DevMuT: Testing Deep Learning Framework via Developer Expertise-Based Mutation

Yanzhou Mu[†], Juan Zhai[‡], Chunrong Fang[†*], Xiang Chen[#], Zhixiang Cao[#], Peiran Yang[†], Yinglong Zou[†], Tao Zheng[†], Zhenyu Chen[†]
[†]State Key Laboratory for Novel Software Technology, Nanjing University
{602022320006,peiranyang,652023320004}@smail.nju.edu.cn,{fangchunrong,zt,zychen}@nju.edu.cn
[‡]University of Massachusetts Amherst
juanzhai@umass.edu
[#]School of Artificial Intelligence and Computer Science, School of Zhang Jian, Nantong University
xchencs@ntu.edu.cn,zxcao@stmail.ntu.edu.cn

ABSTRACT

Deep learning (DL) frameworks are the fundamental infrastructure for various DL applications. Framework defects can profoundly cause disastrous accidents, thus requiring sufficient detection. In previous studies, researchers adopt DL models as test inputs combined with mutation to generate more diverse models. Though these studies demonstrate promising results, most detected defects are considered trivial (i.e., either treated as edge cases or ignored by the developers). To identify important bugs that matter to developers, we propose a novel DL framework testing method DevMuT, which generates models by adopting mutation operators and constraints derived from developer expertise. DevMuT simulates developers' common operations in development and detects more diverse defects within more stages of the DL model lifecycle (e.g., model training and inference). We evaluate the performance of DevMuT on three widely used DL frameworks (i.e., PyTorch, JAX, and MindSpore) with 29 DL models from nine types of industry tasks. The experiment results show that DevMuT outperforms state-of-the-art baselines: it can achieve at least 71.68% improvement on average in the diversity of generated models and 28.20% improvement on average in the legal rates of generated models. Moreover, DevMuT detects 117 defects, 63 of which are confirmed, 24 are fixed, and eight are of high value confirmed by developers. Finally, DevMuT has been deployed in the MindSpore community since December 2023. These demonstrate the effectiveness of DevMuT in detecting defects that are close to the real scenes and are of concern to developers.

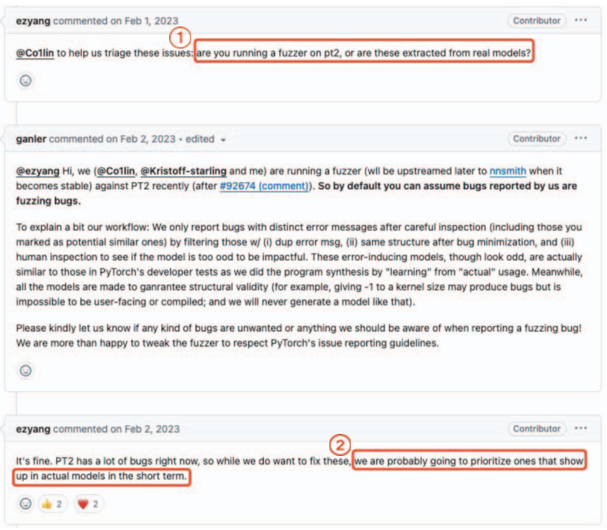
ACM Reference Format:
Yanzhou Mu[†], Juan Zhai[‡], Chunrong Fang[†*], Xiang Chen[#], Zhixiang Cao[#], Peiran Yang[†], Yinglong Zou[†], Tao Zheng[†], Zhenyu Chen[†]. 2024. DevMuT: Testing Deep Learning Framework via Developer Expertise-Based Mutation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695523

∗ Corresponding author

1 INTRODUCTION

Deep learning (DL) applications have achieved widespread success in various industry applications, including safety-critical fields such as autonomous driving [5], medical diagnosis [38], and facial recognition [54]. The development, maintenance, and deployment of such applications rely on DL frameworks. As such, framework defects can cause severe damages, leading to safety accidents and economic losses [51]. Therefore, it is imperative to test DL frameworks and detect defects for their quality assurance. Some researchers [18, 19, 28, 29, 42, 56] adopt DL models as the test inputs and combine them with different mutation operators, such as modifying the model structure and editing layer parameters, to generate more diverse mutants and improve the test effectiveness.

