Deep Learning Library Testing via Effective Model Generation

Zan Wang
[1]College of Intelligence and Computing, Tianjin University
[2]State Key Laboratory of Communication Content Cognition
China
wangzan@tju.edu.cn

Ming Yan
College of Intelligence and Computing, Tianjin University
China
yanming@tju.edu.cn

Junjie Chen*
College of Intelligence and Computing, Tianjin University
China
junjiechen@tju.edu.cn

Shuang Liu
College of Intelligence and Computing, Tianjin University
China
shuang.liu@tju.edu.cn

Dongdi Zhang
College of Intelligence and Computing, Tianjin University
China
zhangdongdi@tju.edu.cn

ABSTRACT

Deep learning (DL) techniques are rapidly developed and have been widely adopted in practice. However, similar to traditional software, DL systems also contain bugs, which could cause serious impacts especially in safety-critical domains. Recently, much research has focused on testing DL models, while little attention has been paid for testing DL libraries, which is the basis of building DL models and directly affects the behavior of DL systems. In this work, we propose a novel approach, LEMON, to testing DL libraries. In particular, we (1) design a series of mutation rules for DL models, with the purpose of exploring different invoking sequences of library code and hard-to-trigger behaviors; and (2) propose a heuristic strategy to guide the model generation process towards the direction of amplifying the inconsistent degrees of the inconsistencies between different DL libraries caused by bugs, so as to mitigate the impact of potential noise introduced by uncertain factors in DL libraries. We conducted an empirical study to evaluate the effectiveness of LEMON with 20 release versions of 4 widely-used DL libraries, i.e., TensorFlow, Theano, CNTK, MXNet. The results demonstrate that LEMON detected 24 new bugs in the latest release versions of these libraries, where 7 bugs have been confirmed and one bug has been fixed by developers. Besides, the results confirm that the heuristic strategy for model generation indeed effectively guides LEMON in amplifying the inconsistent degrees for bugs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Machine learning*.

*Junjie Chen is the corresponding author.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409761

KEYWORDS

Deep Learning Testing, Library Testing, Model Generation, Mutation, Search-based Software Testing

ACM Reference Format:
Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409761

1 INTRODUCTION

In recent years, deep learning (DL) techniques are rapidly developed and become one of the most popular techniques. Also, they are widely adopted in various domains in practice, such as autonomous driving cars [12], face recognition [59], speech recognition [29], aircraft collision avoidance systems [36], and software engineering [15, 16, 18, 21, 42, 70]. Unfortunately, DL systems are also shown to be vulnerable to attacks and lack of robustness [40, 67]. There are also reports of real-world accidents caused by DL systems, which threaten human lives. For example, an Uber autonomous driving car killed a pedestrian in Tempe, Arizona in 2018 [5], and there are reports on Tesla drivers being killed in autonomous piloting mode [6]. Therefore, it is extremely critical to properly test and verify DL systems before they are applied to safety-critical scenarios.

Compared with traditional software systems, DL systems usually involve more complex components, e.g., platform/hardware infrastructure, deep learning libraries, models, source programs for training, and training and testing corpus [30]. Each component may potentially introduce bugs into DL systems. Figure 1 shows the structure of a typical DL system, which consists of the application level, library level, and infrastructure level. Currently, most existing approaches focusing on guaranteeing the quality of DL systems are at the application level, e.g., testing DL models by generating adversarial inputs [48, 52, 66] or measuring the testing adequacy of DL models [45, 46, 55]. However, there is little attention on testing DL libraries, which may also contain bugs [32, 71] that directly affect the performance of DL models. Since DL libraries are the basis of constructing DL systems and the impact of bugs in DL

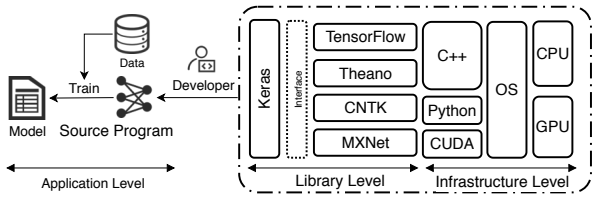

Figure 1: The structure of a typical DL system

libraries tends to be much larger than that in a specific DL model, it is very critical to explore the problem of testing DL libraries.

However, testing DL libraries is challenging. The difficulties are mainly two folds. First, it is difficult to obtain a large number of DL models to effectively trigger bugs in DL libraries. Different from traditional software systems, testing of DL libraries requires DL models, which stack many layers consisting of a large number of neurons and connection weights, as input. The DL models are constructed by the training process based on training data. However, due to the expensive training cost and limited available training data, it is quite difficult to construct a large number of models, let alone the models that can trigger library bugs. Second, it is challenging to expose the bugs triggered by the DL models. In traditional software systems, the test-oracle problem has been well studied and there are some ways of relieving the test-oracle problem [13, 17, 19, 22, 37, 49]. However, due to many uncertain factors in DL libraries, such as randomness and floating-point computing deviation [28], it is challenging to determine whether a found problem is a real bug in the library, or a problem caused by some uncertain factors.

To date, few approaches are proposed to target DL library testing. Pham et al. [56] proposed CRADLE, which is the state-of-the-art approach to detecting bugs in DL libraries. CRADLE utilizes available DL models as input to invoke different DL libraries, then differential testing is adopted to capture the triggered bugs. More specifically, it defines two metrics (to be presented in Section 2) to measure the inconsistent degree of prediction outputs and sets a threshold to distinguish real bugs and uncertain impacts. Exciting results are reported by CRADLE, but it still suffers from two major limitations: (1) Relying on existing models to trigger library bugs can be restricting. The public available models usually focus on popular tasks and only invoke a limited portion of library code, which tends to be well tested. Moreover, it is also tedious, and even unrealistic to collect a large number of available models that can trigger different portions of library code or explore different usage ways of library code. (2) Since the inconsistent degrees from real bugs may be similar to, or even smaller than, those from uncertain impacts, directly setting a threshold for the measured inconsistent degrees to distinguish them can be restricting. That is, before applying the threshold to distinguish them, a smart method is required to amplify the inconsistent behaviors caused by real bugs, and thus differentiate with normal diverse behaviors affected by uncertain factors more clearly.

In this paper, we propose a novel approach, called **LEMON** (deep learning **L**ibrary t**E**sting via guided **M**utati**ON**). LEMON is designed to solve the two challenges in DL library testing. To overcome the first challenge, we design a series of model mutation rules (including intact-layer mutation rules and inner-layer mutation rules) to automatically generate DL models by changing existing models. The mutation rules are designed with the purpose of exploring unused library code and different invoking sequences of library code. In this way, LEMON tries to maximize the ability to explore the code space of DL libraries for the generated models as much as possible. To overcome the second challenge, we propose a heuristic strategy in LEMON to guide the process of model generation towards the direction of amplifying inconsistent degrees for real bugs. In this way, LEMON increases the differentiation between real bugs and uncertain impacts. To summarise, the main contribution of LEMON is generating effective DL models to trigger and expose bugs in DL libraries.

To evaluate the effectiveness of LEMON, we conducted an extensive study based on 20 release versions of four widely-used DL libraries, i.e., TensorFlow [7], Theano [8], CNTK [1], and MXNet [4]. Also, we used 12 popular existing models based on 6 input sets as initial datasets for testing. In particular, LEMON detected 24 new bugs in the latest release versions of these libraries in total, where 7 bugs have been confirmed and one bug has been fixed by developers. The results also demonstrate that the generated models by LEMON detected many unique bugs that cannot be detected by the existing models, and significantly amplified the inconsistent degrees of the detected inconsistencies, i.e., the average amplified rates over the existing models range from 27.06% to 357.30% across different library versions. Furthermore, we investigated the contribution of our heuristic strategy for model generation by comparing with the random strategy, and the results confirm the effectiveness of our heuristic-based model generation.

To sum up, this work makes the following main contributions:

- A novel approach of DL library testing by generating effective DL models via guided mutation.
- A practical open-source tool implementing the proposed approach, including an individual component that conducts efficient mutations for DL models.
- An extensive study on 20 versions of four widely-used DL libraries demonstrating that LEMON detects 24 new bugs in the latest release versions of these libraries and the models generated by LEMON significantly outperform the existing models and the models generated without any guidance (i.e., the variant of LEMON).

