Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing

Jiazhen Gu
School of Computer Science, Fudan University
Shanghai Key Lab. of Intelligent Information Processing
Shanghai, China

Xuchuan Luo
School of Computer Science
Fudan University
Shanghai Key Lab. of Intelligent Information Processing
Shanghai, China

Yangfan Zhou*
School of Computer Science
Fudan University
Shanghai Key Lab. of Intelligent Information Processing
Shanghai, China

Xin Wang
School of Computer Science
Fudan University
Shanghai Key Lab. of Intelligent Information Processing
Shanghai, China

ABSTRACT

Deep learning (DL) techniques are proven effective in many challenging tasks, and become widely-adopted in practice. However, previous work has shown that DL libraries, the basis of building and executing DL models, contain bugs and can cause severe consequences. Unfortunately, existing testing approaches still cannot comprehensively exercise DL libraries. They utilize existing trained models and only detect bugs in model inference phase. In this work we propose Muffin to address these issues. To this end, Muffin applies a specifically-designed model fuzzing approach, which allows it to generate diverse DL models to explore the target library, instead of relying only on existing trained models. Muffin makes differential testing feasible in the model training phase by tailoring a set of metrics to measure the inconsistencies between different DL libraries. In this way, Muffin can best exercise the library code to detect more bugs. To evaluate the effectiveness of Muffin, we conduct experiments on three widely-used DL libraries. The results demonstrate that Muffin can detect 39 new bugs in the latest release versions of popular DL libraries, including Tensorflow, CNTK, and Theano.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Software libraries and repositories.*

KEYWORDS

Deep Learning Testing, Library Testing, Model Generation, Fuzzing

*Yangfan Zhou is the corresponding author.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510092

ACM Reference Format:
Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510092

1 INTRODUCTION

Deep learning (DL) techniques have been proven effective in many specific tasks, such as image recognition [29], video understanding [53] and machine translation [49]. As a result, it becomes a trend to include DL-based functionality into traditional software design. DL systems (*i.e.*, software systems based on DL techniques) have been widely adopted in various domains in practice, *e.g.*, self-driving cars [28], virtual assistants [34], and software operations [11, 12, 25]. However, DL systems are shown to be lack of robustness, and thus cause real-world accidents. For instance, a Tesla driver was killed in self-driving mode that failed to brake the car in 2016 [6], and an Uber autonomous driving car killed a pedestrian in 2018 [7]. Errors/defects in DL systems can cause severe consequences, and even jeopardize human lives. Therefore, it is a critical task to test DL systems before deploying them in real production scenarios.

Unfortunately, how to test a DL system still remains an open challenge to the software engineering community. Many recent approaches focus on testing its core component, the DL model, *i.e.*, a deep neural network trained with a set of training data. Extensive work aims at improving the robustness of DL models via generating adequate test cases, *e.g.*, adversarial inputs or corner cases [48, 60]. Many studies also focus on designing criteria to measure the test adequacy [35, 43].

However, the execution of DL models relies on their back-end libraries (*i.e.*, DL libraries). Even with a correct model design, the outputs can be wrong if the underlying library contains bugs. Specifically, DL libraries provide high-level interfaces of the underlying various computation implementations (*e.g.*, matrix transformation, gradient calculation and weight update) over hardware infrastructure (*e.g.*, CPU and GPU). Bugs in DL libraries can inevitably cause unexpected outputs, or even fatal failure of DL systems [33]. But

one may tend to blame the DL model design, instead of its underlying library, when debugging [44], incurring more difficulty to the process. Hence, it is critical to investigate how to test DL libraries.

Recent efforts (*i.e.*, CRADLE [44] and LEMON [50]) on DL library testing focus on the *inference phase* of DL models. They adopt differential testing [26] to detect bugs, by comparing the inference results of existing, already-trained DL models with different DL libraries. Specifically, CRADLE directly use such models as test inputs, while LEMON further mutates such models as test inputs. However, even with these approaches, bugs still exist in DL libraries, as we have found in this work. The key reason is that they rely on the inference phase of already-trained models, which cannot exercise the library codes comprehensively. Such already-trained models typically involve only a small set of DL library functions. Moreover, DL libraries also play an important role in the model *training phase*, *e.g.*, the library codes for back propagation [30]. These library codes also cannot be exercised as well. But bugs in these codes can cause incorrect training results, *i.e.*, wrong resulting models.

Unfortunately, solving these concerns is a challenging task. First, it is hard, if not infeasible, to obtain tremendous, diverse already-trained models to comprehensively exercise library codes. Mutations based on such existing models also cannot solve this problem as they inherit the model structures, limiting the exploration of library functions. Moreover, as test oracles are not available generally, existing approaches [44, 50] resort to differential testing, based on comparing the model outputs with different DL libraries. However, such outputs are not existing in the training phase, incurring a huge challenge to applying differential testing.

In this work, we propose Muffin, a fuzzing-based approach to test DL libraries with high functionality coverage. Instead of relying on already-trained models, Muffin obtains diverse test inputs (*i.e.*, models) with an automatic model generation algorithm. It formulates model structure as a Directed Acyclic Graph (DAG), based on which it builds a model layer by layer with an aim to achieve high functionality coverage of DL libraries. To perform differential testing, Muffin relies on data trace analysis in the training phase. In particular, we divide the model training phase into three different stages (*i.e.*, forward calculation, loss calculation and gradient calculation), and accordingly design a set of metrics on the data traces to measure the consistency of results by different DL libraries. Inconsistencies can thus indicate potential bugs.

We apply Muffin to test 15 release versions of three widely-used DL libraries, *i.e.*, TensorFlow [5], CNTK [2], and Theano [47]. Muffin detects 39 new bugs (including 21 crash bugs) in the latest release versions of these libraries. Extensive experiments based on 6 popular datasets show that compared with existing approaches, Muffin is capable of detecting more inconsistencies within a comparable testing time. Furthermore, we investigate the benefit of our model generation method through comparing Muffin with layer-by-layer testing. The results show that Muffin is capable of detecting more inconsistencies and crashes. Our experiments prove the effectiveness of Muffin.

Muffin contributes to the software testing art in the following three aspects:

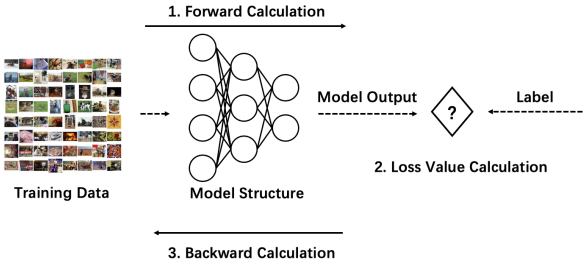


Figure 1: The training phase of a DL model

- We propose Muffin, a DL library testing approach based on a novel DL model fuzzing method, which can exercise DL library functionalities more comprehensively.
- We make differential testing feasible in testing the model training phase, by proposing a data trace analysis method to detect inconsistencies between different test targets.
- We implement our ideas as an open-source available software tool Muffin, which can facilitate real-world DL library testing tasks, as well as further follow-up research.
- We conduct an extensive study on 15 versions of three widely-used DL libraries. The results show that Muffin can detect 39 new bugs, which cannot be detected by previous methods.

The rest of paper is organized as follows. Section 2 introduces background knowledge about DL model and DL library. Section 3 elaborates the design and implementation details of Muffin. We demonstrate the experimental setup in Section 4, and analyze the results in Section 5. Further discussion is provided in Section 6. We introduce related work in Section 7 and conclude the paper in Section 8.

2 BACKGROUND

2.1 Deep Learning Model

DL models are designed to automatically draw statistical rules from training data [23]. A DL model typically consists of a number of neurons with a layered, connected structure. The neurons between layers are connected with links. Different links are associated with different weights, which are obtained through training with input data. Each layer conducts a specific kind of transformation (*e.g.*, convolution and pooling) for the input data with specific weights. In particular, the same layer can be adopted multiple times in a DL model, which has different weight values on the links and thus produces diverse results.

Essentially, a developer would design the architecture of a DL model such as the types of layers, how layers are connected and the loss function. Then the training process of a DL model is to find the appropriate weight values, so that the outputs can best produce expected results. The training phase typically consists of a huge amount of repeated training steps. Figure 1 outlines the process of a single training step, which can be divided into three stages:

- *Forward Calculation (FC)*: Given a batch of training cases, the model conducts specific calculations according to the layer types and get the corresponding outputs.