Although these methods have shown promising performance, they still struggle to detect the defects that developers are most concerned about. Figure 1 shows an example of the response by PyTorch developers on a defect report [35] submitted by a previous study [30]. It generates an edge model that is rare in the real world and detects an inconsistency defect about the group of PyTorch's "Linear" and "Batchnorm2d" operators. Developers' responses in


ezyang commented on Feb 1, 2023 — Contributor

@Co1lin to help us triage these issues: are you running a fuzzer on pt2, or are these extracted from real models?

ganler commented on Feb 2, 2023 • edited — Contributor

@ezyang Hi, we (@Co1lin, @Kristoff-starling and me) are running a fuzzer (wll be upstreamed later to nnsmith when it becomes stable) against PT2 recently (after #92674 (comment)). **So by default you can assume bugs reported by us are fuzzing bugs.**

To explain a bit our workflow: We only report bugs with distinct error messages after careful inspection (including those you marked as potential similar ones) by filtering those w/ (i) dup error msg, (ii) same structure after bug minimization, and (iii) human inspection to see if the model is too ood to be impactful. These error-inducing models, though look odd, are actually similar to those in PyTorch's developer tests as we did the program synthesis by "learning" from "actual" usage. Meanwhile, all the models are made to ganrantee structural validity (for example, giving -1 to a kernel size may produce bugs but is impossible to be user-facing or compiled; and we will never generate a model like that).

Please kindly let us know if any kind of bugs are unwanted or anything we should be aware of when reporting a fuzzing bug! We are more than happy to tweak the fuzzer to respect PyTorch's issue reporting guidelines.

ezyang commented on Feb 2, 2023 — Contributor

It's fine. PT2 has a lot of bugs right now, so while we do want to fix these, we are probably going to prioritize ones that show up in actual models in the short term.

Figure 1: Developer Response on a Defect Report

① and ② show that they prefer fixing defects in actual models over those generated by fuzzers. Even if the submitted defect is confirmed, developers usually ignore it since the model is rare in real scenes. In other words, the DL framework community mainly focuses on defects that profoundly threaten actual models applied in real scenes, such as unreasonable resource scheduling, abnormal performance, and substandard efficiency. This demonstrates the gap between developers and previously proposed methods.

More specifically, previous work's focus is misaligned with the developers' community and they suffer from three main limitations from the perspective of DL framework developers. First, **they detect impractical defects.** Existing solutions often modify the structure, parameters, and neuron weights to generate more diverse DL models, neglecting whether these modifications occur in real development and whether the generated models reflect actual cases in the real world. Though the generated models can expose defects, the triggered defects are rare in the real world and often automatically closed without any solutions after confirmation. Second, **they overlook critical defects.** They only focus on whether the generated models can execute normally and achieve consistent results across different frameworks in model inference while ignoring other subjects (e.g., resource usage) in other stages (e.g., training) of model lifecycles. It leads to missing the defects related to unreasonable resource scheduling, substandard efficiency, abnormal performance, etc, thus limiting the detection ability. These defects are more frequently exposed during development and often interfere with the users when using frameworks, which is also a concern for developers. Finally, the last limitation is **the consumption of massive resources**. They lack constraints to filter illegal models that may lead to crashes, produce abnormal outputs not caused by framework defects, and trigger abundant false positives. This wastes time analyzing low-value test inputs and requires extra manual efforts to identify false positives, thus reducing test efficiency.

To bridge the gap, we propose a novel DL framework testing method DevMuT, focusing on effectively exploring valuable test input space and detecting diverse defects based on developer expertise. Specifically, we collect practical experience in developing DL models from our industry partners and design seven mutation operators and nine constraints to avoid detecting impractical defects and save efforts caused by illegal models. Particularly, the mutation operators are designed to simulate the common operations of the developers on DL models, and the mutation constraints are adopted to guide how to mutate models and filter models that are meaningless in detecting defects. To detect more diverse and critical defects, we consult our industry partners about the main subject (e.g., resource usage) and defect type they care about during the model construction, training, and inference to design corresponding test oracles and detect diverse defects in these stages. In particular, we analyze common resources (such as memory, GPU, execution efficiency, and model performance) to examine differences across different frameworks. Besides, we detect whether there are crashes or abnormal outputs during execution. Finally, to further improve the effectiveness, we utilize the double-Q learning algorithm [20], a popular reinforcement learning algorithm to select mutation operators and seed models during the mutation process.