2 BACKGROUND

Deep Learning Model. A DL model is composed of multiple layers, each of which consists of a large number of neurons. 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, such as convolution and pooling, for the input data with specific weights. In particular, the same layer can be adopted multiple times in a DL model, and the performance for the same kind of layer may be diverse as controlled by the weights on the links. Currently, there are two popular kinds of DL models, i.e., Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). CNN contains convolution computing and is often used to process data with grid-like topology, such as images. RNN uses loops to keep learned information and is mainly used to process sequential data, such as natural language.

Deep Learning Library. DL platforms generally provide high-level and low-level libraries. DL system developers implement their source programs by using high-level library APIs, which invoke the DL algorithms implemented in low-level libraries. Different low-level libraries are based on different infrastructures and have different input formats and APIs, while high-level libraries can hide the differences between low-level libraries and provide a consistent view on model construction and training.

Similar to traditional libraries, different DL low-level libraries provide different implementations according to the same algorithm/specification. Developers implement source programs by calling high-level libraries, which further invoke low-level libraries to finish different kinds of transformation operations and training process. One of the most popular high-level libraries is Keras [2], which has been widely used in various critical domains [11, 41, 61]. Keras runs on top of four low-level libraries, i.e., TensorFlow, CNTK, Theano, and MXNet, which cover most of the popular libraries. Similar to the existing work [56], we used TensorFlow, Theano, CNTK, and MXNet as the low-level libraries under test and adopted Keras as the high-level library invoking them. Therefore, DL library testing in our work refers to testing low-level libraries.

Metrics for Testing DL Libraries. Differential testing has been applied to test DL libraries [56]. There are various metrics proposed to measure the differences detected between DL libraries. We introduce those metrics briefly here. *D_CLASS* [56] is applied to classification models, which calculates a score for each prediction result by checking the rank of the ground-truth class in the prediction result and then calculates the difference between the two scores. In particular, it considers the rankings beyond Top-K (i.e., Top-5 in the study [56]) not interesting, i.e., setting the score to be 0. *D_MAD* [56] is applied to both classification and regression models and considers all elements in each output vector to calculate its distance with the ground-truth vector. Given the ground-truth vector denoted as $G = (g_1, g_2, \ldots, g_m)$, the prediction output vectors O_j and O_k, *D_MAD* is calculated based on Formulae 1 and 2. When the *D_CLASS* or *D_MAD* value is larger than a pre-defined threshold, an inconsistency is detected and the *D_CLASS* or *D_MAD* value marks the inconsistent degree of the inconsistency.

$$\delta_{O,G} = \frac{1}{m}\sum_{i=1}^{m}|o_i - g_i| \tag{1}$$

$$D\_MAD_{G,O_j,O_k} = \frac{|\delta_{O_j,G} - \delta_{O_k,G}|}{\delta_{O_j,G} + \delta_{O_k,G}} \tag{2}$$

$$R_A = \frac{\delta^A_{L_i,L_j} - \delta_{pre}}{\delta_{pre} + \epsilon} \tag{3}$$

Layer localization metric is also proposed (i.e., by [56]) to localize the root cause of the inconsistency. The formal definition is defined in Formula 3. $\delta^A_{L_i,L_j}$ represents the output difference (defined by Formula 1) of a layer A between L_i and L_j. δ_{pre} is the maximum output difference of the pre-layers of A. R_A represents the change degree between $\delta^A_{L_i,L_j}$ and δ_{pre}. In this formula, ϵ is set to 10^{-7}, which is used to avoid the division by zero problem when A is the first layer in M. The larger the value of R_A is, the larger the possibility that the layer A is the root cause of the inconsistency.

3 APPROACH

To effectively test DL libraries, there are two main challenges to be addressed. *The first challenge is to obtain a large set of DL models, which serve as test inputs for DL libraries, to trigger DL library bugs.* Different from traditional test inputs (e.g., numerical values and strings), a DL model is a structure stacking many layers, each of which contains a large number of neurons and connection weights, and thus traditional test generation tools cannot be used to generate DL models. It is non-trivial to obtain a large number of DL models due to the expensive training cost as well as the limited available training data. Moreover, it is difficult to generate models that can trigger DL library bugs. *The second challenge is that it is hard to expose the bugs triggered by DL models.* Although there are many test oracles to help determine whether a bug is detected in traditional software testing, it is difficult to distinguish whether it is a real bug for DL libraries since they involve many uncertain factors.

In this work, we propose a novel approach, called LEMON, to testing DL libraries via guided mutation. LEMON is designed to solve the above mentioned two challenges. To overcome the first challenge, we design a series of mutation rules to effectively and efficiently generate DL models by mutating existing models (presented in Section 3.1). To overcome the second challenge, we design a heuristic strategy to guide the process of DL model generation, so as to generate models that can amplify the inconsistency between different DL libraries as much as possible for real bugs (presented in Section 3.2). Finally, we introduce the test oracle used in LEMON in Section 3.3. Figure 2 shows the overview of LEMON.

3.1 Model Mutation

The goal of our mutation is to generate models to test DL libraries as sufficiently as possible by exploring unused library code or different usage ways of library code. To achieve this goal, we design a series of mutation rules at the model level. We propose to conduct model-level mutation rather than source-level mutation (i.e., mutating a source program used for training a model) due to two reasons: First, source-level mutation is more costly than model-level mutation since the former has to re-train a model after modifying the source program. In particular, the training process tends to take hours, even longer [24]. Second, model-level mutation can introduce more fine-grained changes to a model than source-level mutation [46].

A DL model consists of multiple layers, each of which contains a large number of neurons. Each layer is responsible to one specific functionality such as convolution and pooling. Therefore, we systematically design our model mutation rules in two types, i.e., intact-layer mutation and inner-layer mutation. We introduce them in detail in the following.

Intact-Layer Mutation. Intact-layer mutation involves mutations on the entire layer, and thus tends to introduce relatively large changes to a model. In total, we design seven intact-layer mutation rules, including four (i.e., LR, LC, LA, and AFRm) adopted from the existing work [46] and three newly proposed mutation rules (i.e., LS, MLA, and AFRp) according to our mutation goal. In particular, to explore unused library code, we design mutation rules (i.e., LA and MLA) to insert new layers to a model. In this way, the model could invoke new library code. Moreover, mutation rules to modify (i.e., LS, LC, and AFRp) or remove (i.e., LR, AFRm) existing layers

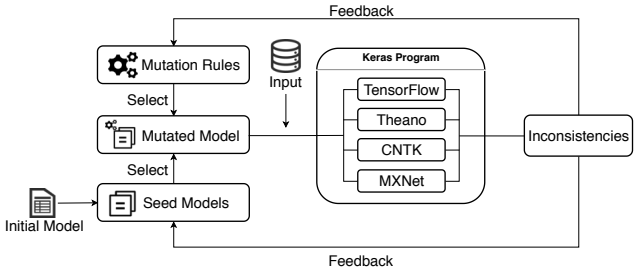

Figure 2: Overview of LEMON

are designed in order to explore different usage ways of library code in the model. Please note that, an explicit constraint for intact-layer mutation is that the output shape[1] of one layer and the input shape of another layer (to be concatenated) should be identical. The detailed intact-layer mutation rules are described in the following:

- *Layer Removal (LR)*: removes a layer, whose input shape and output shape are consistent, from the model.
- *Layer Switch (LS)*: switches two layers, both of which have the same input shape and output shape.
- *Layer Copy (LC)*: copies a layer, whose input shape and output shape are consistent, and then inserts the copied layer to concatenate the original layer.
- *Layer Addition (LA)*: selects a layer, whose input shape and output shape are consistent, from the universal set of layers, and then inserts it to a compatible position in the model.
- *Multi-Layers Addition (MLA)*: LA requires the input shape and output shape of the selected layer to be consistent. MLA gets rid of this constraint, and creates a bundle of layers by concatenating multiple selected layers, where the input shape of the first layer is consistent with the output shape of the last layer in the bundle. Finally, MLA inserts this bundle of layers to a compatible position in the model.
- *Activation Function Removal (AFRm)*: removes the activation function of a layer.
- *Activation Function Replace (AFRp)*: replaces the activation function of a layer with a randomly selected activation function from the universal set of activation functions.