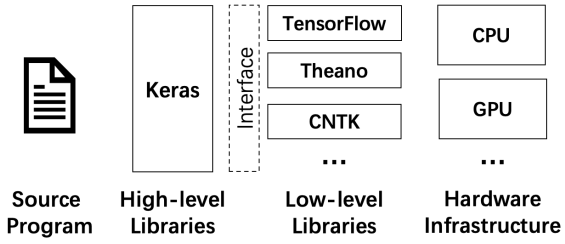


Figure 2: The structure of DL libraries

- *Loss Calculation (LC)*: The model calculates the value of the predefined loss function, which measures the differences between the model outputs and the ground-truth labels.
- *Backward Calculation (BC)*: According to the value of the loss function, the model calculates the gradients of each neuron and updates the corresponding weight values from the output layer to the input layer.

Such training steps continue until the weights converge, *i.e.*, the performance of the model cannot be further improved.

The weight values determine how the DL model processes the input to generate output. Thus, the performance of a DL model, *i.e.*, whether it can produce correct results, is largely determined by the weights. Since the weight values are obtained through the training process, it is critical to detect bugs in model training phase. However, existing work only focuses on detecting bugs in the inference phase [44, 50].

2.2 Deep Learning Library

Figure 2 shows the structure of DL libraries. There are two tiers of libraries (*i.e.*, high-level and low-level). In general, developers implement the source programs with high-level library APIs, which invoke the algorithms implemented in low-level libraries. Different low-level libraries are based on different infrastructures, *e.g.*, CPU, GPU and Tensor Processing Unit (TPU) [1], thus may have different implementations for the same algorithm specification. On the other hand, high-level DL libraries can hide the differences between low-level libraries and provide a consistent abstraction to facilitate DL model development.

Keras [4] is one of the most popular high-level DL libraries that has been widely used in various domains [15, 32, 40]. Keras generally runs on top of three low-level libraries, *i.e.*, TensorFlow, CNTK, and Theano, which cover most of the widely-used libraries. Developers implement source programs by calling APIs provided by Keras, which invoke the assigned backend low-level library to execute the computation.

Specifically, implementing a DL model using Keras mainly contains three parts: loading the data, defining the model architecture, and training the model with the data. It is worth noting that while the training process includes complicated calculations (*i.e.*, FC, LC and BC), it can be simply implemented via calling the "model.fit()" function provided by Keras.

A high-level library is relatively simple, which glues the functions in a low-level library that provide concrete, complicated computation. Low-level libraries are not bug free and they are also not easy to be tested, due to their complication. Similar to existing work [44, 50], we focus on testing low-level libraries, *e.g.*, TensorFlow, CNTK, and Theano. We adopt Keras as the high-level library. Our target is to test the DL library codes involved in the model training phase with high functionality coverage. Specifically, DL libraries contain many auxiliary codes for various tasks such as profiling and hardware adaptation, rather than learning-related ones. Like existing tools to test DL libraries, we focus only on learning-related APIs. In this paper, we use functionality coverage as the coverage metric, which refers to the percentage of the invoked APIs in all the pre-defined, learning-related APIs we considered.

2.3 Challenges

In order to perform comprehensive DL library testing (*e.g.*, test the library codes involved in model training), there are two main challenges. First, it is difficult to obtain a set of DL models as testing inputs that cover most library APIs. A DL model has a layered, connected structure, which hinders the adoption of traditional test input generation approaches. Furthermore, many APIs in DL libraries have specific usage scenarios, *e.g.*,*Convolution Layer* for image processing tasks, *Recurrent Layers* for text processing and various activation functions (*e.g.*, ReLU [41] and leaky-ReLU [57]). Due to such complication, it is non-trivial to obtain a set of well-trained models to achieve high functionality coverage.

The second challenge is the test oracle in model training phase. Existing approaches [44, 50] utilize differential testing based on the model outputs with different DL libraries. Unfortunately, as we have discussed, DL models learn the weight values through training. Therefore, the model outputs not exist in the training phase, causing existing differential testing methods infeasible.

Next, we introduce our approach, Muffin, which is designed to address the above two challenges.

3 APPROACH

3.1 Overview

In this work, we propose Muffin, a novel approach to perform comprehensive DL library testing, *i.e.*, test the library codes related to model training with high functionality coverage. Figure 3 presents the overview of Muffin, which is specifically tailored to solve the two design challenges.

To obtain diverse DL models, we propose a fuzzing-based model generation method. In contrast to existing methods that adopt manually-designed models, the proposed model generation approach allows Muffin to exercise the target library with tremendous, diverse models. Specifically, we divide the model architecture into two parts: structure information (*i.e.*, how layers are connected) and layer information (*i.e.*, what layer types are used). Through formulating the structure information of a DL model as a DAG, Muffin first generates DAGs as the structure information, and then utilizes a greedy layer selection algorithm to generate the layer information. In this way, Muffin can generate diverse DL models (Section 3.2).

To conduct differential testing, Muffin performs data trace analysis in the model training phase. In particular, Muffin profiles the data traces from different training stages (*i.e.*, FC, LC and BC). It then detects the inconsistencies of different libraries based on a

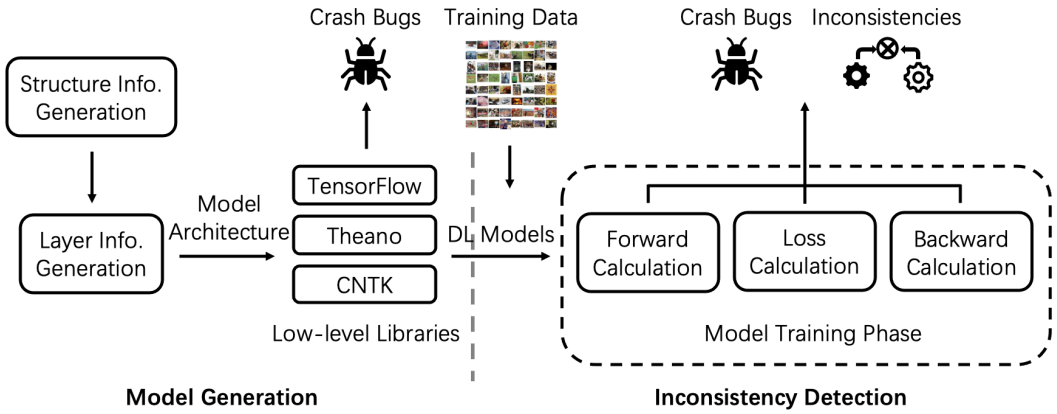


Figure 3: Overview of Muffin

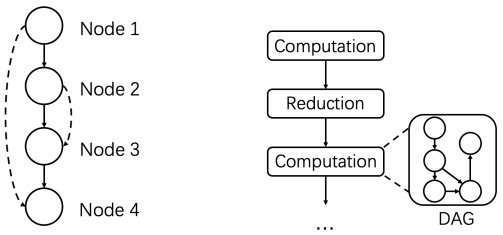
set of proposed metrics, which measures the output variance of consecutive layers. (Section 3.3).

3.2 Model Generation

As discussed in Section 2.1, a DL model has a layered structure with connections between layers. In order to generate a set of diverse DL models to explore library codes, we need to decide what types of layers are used in a model, as well as how these layers are connected. Unfortunately, simply selecting a series of layers and stacking them together can easily cause model failure. For example, the "Add" layer is used to add a list of inputs. If only one input is fed to this layer, the model generation would fail. Besides, the inputs of "Add" layer should also have the same shape to avoid failure generation. Therefore, we design a top-down generation algorithm, which first generates the *structure information* (*i.e.*, the topology of how layers are connected in the model), followed by the generating of the *layer information* (*i.e.*, specific layer types adopted in the model).

3.2.1 Structure Information Generation. Given a set of inputs, a DL model performs specific computation layer by layer, so as to yield the outputs. Therefore, the computation flow of a DL model can be abstracted as a DAG. Specifically, every vertex in the DAG represents a layer, and every edge between two vertices represents a link between the corresponding layers in the original model. Such an abstraction method is also applied in current model representation. For example, TensorFlow uses a DAG to represent the computational graph of a DL model [8]. Therefore, we utilize a DAG to represent the *structure information* of a DL model.