To evaluate DevMuT, we collect 29 models as seeds from nine kinds of industrial tasks. We apply DevMuT on three popular DL frameworks (i.e., PyTorch [43], JAX [24], and MindSpore [33]) with the above DL models as the test inputs. Compared with the state-of-the-art baseline methods [18, 28], DevMuT achieves an average increase of 71.68% on the generated model diversity. Meanwhile, DevMuT can effectively generate legal models with an 81.17% legal rate with an improvement of 28.20% compared with baselines. Moreover, DevMuT detects 117 defects, with three new kinds of defects not detected by baselines (i.e., six abnormal memory usage defects, six abnormal performance defects, and ten train crash defects). Among them, developers have confirmed 63 defects. 24 defects have been fixed, and eight defects are labeled with specific tags such as "Main", representing the defects that receive enough attention from developers. We also successfully contributed a pull request about fixing defects in MindSpore.

Our contributions can be summarized as follows:

- **Perspective.** We are the first to leverage developer expertise in designing mutation operators and constraints. Additionally, we extend the detection scope to include the construction and execution stages (including model training) of the model lifecycles rather than only the inference stage.
- **Approach.** We design a practical DL framework testing method DevMuT, which adopts the preset mutation operators and constraints combined with double-Q learning to effectively generate new models common in real scenes and detect more diverse defects.
- **Evaluation.** We collect a large-scale benchmark and conduct extensive experiments to evaluate the effectiveness of DevMuT. The results show that DevMuT detects 117 defects (63 confirmed) with three new types of 22 defects that baselines cannot detect. It also detects eight defects labeled with specific tags like "Main" to indicate higher priority for developers.
- **Practicality.** DevMuT has been applied in the MindSpore community [14] for continuous quality assurance since December 2023. We also implement DevMuT and share our proposed method on GitHub [47] for open science.

2 BACKGROUND AND RELATED WORK

2.1 DL Model and DL Framework

DL model comprises multiple middle layers consisting of one or more DL operators, such as convolution, pooling, and activation. These layers contain many neurons, and the connections between layers depend on the different weights of the neurons. These weights are updated with the external input data to fit specific scene tasks like image classification. The whole lifecycle of DL models includes the construction, execution, and deployment as shown in Figure 2. These stages require fundamental support from DL frameworks to ensure performance and correctness. Framework defects exposed in the model lifecycle are more diverse and highly relevant to actual scenes. Therefore, we introduce the model lifecycle and the correlations with DL frameworks. Developing DL models involves designing the architecture based on tasks, including the depth, width, neurons, and connections of different middle layers. DL frameworks encapsulate common-used functions as DL operators and hide the implementation complexity [16] for implementing the model architecture [15]. Model execution involves training and inference

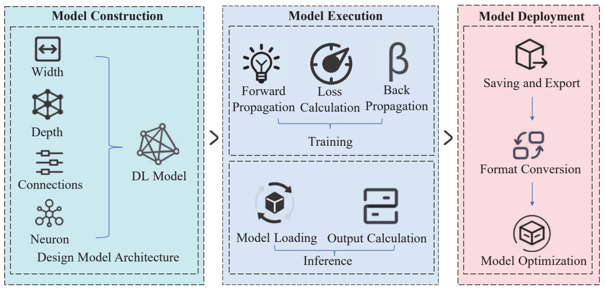


Figure 2: The Lifecycle of DL Models

processes. The training process includes forward propagation, loss calculation, and back-propagation: (1) forward propagation: the input data sequentially passes through the middle layers and calculates with the neuron weights; (2) loss calculation: it calculates the difference between the model outputs and the real labels through the loss function to evaluate the accuracy; (3) back-propagation: the model updates the weights from the last output layer and reverses to the first input layer by adjusting the gradients of each weight and optimizing to reduce the loss. The three steps are repeated to reduce losses and improve model performance until it stabilizes. After training, the model performs inference on new data by executing forward propagation and conducting predictions. DL frameworks also implement the functionalities for these computing tasks to improve efficiency and simplify execution scripts. Moreover, DL frameworks efficiently schedule computing resources for smooth execution and optimize global resource scheduling to ensure stable and efficient execution under parallelism requirements. Deployment involves saving and exporting the trained DL models, converting them into specific formats, ensuring compatibility with the target environment, and optimizing for inference speed, resource use, and stable execution. DL frameworks efficiently support cross-platform compatibility for different environments and enhance inference speed and resource utilization through optimization techniques.

2.2 Testing DL Frameworks via DL Models

Existing works that adopt DL models as test inputs for DL frameworks can be divided into two types: (1) one kind of method utilizes mutation operators to mutate existing DL models in public repositories [19, 28, 42, 56]; (2) another kind of methods generate models based on predefined templates and constraints [18, 29–31, 55].