Inner-Layer Mutation. Inner-layer mutation is operated on the neurons of a layer, which is more fine-grained than intact-layer mutation. The computation of a layer relies on its neurons, and thus changing the properties of neurons (e.g., weights and activation states) facilitates more sufficiently testing on the library code used in the layer and is also helpful to explore different usage ways of library code. In particular, we design five inner-layer mutation rules in total, which are adapted from the existing work [46]. Since a layer usually contains a large number of neurons and only changing one neuron has very slight impacts on the model, we randomly select 30% of neurons in the layer to apply inner-layer mutation.

- *Gaussian Fuzzing (GF)*: adds noise to the weights of a neuron following Gaussian distribution $N(\mu, \delta^2)$. If δ is large, the noise added to the weight is large, which is more likely to produce an invalid model. Therefore, we set μ to be 0 and δ to be 10% of the standard deviation of the weights for all the neurons in the layer.
- *Weights Shuffling (WS)*: shuffles the connection weights of a neuron with the previous layers.
- *Neuron Activation Inverse (NAI)*: inverts the activation state of a neuron by changing the sign of the output value of a neuron before passing it to the activation function.
- *Neuron Effect Block (NEB)*: eliminates the effects of a neuron on the next layers by setting the connection weights of the neuron to the next layers to be 0.
- *Neuron Switch (NS)*: switches two neurons in a layer to exchange their effects on the next layers.

First-Order and High-Order Mutation. To increase the diversity of the mutated models, LEMON considers both first-order mutation and high-order mutation. First-order mutation refers to applying only one mutation rule to the initial model, while high-order mutation refers to iteratively applying a series of mutation rules to the initial model. In other words, n^{th}-order mutation on the initial model is actually equivalent to first-order mutation on the model produced by $(n-1)^{th}$-order mutation on the initial model. As shown in Figure 3, the *NEB* mutation rule is applied to the middle *Dense* layer. 30% of neurons in the layer are blocked to cut off the connections with the next *Dense* layer (i.e., setting the weights to be 0), which is highlighted using the *red color* and *dashed lines* in Figure 3.

3.2 Heuristic-Based Model Generation

Since the mutation space is infinite, it is infeasible to generate all mutated models and then select a set of models with the largest inconsistent degrees from them to help expose bugs. Please note that in LEMON, we also adopt differential testing as the test oracle and use *D_MAD* in Formula 2 to calculate the inconsistent degree of an inconsistency, which is to be presented in Section 3.3 in detail. One of the most cost-effective solutions is that, for each generation, we generate a model that can produce larger inconsistent degrees of inconsistencies than before mutation as much as possible. That is, we should generate models towards the direction of amplifying the inconsistent degrees of inconsistencies. Based on this insight, we propose our heuristic-based model generation method. More specifically, in each iteration of our model generation, LEMON first selects a seed model to mutate (presented in Section 3.2.1) and then selects a mutation rule to apply (presented in Section 3.2.2).

3.2.1 Seed Model Selection. In our context, the initial seed model refers to a given existing model. LEMON generates mutated models from this seed model via first-order or high-order mutation. Since our model generation is conducted iteratively, the generated models in the previous iteration can also be used in the following iterations. In particular, to facilitate the model generation towards the direction of amplifying inconsistent degrees of inconsistencies, LEMON also treats the mutated models, which has larger inconsistent degrees than before mutation, as seed models. The metric defined by Formula 2 measures the inconsistent degree of an inconsistency produced by a model with an input on different low-level libraries. Let M_s and M_m be the seed model and the mutated model, respectively. Given a set of input $\{I_1, I_2, \ldots, I_n\}$ and a set of DL libraries $\{L_1, L_2, \ldots, L_m\}$, $ACC(M)=\sum_{i=1}^{n}\sum_{j,k=1}^{m} D\_MAD_{G,O_{ji},O_{ki}}(k > j)$

[1]The shape refers to the number of dimensions and the size of each dimension.

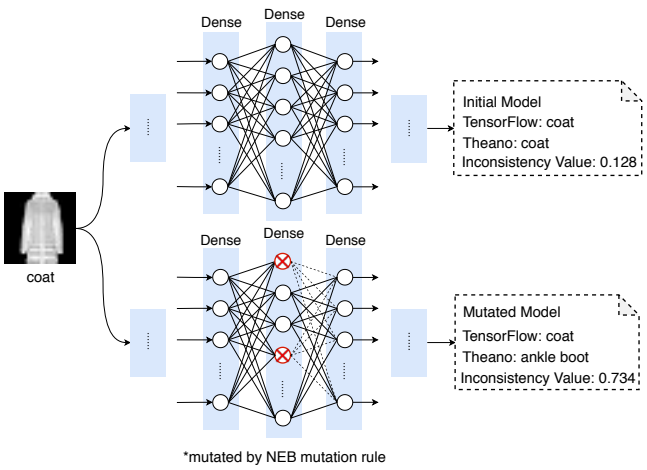

Figure 3: An inconsistency example detected by LEMON (it is a sketch map for layers, neurons, and connections)

indicates the accumulated inconsistent degrees of inconsistencies for all inputs under all DL library pairs for model M. If $ACC(M_m)$ is larger than $ACC(M_s)$, we regard the mutated model M_m as inconsistent-degree amplifying.

Intuitively, if a seed model is rarely selected to mutate before, we should give it a larger chance in order to increase the diversity of the generated models. For a seed model s_i, LEMON records the number of times that s_i has been selected to mutate, denoted as c_i, and then calculates a score for s_i, as shown in Formula 4. The larger the score of a seed model is, the higher the chance that the seed model is selected for next mutation is.

$$score_i = \frac{1}{c_i + 1} \tag{4}$$

According to the above intuition, we design a seed model selection procedure based on Roulette Wheel Selection [44]. More specifically, for a seed model s_i, LEMON calculates the probability that s_i is selected for next iteration among all the seed models based on its *score* value. The probability is defined in Formula 5, where r is the total number of seed models. Then, LEMON selects a seed model according to their calculated probabilities in each iteration.

$$p_i = \frac{score_i}{\sum_{k=1}^{r} score_k} \tag{5}$$

3.2.2 Mutation Rule Selection. Based on a selected seed model, LEMON then selects a mutation rule. However, different mutation rules may have different effectiveness in amplifying inconsistent degrees for a given seed model. Intuitively, if a mutation rule has frequently generated models that amplify inconsistent degrees of inconsistencies, it should be more likely selected for the following mutations. Therefore, for each mutation rule *MU*, LEMON calculates its priority score, denoted as *Ratio(MU)*, i.e., the number of times a model generated by *MU* amplifies inconsistent degrees of inconsistencies over the total number of times *MU* is selected for mutations. Then, mutation rules are ranked based on the descending order of their priority scores. However, it is not optimal to directly select the Top-1 mutation rule, since the ranking results are acquired based on historical iterations and cannot totally represent future results. Therefore, each mutation rule should have certain possibility to be selected, and in the meanwhile, the mutation rule ranked higher should be selected with a larger possibility.

Based on the above analysis, the mutation rule selection is actually affected by the most recent behavior of mutation rules, which makes it a typical Markov Chain (MC). Assuming the desired distribution to be equilibrium distribution [25], LEMON adopts Metropolis-Hastings (MH) algorithm [38], the most popular Markov Chain Monte Carlo (MCMC) method, to guide our mutation rule selection. More specifically, MH obtains random samples from a probability distribution, which refers to sampling the next mutation rule (denoted as MU_b) based on the current mutation rule (denoted as MU_a) according to the proposal distribution in our context.