Although it is not difficult to generate a DAG, the corresponding model structure may be too simple or too complicated, which is rarely used in practice. Inspired by recent studies in Neural Architecture Search (NAS) [20], that targets on automating the design of model architectures, we summarize two model structure templates, as shown in Figure 4. Specifically, Figure 4(a) shows the chain structure with skips. Chain-structured architecture is the simplest example of the model structure topology. Through permitting arbitrary skip connections between nodes, this template can cover many commonly-used DL models (*e.g.*, fully-connected networks, VGG [45] and DenseNet [31]). On the other hand, the cell-based



(a) Chain structure with skips (b) Cell-based structure

Figure 4: Examples of DL model structure templates

structure in Figure 4(b) builds upon the observation that many specifically-designed model architectures consist of repetitions of fixed structures [52], *e.g.*, ResNet [29]. Each cell in the structure is a small DAG that conducts a specific transformation, *e.g.*, the computation cell in Figure 4(b) contains computation layers, while the reduction cell is used for downsampling. It is worth noting that originally the same cells (*e.g.*, computation cells) should have the same DAG. Since our target is generating diverse structures instead of finding the architecture with the best performance, we remove this restriction in the proposed template (*i.e.*, same cells may have different DAGs). In addition, we also guarantee that the generated DAG has only one vertex whose in-degree is 0 as the input layer, one vertex whose out-degree is 0 as the output layer. There is also no isolated vertex in the generated DAG. In this way, Muffin generates a DAG as the model structure information.

3.2.2 Layer Information Generation. Given the generated structure information, we need to refine the layer information, *i.e.*, determine the specific layer type for each vertex in the DAG. As discussed before, stacking layers without guidance can easily cause model failure. Specifically, there are two types of restrictions when selecting layers. The first restriction is the *input number* restriction. In particular, most layers (*e.g.*, "Convolution") are SI (Single-Input) layers, while some layers (*e.g.*, "Concatenation") are MI (Multiple-Input) ones. If more than one input is fed to an SI layer, this layer

can only process one of the inputs, leading to the existence of invalid connections between layers. The corresponding DAG thus is equivalent to the DAG without the invalid connections, which lowers the DAG diversity. Therefore, we have to choose the proper layer according to the number of inputs. Considering that an edge in a DAG represents the data flow direction, we can determine the number of inputs that the corresponding layer takes based on the *in-degree* of the vertex.

The second restriction is the *input/output shape* restriction. Specifically, MI layers require the inputs to have the same shape in specific axis(es) so as to conduct the transformation properly, *e.g.*, the inputs of "Concatenation" layer should have the same shape except for the concatenation axis. Therefore, before feeding the inputs to an MI layer, we adopt additional "Reshaping" layers to reshape the inputs into the same shape. In addition, the input shape of the input layer (*i.e.*, the vertex with 0 in-degree), and the output shape of the output layer (*i.e.*, the vertex with 0 out-degree) need to be properly set according to the training data and the task type (*e.g.*, classification, regression). We still resort to the "Reshaping" layer to reshape the output size, while the input shape is directly set based on the shape of input data.

Furthermore, in order to increase the diversity of the generated models, intuitively, we should give a larger chance to the layer that is rarely used before. Based on this intuition, we design a layer selection procedure based on Fitness Proportionate Selection [21]. Specifically, for a specific layer l, Muffin records the number of times that l has been selected to construct a model, denoted as c. Then Muffin calculates $s = \frac{1}{c+1}$ as the score for l. Based on s, the probability that l is selected among all layer types can be calculated as follows:

$$p = \frac{s}{\sum_{k=1}^{r} s_k} \tag{1}$$

where r is the total number of possible layers. Since we divide layer types into two categories (*i.e.*, SI and MI), score s and probability p are calculated based on the layers belonging to the same category. In this way, Muffin generates the layer information via selecting a specific layer for each vertex in the DAG. A DL model can thus be constructed according to the generated structure information and layer information.

3.2.3 Entire Algorithm. We formally describe our fuzzing-based model generation method in Algorithm 1. This algorithm takes 5 parameters, where N_m is the total number of models to generate, serving as the terminating condition; MAX_c and MAX_v are parameters to control the size of DAG; L_i and L_o should be manually set according to the input data and target task. Lines 2-33 iteratively generate a set of DL models. Specifically, lines 3-13 randomly choose a template and generate a DAG as the structure information. Lines 17-18 set the input layer. Lines 22-24 select SI layers for the 1 in-degree vertices, and Lines 26-29 select MI layers for the vertices with more than 1 in-degree. Lines 30-31 set the output layer. Finally, lines 32-33 construct a DL model m based on the generated structure information and layer information, then adding m to the result set M.

Algorithm 1: Model Architecture Generation

Input: N_m: Number of generated models
MAX_c: Maximum number of cells in a model
MAX_v: Maximum number of vertices in a DAG
L_i: Input shape
L_o: Output shape
Output: M: A set of generated models

1 $M \leftarrow \emptyset$;
2 **while** $Size(M) < N_m$ **do**
/* select a template and generate Structure Information SI */
3 $p \leftarrow Random(0, 1)$;
4 **if** $p < 0.5$ **then**
5 $N_v \leftarrow RandomInt(1, MAX_v)$;
6 $SI = CreateChainDAG(N_v)$;
7 **else**
8 $N_c \leftarrow RandomInt(1, MAX_c)$;
9 $G \leftarrow \emptyset$;
10 **for** *i from 1 to* N_c **do**
11 $G_i = RandomDAG()$;
12 $G \leftarrow G \cup \{G_i\}$;
13 $SI = CreateCellDAG(N_c, G)$;
14 $LI \leftarrow \emptyset$; /* Layer Information */
/* generate model according to DAG */
15 **foreach** *node j in* $TopologicalSequence(SI)$ **do**
16 **if** *the in-degree of node j is 0* **then**
17 $LI \leftarrow LI \cup \{SetInputLayer(L_i, j)\}$;
18 **else**
19 $P_j \leftarrow GetAllDirectPredecessors(j)$;
20 **if** $Size(P_j) == 1$ **then** /* SI layer */
21 $shape \leftarrow GetShape(P_j[0])$;
22 $LI \leftarrow LI \cup \{SetSILayer(shape, j)\}$;
23 $UpdateSIScore()$
24 **else** /* MI layer */
25 $shape \leftarrow RandomShape()$;
26 $LI \leftarrow LI \cup ReshapingLayers(P_j, shape)$;
27 $LI \leftarrow LI \cup \{SetMILayer(shape, j)\}$;
28 $UpdateMIScore()$
29 **if** *the out-degree of node j is 0* **then**
30 $LI \leftarrow LI \cup SetOutputLayer(j, L_o)$;
31 $m \leftarrow ConstructModel(SI, LI)$;
32 $M \leftarrow M \cup \{m\}$;
33 **return** M;

3.3 Inconsistency Detection

In order to detect inconsistencies and perform differential testing accordingly, Muffin requires proper metrics to measure the differences between the execution results of different libraries. However, the metrics proposed by the existing work are designed only for already-trained models, which calculate the inconsistency between the ground-truth label and model outputs. Since our target is to test DL library in the training phase (*i.e.*, without a trained model),

such metrics cannot be directly applied. Instead, we propose a new metric based on the variance of outputs in consecutive layers.

As demonstrated in Section 2.1, the model training phase includes repeated training steps, and each training step can be divided into three stages: FC, LC and BC. Specifically, in FC stage, the model performs calculation from input layer to output layer. In LC stage, the model calculates the value of loss function. In BC stage, the model calculates gradients from the output layer to the input layer. Resorting to dynamic analysis, we can collect the data traces of different libraries and compare the differences.

In particular, we utilize the *Functional API* mechanism provided by Keras to collect dynamic traces. More specifically, we profile the results produced by every layer in FC stage, the loss value in LC stage, and the gradient value of each layer in BC stage. Based on the dynamic trace, Muffin gradually compares the values of layer outputs, the loss, and the gradients, so as to detect the suspect behavior of specific layers.