Mutation-Based Testing Methods. Pham et al. [42] proposed CRADLE, the first model-level testing method that utilizes differential testing by analyzing model execution on different backends of Keras [27] (e.g., TensorFlow [50], CNTK [10]). Wang et al. [56] proposed LEMON and adopted mutation operators in DeepMutation [32] and DeepMutation++ [22] combined with the MCMC strategy, which has been widely used in many fields [8, 9] to guide the mutation process. It utilizes differential testing on the generated mutants to detect defects. Guo et al. [19] proposed Audee, which combined genetic evolution strategy [53] with input-mutation to generate mutants that can trigger higher inconsistency and outlier output. Li et al. [28] proposed COMET and added new types of mutation operators, such as parameter-mutation and outlier mutation, compared with LEMON. Its execution process is similar to LEMON.

Generation-Based Testing Methods. Gu et al. [18] proposed Muffin, which generates new models based on the structure of the directed acyclic graph. It focuses on detecting inconsistency defects in three model execution processes: forward propagation, loss calculation, and back-propagation calculation. Luo et al. [31] proposed Graphfuzz to test DL inference engines based on graph theory. It combines six mutation operators with the MCTS algorithm [11] to explore the generation of new model structures. Liu et al. [29] proposed Gandalf, which generates different models using context-free syntax combined with the DQN algorithm [34]. It introduces 15 metamorphic relationships to enhance the testing effectiveness of different DL frameworks. Liu et al. [30] proposed Neuri, which generates models through inductive rules. It first collects the call relationships of framework interfaces from external resources and analyzes legal interface groups to generate test inputs that can explore the deeper execution behavior of DL frameworks.

Comparison with Previous Studies. As the mutation-based method, DevMuT shows the following novelties. (1) The mutation operators adopted in existing methods are directly from the DL model testing work [22, 23, 32]. In contrast, the mutation operators in DevMuT are abstracted from the common operations of developers on DL models. (2) Existing methods fail to filter out illegal models, while DevMuT goes a step further by filtering them based on developer tactics. (3) Existing work focuses on detecting the inconsistency defects exposed in model inference while DevMuT detects other kinds of defects from the lifecycle of DL models closer to real scenes.

3 MUTATION DESIGN BASED ON DEVELOPER EXPERTISE

To detect defects that are more relevant to real scenes, we interview developers to gather their expertise in developing DL models and framework defects. Based on the results, we aim to design mutation operators to simulate their common operations in development and design mutation constraints abstracted from development tactics to guide the mutation and filter illegal models. Next, we introduce the interview process, present the interview result analysis, and conclude the findings. Finally, we further introduce the design of the mutation operators and constraints based on the findings.

3.1 Interview Process

We design an open guideline to clarify our background and interview process with different types of questions, e.g., short answer, choice, and open-ended questions. Details can be found on our website [47]. Here are five aspects of interview questions.

Part I: We ask them five short questions about their basic profiles including jobs, duties in DL framework development, and working experience in years.

Part II: We ask them three short questions and one multi-choice question about the subjects (e.g., model scripts) they often process and their specific operations on these subjects.

Part III: We ask them five short answer questions about their experience in identifying and dealing with those weird models that cannot fit into real scenes.

Part IV: We ask them six short answer questions about the defect they care about with a specific introduction about the damage to software, symptoms, root causes, trigger frequency, etc. We also asked how they evaluate the importance of defects.

Part V: We ask them six open-ended questions about their opinions on existing frameworks and expectations for optimizations.

We invite six professionals from leading Internet companies as interviewees who are our industry partners, specialize in DL framework quality assurance, and are responsible for developing, testing, and detecting defects in new releases. With over four years of experience in defect detection, localization, and fixing, the interview quality can be guaranteed. We begin by introducing the guidelines and providing background information on our research. We inform the participants that the interview will be recorded, obtain their approval, and ensure confidentiality. Each interview is conducted online and lasts approximately 30 minutes. After the interview, we verify that no sensitive information is disclosed and send our gratitude for their participation.

Results Analysis. After all interviews, one author transcribes records and extracts the development operations and defect types. The other three authors verify the results and provide optimization suggestions. All the authors then discuss different opinions until reaching a consensus. We conclude with three main findings from the interview results.

Findings about the Common Operations in Development. The operations refer to the developers' modifications on various model scripts. We find two types of scripts that developers frequently modify: (1) structure scripts that design the architecture of models and (2) execution scripts that control the training and inference. For the structure scripts, we find that developers often introduce new structures to replace existing structures, add them to original models, delete existing structures, and modify the parameters to improve model performance. For the execution scripts, we find that developers often modify data, loss function, and optimizer to enhance the generalization ability and improve convergence speed. Specifically, they often conduct data augmentation, change the loss function and optimizer type, and modify their parameters. These findings motivate us to identify the mutation objects and operators, i.e., we adopt the structure script and execution script as the mutation objects while the operations are implemented as mutation operators.