Following the existing MCMC work [23], we set the geometric distribution as the probability distribution. It is the probability distribution of the number X of Bernoulli trials needed to obtain one success. If the success probability on each trial is p, the probability the k_{th} trial being the first success is $Ps(X = k) = (1-p)^{k-1}p$. Since mutation rules are selected randomly, the proposal distribution is symmetric. Therefore, the possibility of selecting MU_b given MU_a is calculated by Formula 6. In this formula, k_a and k_b are the rank of MU_a and MU_b in the ranking list of mutation rules. In particular, when $Ps(MU_b) > Ps(MU_a)$, $Pa(MU_b|MU_a) = 1$. Please note that, it is possible to select a mutation rule that cannot be successfully applied to the selected seed model, LEMON skips simply it.

$$Pa(MU_b|MU_a) = \frac{Ps(MU_b)}{Ps(MU_a)} = (1-p)^{k_b - k_a} \tag{6}$$

3.2.3 Overall Algorithm. We formally describe our heuristic-based model generation in Algorithm 1. The initial set of seed models only contains the given existing model and the initial priority score of each mutation rule is 0. In this algorithm, Line 1 randomly selects a mutation rule as the current one MU_a, Lines 2-30 iteratively generate a set of mutated models, and Line 31 outputs the final set of generated models. Lines 3-14 conduct the Roulette Wheel Selection process to select a seed model. Lines 15-20 selects the next mutation rule MU_b based on the current one MU_a. Lines 21-22 generate a new model m by applying MU_b to s and also add m into *Models*. Lines 23-25 judge whether m amplifies the accumulated inconsistent degrees for all inputs under all DL library pairs and updates the seed model set. Line 26 updates the score of s according to Formula 4. Lines 27-29 update the priority score of MU_b according to the ratio defined in Section 3.2.2, and then re-rank all the mutation rules for the next iteration.

3.3 Test Oracle in LEMON

Following the existing work [56], we also adopt differential testing as the test oracle to determine whether a bug in a DL library is detected. More specifically, LEMON adopts *D_MAD* (presented in Section 2) to measure the inconsistent degree between two prediction results. LEMON does not use *D_CLASS* since it aims to measure the differences on trained real models, i.e., models trained with real training data such that the ground-truth class tend to have a high rank in the prediction result. However, LEMON uses mutated models, which may produce low-ranked prediction results. Therefore, the *D_CLASS* value is likely to be 0 and thus not informative.

Algorithm 1: Heuristic-Based Model Generation

```
Input : Rules: A list of mutation rules
        Seeds: A list of seed models [M_s]
        N: The number of generated models, serving as the terminating condition
Output: Models: A set of generated models
MU_a ← random(Rules)
while Size(Models) < N do
    foreach i from 1 to Size(Seeds) do
        Pro[i]←calProb(Seeds[i])/* calculate the probability for Seeds[i] by Formula 5 */
    end
    r ← random(0,1)
    bound ← 0
    foreach i from 1 to Size(Seeds) do
        bound ← bound + Pro[i]
        if r ≤ bound then
            s ← Seeds[i]/* s is the selected seed model */
            break
        end
    end
    k_a ← position(MU_a)
    do
        MR_b ← random(Rules)
        k_b ← position(MR_b)
        f ← random(0,1)
    while f ≥ (1 − p)^(k_b − k_a);
    m ← Mutate(s, MU_b)
    Models ← Models ∪ {m}
    if ACC(m)≥ ACC(s) then
        /* ACC is defined in Section 3.2.1 to calculate the accumulated inconsistent degrees */
        Seeds ← Seeds ∪ {m}
    end
    updateScore(s)/* update the score of s defined in Formula 4 */
    updateRatio(MU_b)/* update the priority score of MU_b defined in Section 3.2.2 */
    Rules ← sort(Rules)
    MU_a ← MU_b
end
return Models
```

After acquiring the inconsistent degree between two prediction results, i.e., the *D_MAD* value, we determine whether a real inconsistency is detected. Following the existing work [56], if the *D_MAD* value exceeds a threshold T, we regard it as a real inconsistency; Otherwise, the difference is considered to be caused by uncertain factors in deep learning. In particular, since LEMON is designed to amplify the inconsistent degrees of inconsistencies, it would be clearer to distinguish real bugs and uncertain impacts. As shown in Figure 3, the inconsistent degree produced by the initial model under an input is only 0.128, while that produced by our mutated model under the same input reaches 0.734. Moreover, we manually checked the inconsistencies detected by the two models under the same input, and found that both of them are caused by the same buggy layer. That indicates that our model generation is indeed able to amplify the inconsistent degrees of inconsistencies and thus expose real bugs more effectively.

Besides the above discussed inconsistencies, there are still other two kinds of bugs. If the prediction results of some libraries are NaN (Not a Number) but those of other libraries are not, a bug is detected obviously. We call such bugs *NaN bugs*. If some models crash during execution but other models do not, a bug is also obviously detected. We call such bugs *Crash bugs*. To avoid confusion, we use the inconsistencies to represent the first kind of inconsistencies (excluding NaN and crash bugs) in this paper.

Table 1: Statistics information of DL libraries under test

ID	TensorFlow		Theano		CNTK		MXNET	
	Ver	#SLOC	Ver	#SLOC	Ver	#SLOC	Ver	#SLOC
E_1	1.14.0	2,261K	1.0.4	156K	2.7.0	331K	1.5.1	423K
E_2	1.13.1	1,970K	1.0.3	155K	2.6.0	328K	1.4.1	378K
E_3	1.12.0	1,874K	1.0.2	154K	2.5.1	320K	1.3.1	341K
E_4	1.11.0	1,829K	1.0.1	154K	2.4.0	313K	1.2.1	300K
E_5	1.10.0	1,779K	1.0.0	153K	2.3.1	304K	1.1.0	266K
Total	—	9,713K	—	773K	—	1,596K	—	1,708K

4 EVALUATION

In the study, we address the following research questions:

- **RQ1**: How does LEMON perform in detecting bugs in DL libraries?
- **RQ2**: Does our heuristic-based model generation contribute to LEMON?
- **RQ3**: How does LEMON perform in terms of efficiency?

4.1 Libraries and Datasets

Libraries. As presented in Section 2, in the study, we used four widely-used DL libraries, i.e., TensorFlow, CNTK, Theano, and MXNet, as subjects. To sufficiently evaluate the effectiveness of LEMON, we used 20 release versions of the four libraries in total. Based on the 20 versions, we constructed five experiments (indexed E_1 to E_5 in Table 1) to conduct differential testing. In this table, each row represents each differential testing experiment, and columns "Ver" and "#SLOC" present the library version and the corresponding number of source lines of code. In our study, the total SLOC of all studied libraries is up to about 14 million. In particular, the first experiment (i.e., ID is E_1) is based on the latest release versions of the four libraries. We used Keras (version 2.2.4) as the front end (i.e., the high-level library) to construct models. Due to limited space, we used TF, TH, CN, and MX to represent TensorFlow, Theano, CNTK, and MXNet in the following tables and figures.

Models and Datasets. To test these DL libraries, we used 12 popular DL models based on 6 popular input sets, as the initial seed models in LEMON, which have been widely used in many existing studies [43, 56]. In particular, we considered the diversity of the models and input sets adopted by considering the model structures (including both CNN and RNN models), the scales of models (including both large and small models in terms of the number of weights and layers in a model), and the domains of input sets (including both images and sequential data). Here, Sine-Wave and Stock-Price are sequential data, where the former is a set of *sine* function values and the latter is a set of Disneyland Stock Price data between 1997 and 2016. Table 2 shows their detailed information. For each model, we randomly sample 1, 500 inputs from the corresponding validation set, as the input data in our study. We can observe that the number of layers ranges from 3 to 159 and the number of weights ranges from 27K to more than 143 million, which indicates an extremely large mutation space. Please note that, we directly loaded weights from Keras for models trained with ImageNet and trained the other

Table 2: Statistics information of datasets

ID	Model	Input Set	#Weight	#Layer	Domain	Net.
1	AlexNet	CIFAR-10	1,251K	17	Image	CNN
2	LeNet5	Fashion-MNIST	413K	10	Image	CNN
3	LeNet5	MNIST	62K	13	Image	CNN
4	LSTM-1	Sine-Wave	71K	5	Sequential	RNN
5	LSTM-2	Stock-Price	27K	3	Sequential	RNN
6	ResNet50	ImageNet	25,637K	50	Image	CNN
7	MobileNetV1	ImageNet	4,254K	88	Image	CNN
8	InceptionV3	ImageNet	23,852K	159	Image	CNN
9	DenseNet121	ImageNet	8,063K	121	Image	CNN
10	VGG16	ImageNet	138,358K	23	Image	CNN
11	VGG19	ImageNet	143,667K	26	Image	CNN
12	Xception	ImageNet	22,910K	126	Image	CNN

* LSTM-1 and LSTM-2 are two different LSTM-based models trained with sequential data.

models using their provided source programs and corresponding training sets, respectively.