However, due to normal uncertain factors such as floating-point deviation [22], we cannot determine whether a value difference is caused by potential bugs or normal factors. Specifically, there are many small deviations (less than 10^{-6}) in layer outputs, which may be gradually amplified or reduced, *e.g.*, by pooling or activation functions. We consider that normal factors would only lead to slight layer output difference, *i.e.*, if the differences of layer outputs change dramatically, it indicates a suspicious behavior. Therefore, instead of comparing the outputs from one single layer, we consider the difference-changes between two consecutive layers. Only when the deviation is amplified would Muffin consider inconsistency. Since outputs from different layers may have different shapes, we first use the following Chebyshev distance (*i.e.*, L_∞ distance) [9] to measure the difference of the outputs from the same layer.

$$D(X, Y) = \max_m(|x_m - y_m|) \tag{2}$$

In the above equation, X and Y are two tensors (*i.e.*, output of a layer is typically a high-dimensional tensor), while x_p and y_p are elements in X and Y, respectively. Chebyshev distance defines that the distance between two tensors is the greatest of their differences along any coordinate dimension. In this way, we can avoid the influence of different tensor shapes from different layers when measuring the differences.

We now describe the inconsistency detection procedure of Muffin. For brevity, we denote n as the total number of layers, $l_i, i \in [1, n]$ as the i^{th} layer, O^i_j and O^i_k as the outputs of l_i using library j and k, respectively. $P(i)$ denotes the set of layers that are direct predecessors of l_i in the DAG, *i.e.*, each layer in $P(i)$ is l_i's previous layer.

In FC stage, Muffin compares the differences of the output tensors from l_i and its predecessors l_p. If the difference of l_p is smaller than ϵ, while the difference of l_i is larger than a user-defined threshold t, then Muffin determine that an inconsistency is detected in l_i. The inconsistency layers detected in FC stage can be formally defined as follows:

$$Inc\_FC = \{l_i, i \in [1, n] \mid (D(O^i_j, O^i_k) > t) \wedge (D(O^p_j, O^p_k) < \epsilon, p \in P(i))\}$$

Table 1: Versions of libraries under test

ID	Keras	TensorFlow	Theano	CNTK
E1	2.3.1	2.0.0	1.0.4	2.7.0
E2	2.3.1	1.15.0	1.0.3	2.6.0
E3	2.2.4	1.12.0	1.0.2	2.5.0
E4	2.2.4	1.11.0	1.0.1	2.4.0
E5	2.2.4	1.10.0	1.0.0	2.3.0

In LC stage, the model calculates loss value based on the results from the output layer. To avoid the transmission of errors, Muffin only performs inconsistency detection in LC stage when the difference of model outputs is smaller than ϵ. It is worth noting that a small difference in model outputs does not mean that there is no inconsistency in middle layers. Large difference could be masked due to the existence of downsampling layers such as pooling. Since the result of a loss function is a number, we directly compare the absolute difference as follows:

$$Inc\_LC = \{L \mid ((|LO_j - LO_k| > t) \vee (|LG_j - LG_k| > t)) \wedge (D(O^n_j, O^n_k) < \epsilon)\}$$

In the above equation, L denotes the loss function, LO_j and LO_k are the output results of L, LG_j and LG_k are the gradient results of L, O^n_j and O^n_k are the model outputs, using library j and k.

In BC stage, the model calculates gradients to update weights from the output layer to the input layer. Similarly, Muffin only conducts inconsistency detection if the difference in loss function is smaller than ϵ. We formulate the inconsistency detection in BC stage as follows:

$$Inc\_BC = \{l_i, i \in [1, n] \mid (D(G^i_j, G^i_k) > t) \wedge (D(G^s_j, G^s_k) < \epsilon, s \in S(i))\}$$

where $S(i)$ denotes the set of layers that are direct successors of l_i; G^i_j and G^i_k denote the gradient result of l_i using different libraries. Especially, the successor of the output layer is the loss function.

4 EVALUATION SETUP

In the evaluation, we evaluate the performance of Muffin through answering the following research questions.

- **RQ1:** How does Muffin perform in detecting bugs in DL libraries?
- **RQ2:** Can Muffin achieve better performance compared to other methods?
- **RQ3:** How do the different parameter settings affect the performance of Muffin?

4.1 Libraries and Datasets

4.1.1 Libraries. We use three widely-used DL libraries (*i.e.*, TensorFlow, Theano, and CNTK) as the back-end low-level libraries as our testing targets, and Keras as the front-end high-level library. To sufficiently illustrate the effectiveness of Muffin, we utilize a total of 15 release versions of the three back-end libraries, and construct five experimental environments for differential testing, *i.e.*, E1-E5 in Table 1. In particular, in E1, Keras 2.3.1 is the latest version that supports multiple back-ends; Theano 1.0.4 and CNTK 2.7.0 are the latest versions, while TensorFlow 2.0.0 is the latest version that

supported by Keras. For the sake of brevity, we use TF, TH, and CK to represent TensorFlow, Theano and CNTK in the following figures and tables.

4.1.2 Datasets. Our approach is not sensitive to datasets, *i.e.*, theoretically any data type can be used for testing. In order to facilitate subsequent comparative experiments with existing approaches, we selected 6 widely-used datasets in existing studies [50], *i.e.*, MNIST, F-MNIST, CIFAR-10, ImageNet, Sine-Wave and Stock-Price. Specifically, the first four are popular image classification datasets, while the last two are sequence datasets. In particular, Sine-Wave is the sine function value sequence, and Stock-Price is the Disneyland stock price sequence from 1997 to 2016.

4.2 Competitors

In order to demonstrate the effectiveness and efficiency of Muffin, we compare Muffin with the state-of-the-art approach, LEMON [50]. LEMON performs DL library testing though mutating existing models to generate a huge amount of new test inputs. Following the evaluation setup in [50], we use 11 existing models (*i.e.*, AlexNet, LeNet5, ResNet50, MobileNetV1, InceptionV3, DenseNet121, VGG16, VGG19, Xception, LSTM-1, LSTM-2) as the seed models for mutation. By comparing with LEMON, we evaluate whether Muffin, based on directed test cases (*i.e.*, model) generation, can outperform LEMON in exposing bugs. Since LEMON cannot perform testing in the LC and BC stages, we only compare Muffin with LEMON through analyzing the number of inconsistencies and bugs detected in the FC stage. In addition, Muffin is designed to perform comprehensive library testing, so we also compare the functionality coverage achieved by Muffin and LEMON.

Besides, our DAG-based model generation is the core component of Muffin. Thus, it is also interesting to investigate the effectiveness of this component. To this end, we implement Muffin-UT, a simplified Muffin-version method based on unit testing. Muffin-UT differs from Muffin only in the model generation part. Specifically, in Muffin-UT, a functional layer (*e.g.*, Conv2D) is a to-be-tested unit. Muffin-UT creates models with only one functional layer, and simple reshaping layers to cope with input/output, *i.e.*, dimension transformation to match the input/output requirements of the to-be-tested layer. By comparing with Muffin-UT, we show that unit testing is still inadequate to test DL libraries. Muffin, by generating diverse models, can expose bugs which are difficult to be detected by traditional approaches.

4.3 Measurements

4.3.1 Number of inconsistencies. Since the proposed approach conducts inconsistency detection in layer level during model training, an inconsistency between two low-level DL libraries means that they produce different calculation results given the same input under a specific layer. In order to eliminate duplicated inconsistencies caused by the same function, we only count the inconsistencies produced by the same layer once. In particular, for Muffin and Muffin-UT, we compare the number of inconsistencies detected in different training stages, *i.e.*, FC, LC and BC, respectively. For LEMON, we only count the inconsistencies detected in FC stage. Although different inconsistencies may be the manifests of the same potential bug, more failure-triggering tests (*i.e.*, the model and input data that trigger the inconsistency) reflecting a fault in different ways provide more information for fault localization. Therefore, the number of detected inconsistencies can reflect the effects of these methods to some extent.

4.3.2 Number of detected bugs. Although we count the number of detected inconsistencies, it is more important to measure the number of unique bugs revealed by Muffin. Based on the voting mechanism of differential testing, we can localize the buggy layer in the library. To avoid false positives, we further check the buggy layer manually. Specifically, we save all intermediate layer outputs during the testing. When an inconsistency is reported, two authors check the corresponding source codes in different libraries and compare the results. If their identical layer produces different results and their implementation ideas are different, the third author will join manual inspection so as to conclude whether the report is true or false positive.

4.3.3 Number of NaN/Crash bugs. Besides inconsistent calculation results, bugs in DL libraries may lead to NaN (Not a Number) and crashes as well [44]. Generating DL models that trigger NaN/crashes can also provide valuable information for identifying potential bugs. Therefore, we count the number of models with NaN or crashes generated by three methods. In particular, we only count the NaN/crash when at least one of the DL library can execute properly, *e.g.*, TensorFlow produces normal results while Theano and CNTK produce NaN. In addition, in order to avoid duplication, the NaN caused by the same layer, and crashes with the same error message are only counted once.

