool unit starts inside the image along a certain dimension. This check is only conducted when there is a padding option set (Line 1). However, it is still possible that the pool unit steps out of the image for the no-padding cases, which could further result in a division-by-zero operation, thus triggering NaN output afterwards. We detect this bug with Audee and it has been fixed by the PyTorch team [35].

| Answer to RQ4: The inconsistencies may be caused not only by implementation errors, but also by implementation differences and minor precision issues. The bugs in DL frameworks are usually caused by the non-robust handling for corner cases or the incorrect implementation logics. |

### 4 THREATS TO VALIDITY

The selection of DNNs and inputs could be a threat to validity, which may affect the results. To mitigate this threat, we select 7 widely used DNNs and 10,000 seed inputs for evaluating network mutation. For input and weight mutation, we randomly select 20 DNNs and 10,000 seed inputs. Note that, we mainly test the DL functionalities using convolutional and recurrent based DNNs in this work. Other functionalities (e.g., Transformer and Attention) are not included, due to the substantial experiment scale. However, this does not mean Audee is not capable of testing such functionalities, as long
as relevant seed DNNs and API configurations are given. We leave this part for our future work.

The thresholds in the localization may also affect the results when applying in other DNNs. To mitigate this threat, we carefully select the conservative thresholds by studying the distribution of top-1 change rates of 1,145 inconsistencies and 1,332 consistencies, respectively (i.e., Figure 4). Moreover, we manually check all unique inconsistencies that are localized by Audee and find there are no false positives in our results. The thresholds may miss the bugs which have small change rates. However, our evolitional approach can be used to amplify the change rate based on the input and weight mutations.

Another threat would be the randomness when comparing the NaN detection between TensorFuzz and Audee. To mitigate this issue, we run TensorFuzz and Audee for five times and calculate the average results.

5 RELATED WORK

In this section, we summarize the relevant work including DL framework testing and DL model testing.

5.1 DL Framework Testing

In recent years, there have been a lot of work focusing on the performance enhancement and quality assurance of DL frameworks. Liu et al. [25] and Guo et al. [17] conducted empirical studies to evaluate the performance and robustness of models during the training and inference phases. Their results revealed the implementation differences between multiple DL frameworks. For example, the configuration optimized for one framework does not necessarily work well on another framework [25]. Moreover, even the same DNN may produce different outputs on different frameworks [17]. These results motivate us to detect inconsistencies across DL frameworks through automated methods.

Zhang et al. [58] performed an empirical study on the bugs of DL systems using TensorFlow. Nargiz et al. [19] further conducted comprehensive studies on Keras, TensorFlow, and PyTorch with a summarized taxonomy of faults. Different from our work, they mainly focused on the application level (e.g., incorrect API usage and incorrect shape handling in the training program) instead of the DL framework implementations. Mahdi et al. [29] studied the assertion-based oracle approximation assertions in the current DL libraries. Our work adopts the differential testing technique for detecting inconsistencies without the need of explicit oracles. Saikat et al. [11] proposed an approach on testing the probabilistic programming systems. They used the generation-based approach to produce test cases based on some manually defined templates. By contrast, Audee adopts the search-based testing method to generate test cases for testing DL frameworks.

The most relevant work to Audee is CRADLE [34]. Our work distinguishes from CRADLE in the following aspects: 1) CRADLE applied the existing DNNs and inputs to detect bugs while we proposed a search-based testing approach for generating bugs; 2) we specially proposed an approach for detecting NaN errors, which is not involved in CRADLE; 3) our source localization is more fine-grained and can not only locate the buggy layers but identify buggy parameters as well; and 4) we further proposed an empirical study to better understand the inconsistencies and bugs. TensorFuzz [30] proposed a fuzzing technique for detecting NaN bugs by generating only “surprising” input data. By contrast, our search-based testing tends to generate unbalanced distributed values, which is proved to be more effective in triggering NaNs.

5.2 DL Model Testing

There are also many work focusing on the quality evaluation of DL models. For example, DeepXplore [33] introduced the neuron coverage for systematically measuring corner case behaviours of DNNs. DeepTest [50], DeepHunter [55], and DeepStellar [10] proposed the coverage-guided testing techniques to evaluate the CNN and RNN models. DiffChaser [56] adopted the genetic algorithm to identify disagreements between two DNNs, but it could only generate minor disagreements which are usually caused by the optimization differences. Audee aims at generating significant inconsistencies which are more likely to be caused by the bugs. Zhang et al. [57] characterized the inputs to DL models from the perspective of uncertainty and leveraged the uncertainty metrics as guidance to generate test inputs that are largely missed by existing techniques. In addition, there are some work proposed to test the ML classifiers [12, 38, 40] such as the KNN and NaiveBayes. It should be noted that, above approaches are orthogonal to our work as they mainly focus on evaluating the quality of DL models while Audee focuses on detecting and localizing bugs in DL frameworks.

6 CONCLUSION

In this paper, we propose Audee to detect and localize inconsistencies and bugs in DL frameworks. The generated test cases consist of diverse DNNs and inputs, with diverse layers, layer parameters and weights contained in these DNNs. After the inconsistencies and bugs are detected, Audee adopts a fine-grained causal-testing based method to localize the sources. We apply Audee in testing 4 widely used DL frameworks, where 26 unknown bugs are found and 7 of them have been confirmed or fixed by the developers. We further conduct an empirical study to understand the root causes of the inconsistencies and bugs. We find that inconsistencies can be caused by the implementation bugs, the slight precision differences, and totally different algorithms. Although the latter two cases are not considered as bugs, they can still lead to some quality issues during the DNN migration between DL frameworks. The bugs are usually caused by the non-robust handling for corner cases and the incorrect code logic in the DL framework implementations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comprehensive feedback. This work was partly supported by the National Science Foundation of China (No. 61872262, 61572349). It was also sponsored by the Singapore Ministry of Education Academic Research Fund Tier 1 (Award No. 2018-T1-002-069), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NSOE003-0001 and NRF Investigatorship NRFI06-2020-0022. We also gratefully acknowledge the support of NVIDIA AI Tech Center (NVAITC) to our research.