4.2 Measurements

Number of Inconsistencies. We denote an inconsistency between two libraries L_i and L_j produced by a model M under an input I as a triple $(M, I, L_i \leftrightarrow L_j)$. Since we aim at testing the low-level libraries, the input triggering the library executions involves both M and I (to M). Therefore, we use a tuple (M, I) to denote a *library-invoking input*. Since the mutation rules tend not to change a model behavior in a large manner, we assume that the inconsistencies exposed by the *library-invoking inputs* $(M_1, I), (M_2, I), \ldots, (M_n, I)$, where $M_1, M_2, \ldots, M_n$ are mutants from the same initial model, on the library pair (L_i, L_j) are likely to reflect the behavior difference between L_i and L_j in the same way. Therefore, we keep only one inconsistency among them and use the inconsistency with the largest inconsistent degree as the representative, which also avoids the influence of equivalent mutants. In this way, we keep the inconsistencies that can reflect library behavior differences in different ways to a large extent. We count the number of these inconsistencies as a measurement in our study. Similar to spectrum-based fault localization (SBFL) [64] and automated program repair (APR) [51], more failure-triggering tests (referring to library-invoking inputs that trigger inconsistencies in our work) reflecting a fault in different ways are more helpful to increase the suspicious score of the root-cause program element in SBFL and filter plausible patches in APR. Therefore, the number of inconsistencies is able to measure the effectiveness of LEMON to some degree. Larger is better.

Number of Detected Bugs. Although we acquire the number of inconsistencies, it is more important to acquire the number of unique bugs detected by LEMON from these inconsistencies. According to the voting mechanism in differential testing [49], the buggy library for each inconsistency can be identified. Then, for each inconsistency, we used the localization method presented in Section 2 to locate which layer is the most suspicious one to cause the inconsistency. Since the localization method has been demonstrated to be very effective in the existing work [56], we use the localized Top-1 layer as the root cause. Here, we use a tuple (the voted buggy library, the localized buggy layer) to denote a unique bug. Since the measurements (including the localization method and the threshold for identifying an inconsistency) may not be completely precise, we further manually check each bug by building a one-layer model that keeps the same parameters of the identified layer to check the result under the same layer input for different libraries. Actually, the rate of false positives for LEMON is very small, i.e., less than 10%. In our experiment, we count the number of bugs after manual analysis to measure the effectiveness of LEMON.

4.3 Compared Approaches

The state-of-the-art approach to testing DL libraries is CRADLE, which is proposed by Pham et al. [56]. The main contribution of CRADLE is the test oracle, i.e., CRADLE adopts *differential testing* to detect inconsistencies based on existing models. LEMON, on the other hand, focuses on model generation, i.e., LEMON proposes to generate effective models via guided mutation to trigger and expose inconsistencies to a large extent. The contributions of the two approaches are actually orthogonal. Since the number of generated models can be very large and collecting the same number of existing models is scarcely possible, it is hard to directly compare LEMON with CRADLE. Alternatively, for each given existing model (i.e., initial model), we analyzed how many inconsistencies/bugs detected by LEMON are only detected by the initial model and how many inconsistencies/bugs detected by LEMON are only detected by the models mutated from the initial model.

Besides, our heuristic-based model generation is the core of LEMON, and thus it is also interesting to investigate the effectiveness of this component. We compared LEMON with its variant that replaces the heuristic-based model generation with the random model generation. More specifically, the random model generation is to generate models via mutation without any guidance, i.e., randomly selecting a seed model and a mutation rule in each iteration. We call this invariant **LEMON$_r$**.

4.4 Implementations and Data Availability

We set the threshold T to be 0.4, indicating that an inconsistency is regarded as detected when the value of D_MAD exceeds 0.4. This setting is relatively large so as to avoid introducing too many false positives, which is also confirmed by our manual analysis. For the terminating condition of LEMON, when the number of mutated models reaches 100 for a given initial model, we terminate LEMON. Following the setting of p in the MH algorithm [14, 23], we set p to be 0.08. Our study is conducted on the Intel Xeon E5-2640 machine with 128GB RAM, Ubuntu 16.04.6 LTS, and two GTX 1080 Ti GPUs.

Our tool and experimental data are publicly available at our project website [3].

5 RESULTS AND ANALYSIS

5.1 Effectiveness of LEMON

5.1.1 New Bugs Detected by LEMON. We first investigated the effectiveness of LEMON in terms of new bugs detected in the latest release versions of libraries, i.e., versions used in E_1, whose results are shown in Table 3. In total, LEMON detects 24 new bugs in the latest release versions of these libraries, including 13 bugs by analyzing the detected inconsistencies, 6 crash bugs, 4 NaN bugs, and 1 performance bug. More specifically, there are 5 TensorFlow bugs, 4 Theano bugs, 2 CNTK bugs, 12 MXNet bugs, and surprisingly 1 Keras bug (the used front-end in our study), indicating that LEMON is able to detect bugs for all the studied libraries. In particular, 7

Table 3: The number of new bugs detected by LEMON

Library	#IC Bugs	#Crash	#NaN	#Total
TensorFlow	4	0	1	5
Theano	3	0	1	4
CNTK	2	0	0	2
MXNet	4	6	2	12
Keras	1 performance bug			24

* IC is short for inconsistency and IC Bugs refer to the bugs analyzed from inconsistencies. The last cell refers to the total number of new bugs detected by LEMON on all the five libraries.

```
+   mxnet::TShape& out_shp = (*out_attrs)[0];
    CHECK_LE(shp.ndim(), 6) << "Transpose support at most 6 dimensions";
-   mxnet::TShape ret(shp.ndim(), -1);
.....
-   CHECK_EQ(shp.ndim(), param.axes.ndim());
+   CHECK_EQ(std::max(shp.ndim(), out_shp.ndim()), param.axes.ndim());
```

Figure 4: The fix of the buggy transpose operator in MXNet

bugs have been confirmed in the corresponding issue repositories and one bug has been fixed by developers. The average number of generated mutants is 24.57 for each detected bug. More specifically, there are five bugs that are detected by generating less than 3 mutants while there are also three bugs that are detected by generating more than 70 mutants. The average time spent on generating and running these mutants for each detected bug is 2.62 hours. We then conducted case analysis according to bug types:

Crash Bugs. We regarded the crash bugs with different crash messages as different crash bugs. One crash bug has been fixed by developers, which occurs in MXNet. This bug cannot be detected by any initial models but is detected by the mutated models from six initial models (i.e., AlexNet, DenseNet121, LeNet5$_F$, LeNet5$_M$, MobilenetV1, and Xception), showing a large-scale influence. More specifically, the crash bug is due to the wrong shape inference of the transpose operator in MXNet, whose fix is shown in Figure 4. The buggy transpose operator only relies on the input tensor for shape inference, causing to throw the exception message: *"Check failed: shp.ndim()==param.axes.ndim() (-1 vs 4)"*, even if the model is valid. The fixed transpose operator uses both the input tensor and output tensor and thus is able to infer all unknown dimension sizes based on these tensors.

Bugs from Inconsistencies. The 13 bugs from inconsistencies involve different types of layers, including LSTM, Conv2D, BatchNormalization, Dense, DepthwiseConv2D, MaxPooling2D, AveragePooling2D, in different libraries. By taking a Theano bug as an example, for a mutated model from *LeNet5$_F$* this bug leads to the accuracy of the mutated model using Theano to be 44.66% while the accuracy of the mutated model using the other three libraries is larger than 92.35%, indicating the significant influence of this bug, confirmed by developers. We find that the *R* value of the *Conv2D* layer is 76399.15 between Theano and TensorFlow while the maximum *R* value of all the other layers is only 5.47, indicating that *Conv2D* is the localized root cause.