4.4 Implementations

In the experiments, we let each method generate a total of 300 models, 50 for each dataset. For LEMON, we use its default parameters. For Muffin, we set the maximum number of cells (*i.e.*, MAX_c) to 5, and the maximum number of nodes (*i.e.*, MAX_v) to 30. In terms of inconsistency detection, we set the threshold t to be 0.15, and ϵ to be $1e^{-5}$. This t value is relatively large so as to avoid many false positives, as shown in Section 5.3. In addition, Muffin do not consider some layers such as "Dropout" and "GaussianNoise", so as to avoid introducing randomness and affecting the execution results.

All the experiments are conducted on the Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz machine with 32GB of RAM, Ubuntu 20.04.2 LTS, and one Nvidia GTX 1080 Ti GPU.

The implementation of Muffin is publicly available on GitHub [1].

5 RESULTS AND ANALYSIS

5.1 Effectiveness of Bug Detection

We first investigate the effectiveness of Muffin in terms of new bugs detected in the latest versions of different libraries, *i.e.*, E1 in Table 1. After manual analysis, Muffin detects 18 bugs in the latest version of these libraries, including 12 bugs in FC stage, 2 bugs in LC stage, 3 bugs in BC stage and 1 NaN bugs, as shown in Table 2. In addition, Muffin also detects 21 crash bugs, mainly from Theano and CNTK.

[1] https://github.com/library-testing/Muffin

Table 2: New bugs and crashes detected by Muffin

Library	FC Bug	LC Bug	BC Bug	NaN	Crash
TensorFlow	0	2	1	0(1)	1
Theano	8	0	2	0	10
CNTK	4	0	0	1	10
Total	18(1)				21

[1] "FC Bug""LC Bug""BC Bug" respectively refer to new bugs found in Forward Calculation, Loss Calculation and Backward Calculation stages.
[2] "NaN" refers to bugs related to NaN calculation.
[3] The number in parentheses means the bug exists in TensorFlow2.0.0 but has been fixed in the latest version. Other bugs are all exists in the latest version.

In particular, for the 4 bugs detected in TensorFlow 2.0.0, we manually check whether these bugs can be reproduced in the latest version (*i.e.*, TensorFlow 2.6.0). The results show that among these bugs, 1 bug has been fixed while the other 3 bugs still exist. After reporting these bugs to the issue repository, 1 bug has been confirmed by developers. Among the 4 bugs detected in CNTK, 1 will be fixed in the future version [3]. We also provide bug case analysis according to different bug types.

FC Bugs. The 12 bugs detected in FC stage involve different layer types, including "AveragePooling2D", "Conv1D" in Theano, and "LSTM", "DepthwiseConv2D", "BatchNormalization" in CNTK. By taking the "AveragePooling2D" bug in Theano as an example. This bug occurs when setting the layer parameter *padding* to "same" and *pool_size* to the same as the shape of the input tensor. By analyzing the results, we find that in this case, Theano would choose a wrong pooling location, resulting in large difference (*i.e.*, more than 13 while $t = 0.15$) between the results from other libraries.

LC Bugs. By taking the "BinaryCrossentropy" bug in TensorFlow as an example. When passing parameter values *output*=[0., 1., 0.] and *target*=[0.9999999, 0.9999999, 0.0000001] to the "BinaryCrossentropy" loss function, theano and CNTK return a value [15.942385, 1.1920930e-07, 1.1920930e-07] while TensorFlow returns [15.333239, -0., -0.], among which the difference of the first element is not negligible. By reviewing the source code, we find that TensorFlow redundantly uses an *epsilon* parameter to clip input values, resulting in errors. This bug has been confirmed by the developers of TensorFlow.

BC Bugs. By taking the "ReLU" bug in Theano as an example. When 0 exists in the input tensor of $ReLU$, theano back-propagates a different gradients value, compared with TensorFlow and CNTK. This bug is caused by the wrong equal sign position of Theano, *i.e.*, $ReLU(z) = z \mid z \geq 0$ in Theano, while $ReLU(z) = z \mid z > 0$ in other libraries. Although such implementation does not affect the results in forward calculation, the implementation in Theano would let the gradient propagate to previous layers in backward calculation (which should not happen). This bug can only be detected in BC stage, proving the effectiveness of Muffin.

NaN Bugs. By taking a TensorFlow bug as an example. Given two NaN value, the "GlobalMaxPooling" layer returns $-INF$, leading to the inconsistency. This bug has been fixed in the latest 2.6.0 version.

Regarding false positives, Muffin reports 19 unique inconsistencies totally, where one false positive is found. The false positive occurs in the "mean_absolute_percentage_error" loss function. This function returns $100 \times mean$, which amplifies the deviation and cause the false alarm. In addition, Muffin detects 25 crash bugs totally. Among them four are due to unsupported models Muffin generates, which can be treated as false positives. But, such false positives have clear error messages, thus can be automatically detected so as to avoid false alarms.

Table 3: Comparison of distinct voted layers

Method	Lib	FC	LC	BC
Muffin	**TF**	3 (2)	1 (1)	1 (1)
	TH	15 (5)	1 (0)	1 (1)
	CK	6 (2)	1 (0)	1 (1)
LEMON	**TF**	2 (0)	-	-
	TH	1 (0)	-	-
	CK	1 (0)	-	-
Muffin-UT	**TF**	4 (2)	2 (2)	2 (2)
	TH	11 (1)	1 (0)	2 (2)
	CK	4 (0)	1 (0)	0

[1] The number in parentheses denotes the number of voted layers that ONLY detected by the corresponding method.

To further illustrate the effectiveness of Muffin, we count the number of distinct voted layers detected by different approaches, as shown in table 3. We can observe that all the 4 layers detected by LEMON can be detected by Muffin and Muffin-UT, indicating that Muffin and Muffin-UT can cover the exploration scope of LEMON. On the other hand, Muffin and Muffin-UT have their own distinct voted layers that cannot be detected by the other, proving the effectiveness of inconsistency detection approach. These results indicate that the natural architecture fuzzing approach adopted in Muffin is a good supplement to unit testing.

5.2 Performance Comparison

In order to further evaluate the performance of Muffin, we compare the number of inconsistencies, NaN, and crash detected by different methods. We present the results under environment E1 as an example[2]. Table 4 shows the inconsistencies detected by three methods under different datasets and environments. Specifically, in the latest library versions (*i.e.*, E1), Muffin finds a total of 54 inconsistencies, 45 of which are found in the FC stage. In comparison, LEMON can only find 7 inconsistencies, much less than Muffin. Similar results can also be observed in other environments, which prove the effectiveness of Muffin in library testing. The main reason is that Muffin can explore more library functions through the model generation approach, while LEMON can only mutate seed models, and thus can hardly cover the functions not being used in seed models. Besides, LEMON also cannot explore the library codes related to loss and gradient calculation. As a result, LEMON only achieves 35.593% functionality coverage (the percentage of the invoked APIs in all the pre-defined, learning-related APIs we considered), while Muffin can achieve 98.305% functionality coverage. The inconsistent APIs that cannot be identified by LEMON include "DepthwiseConv2D", "LocallyConnected1D", "Conv3D" and various loss functions. It is worth noting that although Muffin is not designed to achieve high line coverage, we summarize and report the line coverage results: Muffin achieves 43.22%, which is 2.07 times of that achieved by LEMON (20.85%).