Findings about the Tactics in Development. The tactics are (1) the notes when modifying scripts and (2) the experience of identifying illegal models. For the first kind of tactic, developers provide six guidelines, covering requirements for adding or replacing new structures, shape dimensions, parameter setting ranges, and the position for adding or deleting middle layers. For the second kind of tactic, developers suggest three tactics, ranging from time-consuming to output accuracy and gradient value, to identify illegal models. These findings motivate us to design the constraints to guide the common operations and identify those weird models after modifications.

Findings about the Defects Developers Care About. We conclude the subjects that developers often monitor to detect defects and the relevant defect types. Specifically, developers often focus on resource usage (e.g., CPU, memory, and GPU), model performance such as training loss and inference evaluation, execution efficiency, and output accuracy in model execution. The defects related to these subjects (e.g., substandard performance and crashes) achieve higher fix priority of developers since they believe these defects frequently appear in real-world scenarios and significantly impact users. The findings guide us to design test oracles to detect defects and are introduced in Section 4.3.

We present these findings to our interviewees for confirmation and obtain their approval. Moreover, they give corresponding suggestions about defect detection. All the details of the interviews can be found on our website [47]. In summary, we design seven mutation operators to simulate the common operations of developers on the structure and execution scripts based on the above results. Meanwhile, we also design six constraints to guide mutation and three constraints to filter illegal mutants abstracted from the tactics. Besides, we focus on four types of defects as recommended by developers with ten test oracles for detection. The details of mutation operators and constraints are in the following part of this section while the defects and the test oracles are in Section 4.3.

3.2 Mutation Operator Design

We design the mutation operators based on interview insights from three aspects. **Modified Objects.** Developers frequently modify the model structure and execution script, focusing on middle layers, layer parameters, and loss/optimizer functions to enhance performance. This leads us to target these objects for mutation. **Operator Type.** Developers often add, delete, or replace middle layers, and expand or reduce data outputs or layer parameters. We implement mutation operators to simulate these actions. **Mutation Guidelines.** Developers adhere to guidelines, such as not placing one Conv2D layer immediately after another, to maintain model performance. We incorporate these constraints to ensure the validity of mutations. The details about the mutation operators are as follows. We adopt (1) the model structure script and (2) the execution script as mutation subjects and propose seven mutation operators (MOs).

As the first kind of mutation subject, the model structure can be divided into three types: (1) backbones for extracting features from input data such as the VGG [48], ResNet [21], Inception [49]; (2) cascade operators consisting of multiple DL operators for executing specific tasks (e.g., the RPN [44] structure for identifying target area in object detection); (3) basic operators that cannot be further subdivided, such as Conv2d, etc. The five kinds of common operations on the above three types of structures are as follows and each operation represents one type of mutation operator.

MO1: Existing Structure Replacement. This mutation operator refers to inducing external model structures (such as new backbone, cascade operators, and basic operators) to replace the existing structures in the seed model.

MO2: Shape and Dimension Change. This mutation operator refers to adjusting the input/output size or dimension of the basic operators or the first and last of cascade operators. Developers enlarge dimensions or shapes to enhance learning on complex data, reduce them to lower computing costs, and improve generalization ability.

MO3: Series Connection Addition/Deletion. This mutation operator refers to changing the depth (i.e., the total number of middle layers) of DL models by adding or deleting existing cascade or basic operators that are ordered in series connection. Developers often insert new structures to enlarge the model depth, learn the data feature deeper, and delete replicate structures for higher efficiency.

MO4: Parallel Connection Addition/Deletion. This mutation operator refers to changing the width (i.e., the total number of neurons in each layer) of DL models by adding or deleting branches consisting of cascade operators or basic operators. Developers add new branches to capture multi-dimensional data features, enhance model robustness, and enable parallel training. Besides, They also prune branches to avoid overfitting and simplify computation.

MO5: Parameter Value Change. This mutation operator refers to modifying the parameter values of the basic operators. Developers modify layer parameters to adjust the model's width and depth, avoid overfitting, and adapt to new tasks, as changes in these parameters influence performance.