By taking a TensorFlow bug as another example, among 1500 input images for *Xception*, Theano and CNTK have the same prediction results but TensorFlow has different results for 118 images. The root cause of this bug is identified as the *BatchNormalization* layer, i.e., wrong values of the variables `moving_mean` and `moving_var`. Another user also replies to our bug report to complain that they suffered from this bug for transfer learning.

NaN Bugs. As shown in Table 3, LEMON detects 4 NaN bugs in three libraries. By taking a Theano NaN bug as an example, for a mutated model from *MobileNetV1*, the output using Theano is NaN while that using CNTK is normal, confirmed by developers. In particular, for the initial model *MobileNetV1* the outputs of both Theano and CNTK are normal. That demonstrates that our mutation rules are effective to make the calculation process trigger NaN bugs. Further, we analyzed the output of each layer of this mutated model using Theano, and find that at the 17^{th} layer (a *BatchNormalization* layer), its output starts to be NaN. Moreover, the outputs of the first 16 layers for the model using Theano and CNTK are the same. That indicates that the root cause of this NaN bug lies in the *BatchNormalization* layer of Theano.

We also find an interesting NaN bug in MXNet. For a mutated model from *MobileNetV1*, the output using MXNet is NaN while that using CNTK is normal, and in the meanwhile both of them have normal outputs for the initial model *MobileNetV1*. We then analyzed the output of each layer of the mutated model using MXNet, but surprisingly no NaN happens. Actually, directly obtaining the prediction result (used in LEMON) and obtaining the prediction result by getting the output of each layer are two equivalent ways but uses different Keras interfaces. The former calls the interface `predict` while the latter calls the output attribute of each layer. Moreover, the former outputs NaN while the output of the latter is normal. We infer that the root cause of this NaN bug is in the interface (i.e., `predict`) between Keras and MXNet, and we have submitted a bug report for it and are waiting for responses.

Performance Bug. LEMON also detected an interesting performance bug in Keras. When conducting model mutations in LEMON, LEMON has to repeatedly call the Keras functions `clone_model` and `set_weights`, but the time spent on each single call becomes longer. `clone_model` and `set_weights` take about 4 seconds and 3 seconds in the first iteration, and the time grows to about 34 seconds and 4 minutes, respectively, in the 50^{th} iteration. By a detailed observation, we find that the bug is caused by memory leak, and this performance bug is detected during the mutation process of LEMON (thus only using initial models cannot detect this bug), which has also been confirmed.

5.1.2 Mutated Models v.s. Initial Models. One major contribution of LEMON is generating mutated models, and thus it is interesting to investigate the unique value of the mutated models. We present the detailed results in Table 4. Here, we investigated the effectiveness of LEMON and the unique value of the mutated models for each experiment of differential testing, each used model, and each studied library, respectively. For example, the first number of the third row in Table 4 represents the number of TensorFlow bugs detected by LEMON using the model AlexNet in experiment setting E_1. From Column "Total" in Table 4, we find that LEMON is able to detect bugs in each library at each differential-testing experiment, and on average the number of detected library bugs by LEMON is 22 for the five experiments, which demonstrates the effectiveness of LEMON.

Table 4: Detailed comparison results in terms of the number of detected bugs

ID	Lib	AlexNet			DenseNet			Inception			$LeNet5_F$			$LeNet5_M$			LSTM-1			LSTM-2			MobileNet			ResNet5			VGG16			VGG19			Xception			Total		
		L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I	L	M	I
E_1	TF	1	0	0	0	0	0	0	0	0	1	0	0	2	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	5	2	0
	TH	1	0	0	0	0	0	0	0	0	1	0	0	2	2	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	4	2	0
	CN	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0
	MX	3	2	0	4	3	0	2	1	0	4	3	0	3	3	0	2	1	0	1	0	0	4	3	0	4	2	0	1	1	0	2	2	0	3	3	0	12	7	0
E_2	TF	3	2	1	1	0	0	0	0	0	1	0	0	2	1	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	6	2	0
	TH	1	0	0	0	0	0	0	0	0	2	1	0	2	2	0	0	0	0	0	0	0	1	1	0	2	2	0	0	0	0	1	1	0	0	0	0	6	4	0
	CN	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	0
	MX	1	0	0	2	1	0	1	0	0	2	1	0	2	2	0	1	0	0	1	0	0	2	1	0	4	2	0	1	1	0	1	1	0	2	2	0	10	5	0
E_3	TF	3	2	0	2	1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	4	0	0
	TH	1	0	0	1	1	0	0	0	0	2	1	0	1	1	0	0	0	0	0	0	0	1	1	0	1	1	0	1	1	0	1	1	0	0	0	0	5	3	0
	CN	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	0
	MX	1	0	0	2	1	0	1	0	0	2	1	0	1	1	0	1	0	0	1	0	0	1	0	0	3	1	0	1	1	0	1	1	0	1	1	0	8	3	0
E_4	TF	3	2	0	1	0	0	1	1	0	2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0	4	1	0
	TH	2	1	0	0	0	0	1	1	0	1	0	0	2	2	0	0	0	0	0	0	0	1	1	0	1	1	0	1	1	0	0	0	0	0	0	0	4	2	0
	CN	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0
	MX	2	1	0	3	2	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	2	1	0	3	1	0	1	1	0	0	0	0	1	1	0	8	4	0
E_5	TF	2	1	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1	1	0	0	0	0	1	0	0	4	1	0			
	TH	2	1	0	0	0	0	0	0	0	1	0	0	2	2	0	0	0	0	0	0	0	3	3	0	1	1	0	1	1	0	0	0	0	1	1	0	6	4	0
	CN	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	3	2	0
	MX	1	0	0	3	2	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	2	1	0	2	0	0	1	1	0	0	0	0	3	3	0	9	5	0

* Column "L" presents the number of detected bugs by LEMON, Column "M" presents the number of unique bugs that are detected by the mutated models but are not detected by the initial model, and Column "I" presents the number of unique bugs that are detected by the initial model but are not detected by the mutated models. $LeNet5_F$ refers to the LeNet5 model based on the Fashion-MNIST input set, $LeNet5_M$ refers to the LeNet5 model based on the MNIST input set, DenseNet refers to DenseNet121, MobileNet refers to MobileNetV1, and Inception refers to InceptionV3.

Table 5: Average inconsistent degree comparison for the detected inconsistencies

Lib Pair	E1			E2			E3			E4			E5		
	V_M	V_I	$\Uparrow_{rate}$	V_M	V_I	$\Uparrow_{rate}$	V_M	V_I	$\Uparrow_{rate}$	V_M	V_I	$\Uparrow_{rate}$	V_M	V_I	$\Uparrow_{rate}$
TF↔TH	0.83	0.31	166.45%	0.60	0.13	348.90%	0.64	0.14	357.30%	0.60	0.19	207.76%	0.59	0.15	282.14%
TF↔CN	0.70	0.34	104.71%	0.62	0.20	213.44%	0.59	0.23	160.12%	0.61	0.25	141.33%	0.60	0.24	152.47%
TH↔CN	0.84	0.39	118.30%	0.69	0.26	162.04%	0.74	0.29	152.77%	0.76	0.31	142.99%	0.74	0.33	123.45%
TF↔MX	0.85	0.55	54.08%	0.82	0.50	62.72%	0.76	0.52	47.84%	0.75	0.49	55.57%	0.80	0.49	64.12%
TH↔MX	0.92	0.71	29.16%	0.79	0.44	81.10%	0.90	0.65	37.15%	0.94	0.74	27.06%	0.81	0.57	42.11%
CN↔MX	0.83	0.45	85.06%	0.81	0.38	112.94%	0.77	0.44	72.51%	0.82	0.54	51.84%	0.80	0.46	72.94%

* Columns "V_M" and "V_I" present the average inconsistent degrees achieved by the mutated models and the initial models across all the used models, respectively. Column "$\Uparrow_{rate}$" presents the average improved rates of inconsistent degrees achieved by the mutated models over the initial models.