On the other hand, compared with Muffin-UT, Muffin detects 19 more inconsistencies in E1, which proves the performance of

[2]More experiment codes and results are available at https://github.com/library-testing/Muffin

Table 4: Comparison of inconsistency number

ID	Method	Lib Pair	CIFAR-10			MNIST			F-MNIST			ImageNet			Sine-Wave			Stock-Price			Total		
			FC	LC	BC	FC	LC	BC	FC	LC	BC	FC	LC	BC	FC	LC	BC	FC	LC	BC	FC	LC	BC
E1	Muffin	TF-TH	4	0	0	4	0	0	7	1	1	1	1	0	5	0	0	3	1	0	16	2	1
		TF-CK	3	1	0	4	2	1	6	2	1	3	1	0	12	0	0	8	0	0	16	2	2
		TH-CK	4	1	0	4	0	0	7	1	1	1	0	0	6	0	0	2	0	0	13	1	1
	LEMON	TF-TH	1	-	-	0	-	-	1	-	-	1	-	-	0	-	-	0	-	-	2	-	-
		TF-CK	1	-	-	0	-	-	0	-	-	1	-	-	1	-	-	0	-	-	3	-	-
		TH-CK	0	-	-	0	-	-	1	-	-	0	-	-	1	-	-	0	-	-	2	-	-
	Muffin-UT	TF-TH	1	0	3	4	2	2	2	0	2	5	0	0	1	0	0	0	1	0	9	2	3
		TF-CK	2	1	1	1	2	1	1	3	1	2	2	0	2	0	0	4	1	0	5	3	1
		TH-CK	1	0	0	3	1	1	2	0	1	5	0	0	2	0	0	4	0	0	10	1	1

[1] "FC""LC""BC" respectively represent the number of inconsistencies detected in the three stages.
[2] For inconsistencies caused by the same kind of layer (or loss function), we only count once.

Table 5: Comparison of NaN/Crash number

ID	Method	Lib	CIFAR-10			MNIST			F-MNIST			ImageNet			Sine-Wave			Stock-Price			Total		
			NaN	GC	EC	NaN	GC	EC	NaN	GC	EC	NaN	GC	EC	NaN	GC	EC	NaN	GC	EC	NaN	GC	EC
E1	Muffin	TF	2	0	1	3	0	1	3	0	1	5	0	1	4	0	1	5	0	1	7	0	1
		TH	3	0	10	2	0	10	1	0	3	2	0	3	3	0	8	3	0	4	6	0	10
		CK	2	6	4	3	6	4	3	4	3	4	2	2	4	4	4	5	4	4	7	6	4
	LEMON	-	0	-	0	0	-	0	0	-	0	0	-	0	0	-	0	0	-	0	0	-	0
	Muffin-UT	TF	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	4	0	0
		TH	0	0	3	0	0	3	0	0	2	0	0	2	4	0	3	0	0	3	4	0	3
		CK	0	1	4	0	0	4	0	0	3	0	0	2	5	0	4	0	0	3	5	1	4

[1] "NaN" represents the number of outputs with NaN. For NaN caused by the same kind of layer, we only count once.
[2] "GC" and "EC" are respectively short for "Generation Crash" and "Execution Crash". The crash number have been deduplicated according to error messages.
[3] LEMON does not record NaN/crash information for each backend, so we only obtain the total NaN/crash number triggered by LEMON.

Muffin. It is also worth noting that the number of inconsistencies detected by Muffin reduces in other environments. The key reason is that the numbers of NaN and crash triggered by Muffin increase in old library versions, as shown in Table 5. Taking NaN and crash into consideration, Muffin can still trigger more exceptions (*i.e.*, inconsistency, NaN and crash) than Muffin-UT. In particular, the layer functions where Muffin can detect exceptions while Muffin-UT cannot include "AveragePooling1D", "Conv3DTranspose" and "CategoricalCrossentropy".

Furthermore, we also compare the execution time of the three methods to generate 50 models and perform testing under different datasets. The execution time of Muffin and Muffin-UT consists of the model generation time and the three-stage inconsistency detection time (*i.e.*, FC, LC and BC). The execution time of LEMON consists of model mutation time and inconsistency detection time (only FC). The results are shown in Table 6.

In this table, we can observe that except ImageNet, the execution time of Muffin is the longest in most cases. The reason is that Muffin conducts additional model generation (compared with Muffin-UT), and inconsistency detection in additional two stages (compare with LEMON). Considering that Muffin can detect much more inconsistencies, we think such overhead (*i.e.*, around ten minutes) is acceptable. These results also demonstrate that the proposed approach (model generation and inconsistency detection) do not bring huge overhead to Muffin.

Moreover, when performing library testing with ImageNet, the execution time of LEMON is greatly increased. The reason is that the seed models used by LEMON are much more complicated, compared to those under other datasets. This phenomenon reveals that the execution time of LEMON highly depends on the complexity of seed models. On the other hand, Muffin does not suffer from this problem. The generated model complexity of Muffin can be controlled via setting proper values of MAX_c and MAX_v. Under the same MAX_c and MAX_v value, the execution time of Muffin is quite stable.

Table 6: Comparison of execution time (MIN.)

Dataset	Method	E1	E2	E3	E4	E5
CIFAR-10	Muffin	29.20	20.67	38.75	27.38	16.75
	LEMON	27.20	28.02	32.48	34.70	28.60
	Muffin-UT	16.27	14.72	13.13	11.00	13.47
MNIST	Muffin	32.35	19.82	18.20	14.95	15.78
	LEMON	10.58	10.28	9.88	9.67	9.28
	Muffin-UT	14.87	14.88	12.40	11.02	13.20
F-MNIST	Muffin	24.78	25.00	18.23	15.92	20.38
	LEMON	12.62	12.88	12.27	11.67	11.27
	Muffin-UT	15.85	15.32	12.90	11.67	12.78
ImageNet	Muffin	49.04	40.72	38.12	34.50	28.22
	LEMON	80.25	117.62	114.25	117.25	111.93
	Muffin-UT	34.03	23.52	30.15	37.05	27.55
Sine-Wave	Muffin	25.95	24.37	18.52	15.40	17.45
	LEMON	15.37	14.37	13.67	13.37	13.01
	Muffin-UT	17.78	16.45	13.40	12.60	14.83
Stock-Price	Muffin	22.03	18.78	16.35	13.28	14.28
	LEMON	16.58	15.50	14.35	14.37	13.82
	Muffin-UT	16.07	14.22	12.45	10.85	13.25

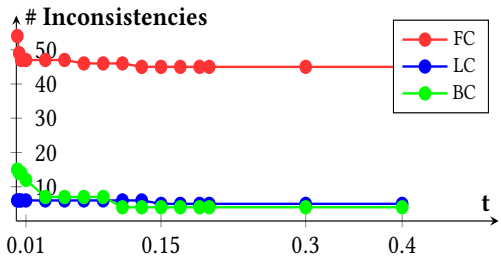


Figure 5: Performance of Muffin under different thresholds

Compared with Muffin-UT, Muffin requires additional DAG-based model generation. In addition, the number of layers in the model also affects the inconsistency detection time. The larger the model, the longer the detection time. As a result, the execution time of Muffin is slightly longer than that of Muffin-UT.

Finally, it is worth noting Muffin does not consider the final model performance (*e.g.*, precision and recall) in specific tasks when generating model architectures, since it is not the objective of a testing tool. In contrast, existing approaches (*e.g.*, mutating existing models) typically generate limited model architectures but can obtain high-performance models, which however, are not more capable in detecting bugs. Instead, Muffin focuses more on model quality in testing. Muffin can generate high-quality models. In 900 executions (3 libraries, each with 300 models), only 77 executions (8.5%) cause four unsupported-crashes (by the same reason). Moreover, we have also shown such models are more capable in exposing bugs, as discussed in Section 5.1.

5.3 Effect of Different Parameter Settings

Muffin introduces four parameters, *i.e.*, MAX_c and MAX_v to control the size of model structure, and threshold t and ϵ for inconsistency detection. Since in all experiments, Muffin achieves satisfying layer coverage (*i.e.*, only one layer cannot be used with all datasets), we consider that the values of MAX_c and MAX_v are set properly. For the thresholds, ϵ is a quite small value (*i.e.*, $1e^{-5}$), thus we only evaluate the number of inconsistencies detected by Muffin with different t values.

Figure 5 shows the number of inconsistencies detected by Muffin under different t values, ranging from 0.001 to 0.4. We can observe that when t is small, Muffin is more sensitive to small variance of differences, thus it detects more inconsistencies. As the value of t increases, the number of inconsistencies decreases slowly, and keeps stable when $t \in [0.15, 0.4]$. Thus, the default t value in Muffin is 0.15. Although bugs incurring small variances may be neglected, such bugs can be revealed under other input values or model architectures (*i.e.*, variance larger than t).