As the second kind of mutation subject (i.e., the execution script), the main objects include the loss function, optimizer, training/test set, and parameters to decide training or inference execution. We design the following two mutation operators on loss function and optimizer as recommended by developers.

MO6: Loss Function Change. This mutation operator involves the loss function's type and parameter value change, as developers often do it to adapt to different kinds of new tasks.

MO7: Optimizer Change. This mutation operator involves the optimizer's type and parameter value change, as developers often do to promote the training process.

Finally, we count their proportion mentioned by interviewees: MO1, MO2, MO3, MO4, MO5, MO6 and MO7 are mentioned by 100%, 50%, 83.3%, 83.3%, 66.7%, 50% and 50% interviewees, respectively.

3.3 Mutation Constraint Design

We collect the tactics for modifying models and estimating whether the models are legal from the interview results. Specifically, we design two types of constraints (MCs): (1) to guide mutation on seed models and (2) to filter illegal models. Such MCs are adopted to reduce false positives and improve the efficiency of DevMuT. We introduce these MCs and show the details on our website [47].

MC1: Structure Functionality Consistency Constraints. The new structures should have the same functionality to keep the consistency of the target model (e.g., replace a normal convolution with a depthwise separable convolution) since replacing new structures of different functions (e.g., replace a normalization layer with an activation layer) may disrupt the stability, increasing execution costs and reducing the interpretability. We collect five kinds of backbones (e.g., ResNet) to extract image data features, seven kinds of cascade operators (e.g., residual block) to enhance the feature learning, and five kinds of basic operators (e.g., avgpool and maxpool operators) with different functionalities for candidate replacement.

MC2: Data Calculation Scale Constraints. We set the change rate of shape or dimension from one-fourth of the original shape or dimension to four times as recommended by developers to avoid excessive computing costs.

MC3: Series Structure Addition/Deletion Constraint. We only add new cascade operators with the same task type and avoid adding the same basic operators at the same position (e.g., add one "Conv2d" after one "Conv2d"). Besides, we only select one of the multiple similar structures to delete (e.g., delete the one entire "Conv2d-Batchnorm2d-Relu" structure in VGG) for the functionality consistency between the mutation model and the origin. According to the developer's recommendation, the range for adding or removing series structures for common image classification and object detection models should not exceed 20% of the original model depth.

MC4: Parallel Structure Addition/Deletion Constraint. Similarly to MC3, the new branch structure should share the same functionality as the original structure. Besides, the same position cannot be added to more than two branches to reduce time costs.

MC5: Parameter Value Range Constraints. The change value range is decided based on their effects on model performance. We select 12 kinds of parameters from four basic operators as the mutation range followed by the recommendation of developers. For example, for the parameter "kernel_size" of the "Conv2d" operator, we follow the developer's recommendation and adopt three different sizes: "3*3", "5*5" and "7*7" to adapt to different data features. The other value change range can be found on our website [47].

MC6: Loss/Optimizer Modification Constraints. The types of loss functions and optimizers with their parameters are decided based on the model tasks. We collect four kinds of optimizers and four kinds of loss functions as the candidates.

Besides, we propose three filter constraints to filter weird models:

MC7: Execution Time Constraints. Developers suggest that the acceptable range for the increased training time of the generated model is less than two to three times and the increased inference time is less than one to two times compared with the original. We take the inference time as the standard and set the threshold three times to eliminate external interference.

MC8: Output Accuracy Constraints. The output value exceeding the accuracy range (e.g., the accuracy range of MindSpore is under 1e38 at "float32") is meaningless and produces outliers like "NAN" "inf". This may induce more false positives and reduce test effectiveness. We adopt the accuracy range of "float16", "float32" and "float64" as the constraints to filter those models with invalid output.

MC9: Gradient value range Constraints. If the output gradient of the generated model contains outliers (exceeds 1e2 or is lower than 1e-3), it is filtered. Such models do not change performance since the early training stage and are worthless for analyzing their execution.

4 APPROACH

4.1 Overview

Figure 3 presents the workflow of DevMuT. It consists of two parts: *Model Mutation* that generates mutants, and *Defect Detection* that performs defect detection. In *Model Mutation*, DevMuT adopts two deep Q-networks to select mutation operators and seed models. DevMuT mutates the seed model under the guideline constraints (MCs 1-6 in Section 3.3) and updates the Q-networks based on the reward calculated from the evaluation of the current mutation. It filters out mutants that violate constraints (MCs 7-9 in Section 3.3) and adds legitimate ones into the generated model pool. The process iterates until mutation finishes (Section 4.2). In *Defect Detection*, DevMuT first detects and records the defects exposed in mutants. Based on rewards, DevMuT selects part of the models and further executes them to detect defects in resource scheduling, execution crashes, model performance, and output accuracy (Section 4.3).