Please note that the bugs detected at different differential-testing experiments could be duplicate since some detected bugs have been hidden for a long time.

Besides, among all the 60 cases (12 models * 5 differential-testing experiments), the mutated models are able to detect at least one unique bug in 76.67% (46 out of 60) cases (shown in Columns "M"), while the initial models detect only one unique bug in only one case (shown in Columns "I") , which is also detected by the mutated models in other cases. Therefore, the results demonstrate the unique value of the mutated models, and the bugs detected by the initial models are a subset of the bugs detected by the mutated models.

Overall, from the detailed results we conclude that regardless of which initial models are used as seed models of LEMON, LEMON does detect a number of bugs including a large proportion of unique bugs, largely augmenting the testing capability of the existing models, which are the models used in CRADLE.

5.1.3 Comparison of Inconsistent Degrees. To investigate the reason for the superiority of the mutated models, we further evaluated whether the mutated models can amplify the inconsistent degrees compared with the initial models, which is an objective of our heuristic strategy. We compared the inconsistent degrees for the detected inconsistencies for this purpose. If an inconsistency is detected by either an initial model or the corresponding mutated models using the same input and library pair, we compared the inconsistent degrees of the inconsistency between them. Table 5 shows the average results across all the used models for each library pair and each differential-testing experiment. From this table, for all the library pairs and all the differential-testing experiments, LEMON indeed significantly amplifies the inconsistent degrees of the inconsistencies, successfully achieving its objective. In particular, the average improved rates of inconsistent degrees achieved by the mutated models over initial models range from 27.06% to 357.30% for different library pairs and differential-testing experiments. The results demonstrate that LEMON is able to effectively amplify the inconsistent degrees by generating mutated models from initial models, which contributes to expose the inconsistencies/bugs more effectively.

5.2 LEMON v.s. $LEMON_r$

The heuristic-based model generation method is one of the main contributions of LEMON, and thus we further investigated the effectiveness of the heuristic-based model generation method compared

Table 6: Comparison between LEMON and LEMON$_r$

Lib Pair	# Only mutated		# Only initial		$\Uparrow_{rate}$	
	LEMON	LEMON$_r$	LEMON	LEMON$_r$	LEMON	LEMON$_r$
TF↔TH	136	108	0	0	49.21%	28.83%
TF↔CN	181	133	0	0	69.79%	35.45%
TH↔CN	114	123	0	0	81.29%	57.16%
TF↔MX	371	269	5	55	27.88%	14.79%
TH↔MX	317	231	5	55	35.89%	27.02%
CN↔MX	382	319	5	55	54.17%	41.34%

* Columns 2-3 present the total number of inconsistency only detected by the mutated models generated by LEMON or LEMON$_r$ (not detected by initial models) across all the models. Columns 4-5 present the total number of inconsistency only detected by the initial models (not detected by mutated models). Column 6-7 present the average improved rates of inconsistent degrees achieved by the mutated models over initial models.

with the random generation method LEMON$_r$. We adopted the library versions used in E_2 as the representative. More specifically, for each initial model we ran LEMON and LEMON$_r$ in the same time period (i.e., one hour) to test the four libraries for fair comparison. We repeated the process five times and calculated the average results to reduce the impact of randomness. Due to the cost of identifying the root-cause layer, we compared LEMON and LEMON$_r$ in terms of the number of detected inconsistencies. In particular, the larger number of detected inconsistencies tends to mean the larger number of detected bugs, which have been demonstrated based on the results in Section 5.1.

Table 6 presents the comparison results between LEMON and LEMON$_r$. From this table, we find that compared with LEMON$_r$, the models generated by LEMON detect more unique inconsistencies which are missed by the initial models, on almost all the library pairs. The total number of inconsistencies that are only detected by the initial models and missed by LEMON is much smaller than that missed by LEMON$_r$. Moreover, the average improved rate of inconsistent degrees achieved by LEMON is always larger than that of LEMON$_r$ for each library pair. The results demonstrate that both LEMON and LEMON$_r$ are able to detect many unique inconsistencies that cannot be detected by the given initial models. LEMON detects more than LEMON$_r$. LEMON$_r$ also misses many inconsistencies detected by the initial models. We also find that LEMON achieves larger improved rates of inconsistent degrees over initial models than LEMON$_r$. The reason is that LEMON$_r$ does not guide the generation of models towards the direction of amplifying inconsistent degrees, and thus it is more likely to diminish inconsistent degrees than LEMON. The evaluation results confirm the contribution of the heuristic-based model generation method in LEMON.

5.3 Efficiency of LEMON

We also analyzed the efficiency of LEMON. Due to the design of our mutation rules (carefully considering the input shape and output shape), LEMON does not generate invalid models. On average, LEMON spent 4.97 minutes at each iteration for each given initial model. More specifically, the average time spent on generating a mutated model by LEMON is only 0.3 minutes and the average time spent on running all the inputs for a model is 4.67 minutes. The results demonstrate the efficiency of LEMON, which facilitates it to be applied in practice.

6 DISCUSSION

6.1 Extensions of LEMON

LEMON can be potentially extended on three aspects.

First, the current LEMON cannot test the library code used for model training since it uses existing pre-trained models as seed models and conducts mutations at the model level without retraining. In the future, LEMON can be extended to consider mutation rules at the source level, which involve the model retraining process, and conducts differential testing in the same way.

Second, LEMON adopts differential testing to solve the test-oracle problem in DL library testing, but it could miss bugs if different libraries produce the same wrong results. To get rid of this limitation, it is promising to introduce metamorphic testing for LEMON to help solve the test-oracle problem, since metamorphic testing does not require different libraries and detects bugs based on the properties of one library.

Third, DL library testing relies on both models and the input data of the models. If an effective model does not have proper input data, the testing capability of the model cannot be fully manifested. Currently, LEMON aims to generate effective models to detect library bugs, and does not consider the impact of the input data. In the future, we will further explore what kind of input data is helpful to show the testing capability of a specific model generated by LEMON as sufficiently as possible.

6.2 Threats to Validity

The *internal* threat to validity mainly lies in the implementations of LEMON and our scripts in the study. To reduce this threat, two authors have carefully checked the code before submission.

The *external* threats to validity mainly lie in the used libraries and models in our study. We adopted four widely-used libraries, i.e., TensorFlow, Theano, CNTK, and MXNet, as subjects, but they may not represent other libraries such as PyTorch. We chose the four libraries, since they are supported by the same front-end Keras while PyTorch is not, and the current implementation of the mutation rules in LEMON only supports the models using the Keras front-end. However, LEMON is a general approach, and it can be used to test PyTorch by just re-implementing the mutation rules in LEMON to support the models using PyTorch, which is also our future work. To reduce this threat, we used 20 different release versions of the four libraries in total. We also try to use a diverse range of model families (12 popular models based on 6 input sets), which have different model structures, in order to trigger more library behaviors.

The *construct* threats to validity mainly lie in randomness, settings, and measurements in our experiment. To reduce the impact of randomness in our study, we constructed five differential-testing experiments instead of repeating each experiment several times. When comparing LEMON and LEMON$_r$, we set the same time limit (i.e., one hour for each initial model) using only one experiment, and thus we repeated each approach five times and calculated the average results to reduce the impact of randomness. The threats from the settings (e.g., the threshold T) and measurements, have been discussed in Sections 4.4 and 4.2. To further reduce them, we reported bugs to the corresponding bug repositories, and some have been confirmed/fixed and some are still waiting for responses.

7 RELATED WORK

Deep Learning Testing. The most related work to ours is CRADLE proposed by Pham et al. [56], which has been discussed before. Besides, Zhang et al. [71] and Islam et al. [32] conducted *empirical studies* to investigate the characteristics of DL source program bugs. Different from them, our work proposes a novel approach to testing DL libraries via effective model generation. Besides, there are some work focusing on testing *machine learning (ML)* libraries [26, 27, 57, 58]. For example, Dwarakanath et al. [27] adopted metamorphic testing to test ML libraries by conducting transformations on training and testing data. Different from them, our work focuses on testing *DL* libraries and proposes to generate effective DL models.