6 DISCUSSION

6.1 Summary of Evaluation

As discussed before, Muffin-UT is designed based on the idea of unit testing, which tests a specific library function at a time. The evaluation results show that Muffin can detect more layer inconsistencies than Muffin-UT. The main reason is that many layer inconsistencies can only be triggered by specific inputs. For instance, the gradients inconsistency of "MaxPooling1D" layer only happens when multiple elements in the input tensor have the same maximum value. In order to trigger such inconsistencies using Muffin-UT, we have to fuzzing the inputs. Since layers in DL libraries typically have huge input value ranges, *e.g.*, high-dimensional tensor inputs where each element ranges from $(-\infty, \infty)$, it is quite challenging to find specific inputs that can trigger corner cases [54]. On the other hand, Muffin performs testing based on generated models. Due to the existence of different layer types, we thus simulates the real calculation process and reduce the input ranges. As a result, the possible corner cases (*i.e.*, multiple maximum values) can be triggered by Muffin. Considering that unit testing is necessary before version release, while bugs can still be detected in the latest versions, we believe DL library testing based on model generation is an effective supplement to unit testing.

Among the bugs detected by Muffin, some of them are actually caused by unclear specifications. For instance, the gradient calculation bug of "categorical_hinge" loss function is actually caused by the different specification of calculating the gradients of "max()" function. Specifically, when there are multiple maximum elements, TensorFlow will divide the gradient with the number of maximum element, while Theano and CNTK do not have this operation. Similarly, for "MaxPooling1D" layer, when there are multiple maximum elements, TensorFlow and CNTK would only apply the gradients to one of the maximum elements, while Theano apply the gradients to all maximum elements. Due to unclear specifications, different DL libraries have different implementations. Although in most cases the results of these implementations are consistent, robustness issues may be caused by the corner cases (*e.g.*, easier to generate adversarial inputs [24]). Therefore, we call for the community to pay more attention on the unclear specification problems in DL libraries.

6.2 Threats to Validity

We now discuss possible threats in this work, and the methods we take to address such threats. First of all, we only evaluate the effectiveness of Muffin on three DL libraries, *i.e.*, TensorFlow, Theano and CNTK. These libraries can be called using the same front-end library (*i.e.*, Keras), which facilitate the implementation and performing differential testing. Other libraries that do not support Keras (*i.e.*, PyTorch) currently are not supported by Muffin. However, the ideas of model generation and inconsistency detection adopted in Muffin are general. For instance, in order to test PyTorch, it requires to replace the Keras APIs used in Muffin with the corresponding PyTorch APIs. To reduce this treat, we evaluate Muffin with a total of 15 different release versions of DL libraries. In addition, we also use diverse models (including the models generated by Muffin, existing models on 6 real datasets, as well as their mutants generated by existing work) to evaluate the inconsistency detection performance of our approach.

Another threat mainly lies in randomness and threshold settings in our experiment. To reduce the randomness, we conduct five experiments with different library versions (*i.e.*, E1-E5, refer to Table 1, Table2 in Supplementary Material). In each experiment, every method generates/mutates the same number of models for 6 commonly-used datasets, and we record and compare the results

and execution time. For threshold settings (*e.g.*, t, MAX_c and MAX_v), since in every experiment Muffin achieves 58/59 function usage, we do not increase MAX_c and MAX_v. For threshold t, as discussed in Section 4.4, we choose a quite large threshold. Slightly changing t (*e.g.*, from 0.15 to 0.4) has little impact on the results.

6.3 Future Directions

Muffin can be potentially improved in the following two aspects. First, Muffin only covers library codes in the layer function granularity through trying to generate model covering all the provided APIs. However, there may still be a large portion of library codes that cannot be covered (*e.g.*, private methods, branches). In the future, Muffin can be extended to consider other coverage metrics (*e.g.*, line coverage, branch coverage), and conducts model generation/mutation to explore more library codes.

Second, Muffin still relies on differential testing to solve the test oracle problem. However, if different DL libraries produce the same wrong results, Muffin cannot identify such bugs. Moreover, in the evaluation, we also notice that under certain circumstances, the model generated by Muffin may cause one library to crash, while the other two produce inconsistent results. In such cases, it requires huge human efforts to identify potential bugs. To get rid of this limitation, we intend to design metamorphic relations based on the properties of DL models, and conducts metamorphic testing to test one library accordingly.

7 RELATED WORK

As discussed before, CRADLE [44] and LEMON [50] are the most related work to ours that targets DL library testing, both of which require existing DL models and only detect bugs in model inference phase. Different from them, Muffin detects DL library bugs in model training phase via DAG-based model generation. In the literature, there is a body of work focusing on testing machine learning (ML) libraries as well [17–19, 56, 59]. For instance, Dutta *et al.* [17] propose ProbFuzz to test probabilistic programming systems via generating programs based on pre-defined templates. Dwarakanath *et al.* [19] adopt metamorphic testing to test image classification applications through mutating the training and testing data. However, these approaches cannot be directly adopted for DL libraries testing.

On the other hand, there is a great deal of researches focusing on the testing of DL models [35–38, 42, 43, 48, 54, 55]. In particular, many research efforts have been put on designing criteria to measure test adequacy [16, 35, 37]. For instance, Pei *et al.* [43] first propose neuron coverage as the criteria for testing DL models. Ma *et al.* [37] further define both neuron and layer level coverage criteria to help gauging the testing quality of DL models. Kim *et al.* [35] propose surprise coverage based on surprise adequacy, which measures relative surprise of each input with respect to the training data. Du *et al.* [16] propose a set of similarity metrics and coverage criteria to analyze stateful DL systems such as Recurrent Neural Networks (RNNs) [39]. Moreover, there are a lot of studies intend to reveal defects in DL models via generating adversarial inputs or finding corner cases [48, 55, 60]. For instance, Tian *et al.* [48] implement DeepTest for detecting erroneous behaviors of DL-based self-driving cars via automatically generating test cases based on image transformations. Similarly, Zhang *et al.* [60] implement DeepRoad, which applies Generative Adversarial Networks (GANs) [24] to test DL-based self-driving cars. Besides, there are also many researches focus on detecting different kinds of bugs in model structures or training parameter settings [51, 62]. For instance, Zhang *et al.* [62] propose DEBAR, a static analysis approach for detecting numerical bugs in DL models. Wardat *et al.* [51] propose a dynamic analysis based approach to detect numerical errors when training DL models. Similarly, Zhang *et al.* [61] propose AUTOTRAINER, a tool that detects and auto-repairs commonly-seen model training problems such as vanishing gradient, exploding gradients and slow convergence. Different from them, our work focuses on testing DL libraries rather than DL models or parameters.

Our work is also related to differential testing, an effective method that use similar programs as cross referencing oracles to detect bugs [26]. Differential testing has been successful in uncovering bugs across various types of programs, such as compilers [58], Java Virtual Machine (JVM) implementations [13, 14], web applications [10], and security-related APIs [46]. In recent years, researchers also utilize differential testing in the area of DL testing [43, 48]. For instance, Pei *et al.* [43] propose DeepXplore, a differential testing framework to identify DL model defects via image transformation. Guo *et al.* [27] propose DLFuzz, a differential fuzzing testing framework that exposes DL model errors through mutating inputs to maximize model output difference. These approaches focus on testing DL models, while Muffin is designed for DL library testing with high coverage.

8 CONCLUSION

In this paper, we propose a novel approach to test DL library codes via direct model generation using library APIs. In order to generate diverse DL models, we use DAG to formulate the model structure and propose a DAG-based model generation algorithm. In order to detect bugs, we divide the model training phase into three stages, and design different measurements for each stage. In this way, our approach can detect library bugs related to model training, which is not covered by previous studies. We implement our approach as an open-source tool called Muffin. To evaluate the performance of Muffin, we conduct a series of experiments based on 15 release versions of three widely-used DL libraries. Muffin detects 39 new bugs in the latest versions of these libraries. Besides, Muffin outperforms other methods in terms of the number of detected unique inconsistencies.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China under Grant 2020YFA0711400 and the Natural Science Foundation of Shanghai (No. 22ZR1407900).