4.2 Model Mutation

This section introduces how DevMuT mutates models as shown in Algorithm 1 (Lines $1-22$) while the core is to select mutation operators and seed models. DL models contain complex structures and numerous parameters and are more complicated after mutation, further increasing the exploration difficulty of the test input space. To overcome this challenge, DevMuT adopts double-Q learning [20], a

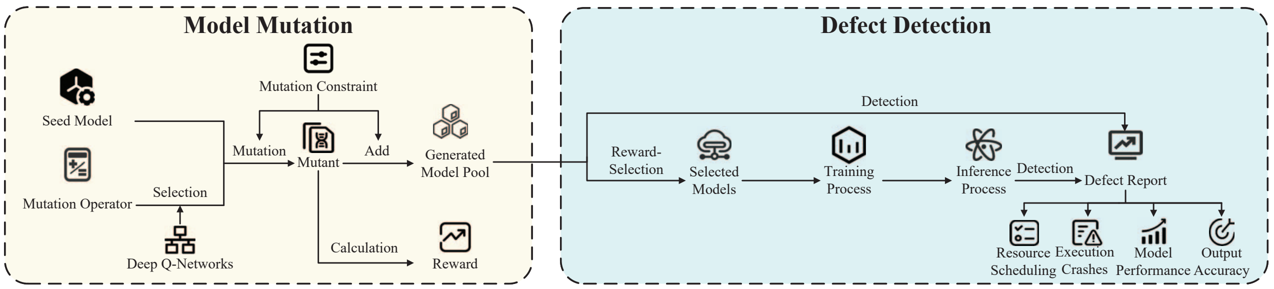


Figure 3: Workflow of DevMuT

Algorithm 1: The DevMuT Algorithm

```
Input  : $M$: the original DL model; $S_{pool}$: the mutation operator pool; $N$: the number of
         mutation execution rounds; $k$: the selected number of models;
Output : $DefectSet$: the set of all the detected defects;
1  $n \leftarrow 0$, d $\leftarrow M$, $DefectSet \leftarrow \emptyset$, $D \leftarrow \{M\}$, init($Q_1$, $Q_2$);
2  while $n \leq N$ do
3      $p \leftarrow$ random.uniform(0, 1);
4      if $p \leq \epsilon$ then
5          s $\leftarrow random.choice(S_{pool})$;
6      else
7          s $\leftarrow S_{pool}[\arg\max(\frac{Q_1(d,*)+Q_2(d,*)}{2})]$;
8      $d' \leftarrow$ s($d$);
9      d.select_num++, d.snums[indexof(s)]++;
10     if judge($d'$) == 1 then
11         r $\leftarrow$ -1, $d'$.r $\leftarrow$ r;
12         $d' \leftarrow UCBSort(D)$;
13     else
14         r $\leftarrow distance(d', d)$, $d'$.r $\leftarrow$ r;
15         if $d'$ not in $D$ then
16             $D$.add($d'$.copy());
17     p $\leftarrow$ random.uniform(0, 1);
18     if $p \leq 0.5$ then
19         s' $\leftarrow S_{pool} \arg\max(Q_1(d', *))$
           $Q_1.update(Q_1(d,s) + \alpha[r + \gamma Q_2(d', s') - Q_1(d,s)])$
20     else
21         s' $\leftarrow S_{pool} \arg\max(Q_2(d', *))$
           $Q_2.update(Q_2(d,s) + \alpha[r + \gamma Q_1(d', s') - Q_2(d,s)])$
22     d $\leftarrow$ d'.copy(), n++;
23 foreach $d$ in $D$ do
24     ifbug $\leftarrow detect\_bug(d, DefectSet, "Mutation")$;
25     if ifbug then
26         $D$.remove($d$);
27 $D \leftarrow Sort(D)[:k]$;
28 foreach $d$ in $D$ do
29     $detect\_bug(d, DefectSet, "Execution")$;
30 return $DefectSet$;
```

reinforcement learning strategy that utilizes two deep Q-networks for selection. It can efficiently identify valuable parts of the test input space and continuously explore it with two Q-networks.

The problem in our scene is modeled using reinforcement learning (RL) to explore the test input space. Key elements of RL are "state" which represents the current situation of the environment, "action" which represents the transformations between two states, and "reward" which represents the feedback received from the environment. In our scenario, "state" is the generated model, while "action" is the mutation operator. Since we aim to generate more diverse models and different from the original model, we adopt the output distance between the generated and original models as the "reward". The workflow based on the above definition is shown in the following parts.