Furthermore, there are a great deal of researches focusing on testing DL models in the literature [10, 20, 31, 45–47, 55, 62, 65, 66, 69]. Many of them proposed criteria to measure test adequacy [39, 45, 46, 55]. For example, Ma et al. [45] proposed a set of multi-granularity testing criteria, including neuron-level and layer-level coverage criteria, for DL models. Besides, many of them proposed to find/generate adversarial inputs [52, 60, 66]. For example, Xie et al. [66] proposed DeepHunter, a fuzz testing framework to test DL models, which conducts semantic-preserving transformation for input of the models under test and uses coverage criteria to guide the fuzzing process to find issues in models. Different from them, our work focuses on testing DL libraries rather than DL models.

Mutation Testing. Our work is also related to mutation testing, which is one of the most effective methods to measure test-suite quality. Mutation testing has been extensively studied in traditional software systems [9, 33–35, 50, 53, 54, 68]. Recently, in the area of DL testing, Ma et al. [46] proposed a mutation testing framework for DL models. Wang et al. [63] proposed to detect adversarial inputs for DL models by mutating models, based on the observation that adversarial inputs are more sensitive than normal inputs in mutated models. Different from them, our work aims to test DL libraries by generating effective models via mutating existing models. That is, the mutated models serve as input of DL libraries rather than the subjects under test.

8 CONCLUSION

In this paper, we propose a novel approach, LEMON, to testing DL libraries by generating effective DL models via guided mutation. More specifically, we design a series of model mutation rules in LEMON to generate DL models by changing existing models. The design goal is to test DL libraries as sufficiently as possible by exploring unused library code or different usage ways of library code. We further propose a heuristic strategy in LEMON to guide the process of model generation so as to generate models that can amplify the inconsistent degrees for real bugs. In this way, it is clearer to distinguish real bugs and uncertain impacts in DL libraries. We conducted an empirical study to evaluate the effectiveness of LEMON based on 20 release versions of TensorFlow, Theano, CNTK, and MXNet. LEMON detected 24 new bugs in the latest release versions of these libraries. The results also demonstrate that the models generated by LEMON outperform existing models and the models generated without guidance in terms of the number of unique bugs/inconsistencies and the achieved inconsistent degrees.

ACKNOWLEDGEMENT

This work has been supported by the National Natural Science Foundation of China 62002256, 61872263, U1836214, 61802275 and Intelligent Manufacturing Special Fund of Tianjin 20191012, 20193155.

REFERENCES

[1] Accessed: 2020. CNTK. https://docs.microsoft.com/cognitive-toolkit/.
[2] Accessed: 2020. Keras. http://keras.io.
[3] Accessed: 2020. LEMON. https://github.com/Jacob-yen/LEMON.
[4] Accessed: 2020. MXNet. http://mxnet.incubator.apache.org.
[5] Accessed: 2020. News. https://www.vice.com/en_us/article/9kga85/uber-is-giving-up-on-self-driving-cars-in-california-after-deadly-crash.
[6] Accessed: 2020. News. https://en.wikipedia.org/wiki/List_of_self-driving_car_fatalities#cite_note-15.
[7] Accessed: 2020. TensorFlow. https://www.tensorflow.org.
[8] Accessed: 2020. Theano. http://deeplearning.net/software/theano/.
[9] Dennis Appelt, Cu Duy Nguyen, Lionel C Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *2014 International Symposium on Software Testing and Analysis*. 259–269.
[10] Houssem Ben Braiek and Foutse Khomh. 2019. DeepEvolution: A Search-Based Testing Approach for Deep Neural Networks. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 454–458.
[11] Ken Chang, Niranjan Balachandar, Carson Lam, Darvin Yi, James Brown, Andrew Beers, Bruce Rosen, Daniel L Rubin, and Jayashree Kalpathy-Cramer. 2018. Distributed deep learning networks among institutions for medical imaging. *Journal of the American Medical Informatics Association* 25, 8 (2018), 945–954.
[12] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.
[13] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *Proceedings of the 31st International Conference on Automated Software Engineering*. 178–189.
[14] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.
[15] Junjie Chen, Xiaoting He, Qingwei Lin, Yong Xu, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. An Empirical Investigation of Incident Triage for Online Service Systems. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*. 111–120.
[16] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 364–375.
[17] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*. 180–190.
[18] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*. to appear.
[19] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53 (02 2020), 1–36.
[20] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical Accuracy Estimation for Efficient Deep Neural Network Testing. *ACM Transactions on Software Engineering and Methodology* (2020). to appear.
[21] 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 *The 35th IEEE/ACM International Conference on Automated Software Engineering*. to appear.
[22] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1 (2018), 4:1–4:27.
[23] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. In *PLDI*. 85–99.
[24] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. 1223–1231.
[25] Yadolah Dodge. 2006. *The Oxford dictionary of statistical terms*. Oxford University Press.
[26] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 ACM Joint Meeting*

on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 574–586.

[27] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 118–128.

[28] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *Comput. Surveys* 23, 1 (1991), 5–48.

[29] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*. 6645–6649.

[30] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study towards Characterizing Deep Learning Development and Deployment across Different Frameworks and Platforms. In *Proceedings of the 34th International Conference on Automated Software Engineering*. to appear.

[31] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel Briand. 2019. Comparing Offline and Online Testing of Deep Neural Networks: An Autonomous Car Case Study. *arXiv preprint arXiv:1912.00805* (2019).

[32] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. *arXiv preprint arXiv:1906.01388* (2019).

[33] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 208–219.

[34] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.

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

[36] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. 1–10.

[37] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 590–600.

[38] Robert E. Kass, Bradley P. Carlin, Andrew Gelman, and Radford M. Neal. 1998. Markov Chain Monte Carlo in Practice: A Roundtable Discussion. *American Statistician* 52, 2 (1998), 93–100.

[39] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering*. 1039–1049.

[40] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533* (2016).

[41] Sunyoung Kwon and Sungroh Yoon. 2017. Deepcci: End-to-end deep learning for chemical-chemical interaction prediction. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*. 203–212.

[42] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th International Symposium on Software Testing and Analysis*. 169–180.

[43] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. 2019. Boosting Operational DNN Testing Efficiency Through Conditioning. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 499–509.

[44] Adam Lipowski and Dorota Lipowska. 2012. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications* 391, 6 (2012), 2193–2196.

[45] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[46] 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 *29th IEEE International Symposium on Software Reliability Engineering*. 100–111.

[47] 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 *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.

[48] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. 2019. NIC: Detecting Adversarial Samples with Neural Network Invariant Checking. In *26th Annual Network and Distributed System Security Symposium*.

[49] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[50] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2014. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering* 41, 5 (2014), 429–444.

[51] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24.

[52] Augustus Odena and Ian Goodfellow. 2018. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875* (2018).

[53] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 354–365.

[54] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. 275–378.

[55] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[56] 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 *Proceedings of the 41st International Conference on Software Engineering*. 1027–1038.

[57] Arnab Sharma and Heike Wehrheim. 2019. Testing Machine Learning Algorithms for Balanced Data Usage. In *12th IEEE Conference on Software Testing, Validation and Verification*. 125–135.

[58] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-Implementation Testing of Supervised Learning Software. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence*. 384–391.

[59] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. 2014. Deep learning face representation by joint identification-verification. In *Advances in neural information processing systems*. 1988–1996.

[60] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd International Conference on Automated Software Engineering*. 109–119.

[61] Kim-Han Thung, Pew-Thian Yap, and Dinggang Shen. 2017. Multi-stage diagnosis of Alzheimer's disease with incomplete multimodal data via multi-task deep learning. In *Deep learning in medical image analysis and multimodal learning for clinical decision support*. 160–168.

[62] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[63] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial sample detection for deep neural network through model mutation testing. In *Proceedings of the 41st International Conference on Software Engineering*. 1245–1256.

[64] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[65] 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 *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 125–137.

[66] 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 *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[67] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine Learning Testing: Survey, Landscapes and Horizons. *arXiv preprint arXiv:1906.10742* (2019).

[68] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *2010 IEEE International Conference on Software Maintenance*. 1–10.

[69] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.

[70] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.

[71] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.