REFERENCES

[1] Accessed: 2021. Cloud TPU. https://cloud.google.com/tpu.
[2] Accessed: 2021. CNTK. https://docs.microsoft.com/cognitive-toolkit.
[3] Accessed: 2021. CNTK ops Pachakge: sqrt. https://docs.microsoft.com/en-us/python/api/cntk/cntk.ops?view=cntk-py-2.7#sqrt-x--name----.
[4] Accessed: 2021. Keras. https://keras.io.
[5] Accessed: 2021. TensorFlow. https://www.tensorflow.org.
[6] Accessed: 2021. Tesla driver dies in first fatal crash while using autopilot mode. https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-

death-self-driving-car-elon-musk.
[7] Accessed: 2021. Uber's self-driving operator charged over fatal crash. https://www.bbc.com/news/technology-54175359.
[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. USENIX Association, 265–283.
[9] Cyrus D Cantrell. 2000. *Modern mathematical methods for physicists and engineers*. Cambridge University Press.
[10] Peter Chapman and David Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proc. of the 18th ACM Conference on Computer and Communications Security, CCS*. ACM, 263–274. https://doi.org/10.1145/2046707.2046737
[11] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 364–375. https://doi.org/10.1109/ASE.2019.00042
[12] Junjie Chen, Shu Zhang, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Yu Kang, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2020. How Incidental are the Incidents? Characterizing and Prioritizing Incidents for Large-Scale Online Service Systems. In *Proc. of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 373–384. https://doi.org/10.1145/3324884.3416624
[13] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proc. of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127
[14] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 85–99. https://doi.org/10.1145/2908080.2908095
[15] Keunwoo Choi, György Fazekas, Mark B. Sandler, and Kyunghyun Cho. 2017. Convolutional recurrent neural networks for music classification. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*. IEEE, 2392–2396. https://doi.org/10.1109/ICASSP.2017.7952585
[16] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: model-based quantitative analysis of stateful deep learning systems. In *Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 477–487. https://doi.org/10.1145/3338906.3338954
[17] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 574–586. https://doi.org/10.1145/3236024.3236057
[18] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 729–739. https://doi.org/10.1145/3338906.3338972
[19] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 118–128. https://doi.org/10.1145/3213846.3213858
[20] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21.
[21] David B. Fogel. 1997. Evolutionary algorithms in theory and practice. *Complex.* 2, 4 (1997), 26–27. https://doi.org/10.1002/(SICI)1099-0526(199703/04)2:4<26::AID-CPLX6>3.0.CO;2-7
[22] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.* 23, 1 (1991), 5–48. https://doi.org/10.1145/103162.103163
[23] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org/
[24] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems*. 2672–2680.
[25] Jiazhen Gu, Jiaqi Wen, Zijian Wang, Pu Zhao, Chuan Luo, Yu Kang, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Bo Qiao, Liqun Li, Qingwei Lin, and Dongmei Zhang. 2020. Efficient customer incident triage via linking with system incidents. In *Proc. of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 1296–1307. https://doi.org/10.1145/3368089.3417061
[26] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *Proc. of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*. IEEE / ACM, 71–80. https://doi.org/10.1109/ICSE-SEIP.2019.00016
[27] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 739–743. https://doi.org/10.1145/3236024.3264835
[28] Abhishek Gupta, Alagan Anpalagan, Ling Guan, and Ahmed Shaharyar Khwaja. 2021. Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array* 10 (2021), 100057. https://doi.org/10.1016/j.array.2021.100057
[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90
[30] Robert Hecht-Nielsen. 1988. Theory of the backpropagation neural network. *Neural Networks* 1, Supplement-1 (1988), 445–448. https://doi.org/10.1016/0893-6080(88)90469-8
[31] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. IEEE Computer Society, 2261–2269. https://doi.org/10.1109/CVPR.2017.243
[32] Jane Hung, Allen Goodman, Deepali Ravel, Stefanie Lopes, Gabriel Rangel, Odailton A. Nery, Benoit Malleret, Francois Nosten, Marcus V. G. Lacerda, Marcelo U. Ferreira, Laurent Rénia, Manoj Duraisingh, Fabio T. M. Costa, Matthias Marti, and Anne E. Carpenter. 2020. Keras R-CNN: library for cell detection in biological images using deep neural networks. *BMC Bioinform.* 21, 1 (2020), 300. https://doi.org/10.1186/s12859-020-03635-x
[33] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 510–520. https://doi.org/10.1145/3338906.3338955
[34] Veton Kepuska and Gamal Bohouta. 2018. Next-generation of virtual personal assistants (Microsoft Cortana, Apple Siri, Amazon Alexa and Google Home). In *IEEE 8th Annual Computing and Communication Workshop and Conference, CCWC*. IEEE, 99–103. https://doi.org/10.1109/CCWC.2018.8301638
[35] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proc. of the 41st International Conference on Software Engineering,ICSE*. IEEE / ACM, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108
[36] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. ACM, 120–131. https://doi.org/10.1145/3238147.3238202
[37] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *Proc. of the 29th IEEE International Symposium on Software Reliability Engineering, ISSRE*. IEEE Computer Society, 100–111. https://doi.org/10.1109/ISSRE.2018.00021
[38] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 175–186. https://doi.org/10.1145/3236024.3236082
[39] Tomás Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proc. of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH*. ISCA, 1045–1048.
[40] Amita Muralikrishna, Luís Eduardo Antunes Vieira, Rafael Duarte Coelho dos Santos, and Adriano P. Almeida. 2020. Total Solar Irradiance Forecasting with Keras Recurrent Neural Networks. In *20th International Conference on Computational Science and Its Applications - ICCSA (Lecture Notes in Computer Science, Vol. 12253)*. Springer, 255–269. https://doi.org/10.1007/978-3-030-58814-4_18
[41] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. of the 27th International Conference on Machine Learning, ICML*. Omnipress, 807–814.
[42] Augustus Odena, Catherine Olsson, David G. Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proc. of the 36th International Conference on Machine Learning, ICML (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 4901–4911.

[43] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proc. of the 26th Symposium on Operating Systems Principles, SOSP*. ACM, 1–18. https://doi.org/10.1145/3132747.3132785

[44] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proc. of the 41st International Conference on Software Engineering,ICSE*. IEEE / ACM, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[45] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proc. of the 3rd International Conference on Learning Representations, ICLR*, Yoshua Bengio and Yann LeCun (Eds.).

[46] Varun Srivastava, Michael D. Bond, Kathryn S. McKinley, and Vitaly Shmatikov. 2011. A security policy oracle: detecting security holes using multiple API implementations. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 343–354. https://doi.org/10.1145/1993316.1993539

[47] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016).

[48] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proc. of the 40th International Conference on Software Engineering,ICSE*. ACM, 303–314. https://doi.org/10.1145/3180155.3180220

[49] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. 2019. Learning Deep Transformer Models for Machine Translation. In *Proc. of the 57th Conference of the Association for Computational Linguistics, ACL, Volume 1: Long Papers*. Association for Computational Linguistics, 1810–1822. https://doi.org/10.18653/v1/P19-1176

[50] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proc. of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 788–799. https://doi.org/10.1145/3368089.3409761

[51] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *Proc.of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 251–262. https://doi.org/10.1109/ICSE43902.2021.00034

[52] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. 2019. A Survey on Neural Architecture Search. *CoRR* abs/1905.01392 (2019). arXiv:1905.01392 http://arxiv.org/abs/1905.01392

[53] Chao-Yuan Wu, Christoph Feichtenhofer, Haoqi Fan, Kaiming He, Philipp Krähenbühl, and Ross B. Girshick. 2019. Long-Term Feature Banks for Detailed Video Understanding. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. Computer Vision Foundation / IEEE, 284–293. https://doi.org/10.1109/CVPR.2019.00037

[54] Weibin Wu, Hui Xu, Sanqiang Zhong, Michael R. Lyu, and Irwin King. 2019. Deep Validation: Toward Detecting Real-World Corner Cases for Deep Neural Networks. In *Proc. of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. IEEE, 125–137. https://doi.org/10.1109/DSN.2019.00026

[55] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proc. of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 146–157. https://doi.org/10.1145/3293882.3330579

[56] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. 2020. METTLE: A METamorphic Testing Approach to Assessing and Validating Unsupervised Machine Learning Systems. *IEEE Trans. Reliab.* 69, 4 (2020), 1293–1322. https://doi.org/10.1109/TR.2020.2972266

[57] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. 2015. Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR* abs/1505.00853 (2015). arXiv:1505.00853

[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 283–294. https://doi.org/10.1145/1993316.1993532

[59] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[60] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. ACM, 132–142. https://doi.org/10.1145/3238147.3238187

[61] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *Proc. of the 43rd IEEE/ACM International Conference on Software Engineering,ICSE*. IEEE, 359–371. https://doi.org/10.1109/ICSE43902.2021.00043

[62] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proc. of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 826–837. https://doi.org/10.1145/3368089.3409720