DevMuT first initializes the parameters (Line 1): (1) the selected seed model d to mutate; (2) two networks Q_1 and Q_2 for selecting mutation operators; (3) the set $DefectSet$ that stores detected defects; (4) the mutation counter n; (5) the set D stores the generated models. Then it executes N rounds of mutation (Lines 2-22). To effectively select mutation operators, DevMuT utilizes the ϵ-greedy policy [34] strategy: it generates a probability p from the symmetric distribution (0,1) (Line 3) and compares it with the preset threshold ϵ (Line 4). If p is smaller than ϵ, DevMuT randomly selects one mutation operator (Line 5). Otherwise, it calculates the output average of Q_1 and Q_2 for selecting the mutation operator s (Line 6-7). The probability p determines the likelihood of selecting a mutation operator strategy. Operators are more likely selected based on the output of two Q-networks with larger p values, as this double Q-network strategy explores a more diverse test input space. Then DevMuT mutates d with s to generate the model d' (Line 8). If the current mutation fails (e.g., generate crash models) or generates those illegal models that violate three constraints (Line 10), DevMuT sets the reward r to -1 for punishing the selection of s based on d (Line 11) and selects a new seed model with the maximum value of UCB (Line 12) and assign it to d'. UCB [2] is often adopted to avoid trapping into the local optimum and give more chances to explore new space. The definition of UCB is shown as follows:

$$UCB(d) = d.r + c * \sqrt{\frac{\ln(d.select\_num)}{d.snums[indexof(s)]}} \tag{1}$$

In the above formula, $d.r$ represents the current reward of d, $d.select\_num$ represents the selected counts of the seed model, $d.snums$ represents the selected counts of s in d, c is the weight to balance the left and right parts, respectively. $d.select\_num$ and $d.snums$ are updated after the mutation (Line 9).

If the generated model d' does not violate the constraints (Line 13), DevMuT calculates reward r between the current seed model d and d' (Line 14) and update the model pool D (Lines 15-16). The definition of the reward is as follows: Given two models D_1, D_2 with n layers $f = \langle L_0, L_1, \cdots, L_n \rangle$ and an input tensor x, the output of the i-th layer is recorded as $f_{L_i}(x)$. We adopt Chebyshev distance [4] for calculating the distance between D_1 and D_2 on layer i:

$$r_{f_{L_i}}^{D_1,D_2}(x) = mean(|D_{1 f_{L_i}}(x) - D_{2 f_{L_i}}(x)|) \tag{2}$$

We adopt the final layer as the basis for calculating reward. Then, DevMuT randomly updates one of the two Q-networks (Lines 17-21). We follow the original update process of double Q-learning, i.e., combine the currently expected reward $Q_1(d,s)$ or $Q_2(d,s)$ (left part of the Lines 19 and 21) with the future expected reward $Q_1(d', s')$ or $Q_2(d', s')$ (right part of the Lines 19 and 21). The parameters

α and γ are adopted to control the update speed and measure the contribution of the future reward, respectively. Finally, DevMuT utilizes d' as the seed model of the next round (Line 22).

4.3 Defect Detection

This section introduces DevMuT for defect detection and presents the test oracles. As shown in Algorithm 1 (Lines 23-29), DevMuT further detects defects within mutation by analyzing the models in set D that stores generated legal models (Line 24) and removes the defective models from D (Lines 25-26). DevMuT then sorts D based on the reward value and selects the top k models (Line 27) for further analysis. It further trains the selected models and executes inference to detect defects (Lines 28-29). Finally, DevMuT returns $DefectSet$ as the output (Line 30). Since these subjects have different value ranges on different models due to the input data, scene task types, etc, we run all the models across different frameworks to collect preliminary results (e.g., memory usage size) about these subjects. Then, we further set the thresholds for determining the generated models expose the relevant defects. It also records crashes during execution. We use a table to show the thresholds of each model about these subjects, which can be found on our website [47]. The test oracles for determining whether these subjects expose defects are as follows.

Performance Defect. The performance involves the training loss, execution efficiency, and inference evaluation. We design two test oracles about loss: (1) the model outputs an abnormal loss value (e.g., NAN and inf) on one framework but not on another; (2) when the loss value is normal, we adopt dynamic time warping (DTW) distance [36] to measure the loss similarity of the models across different DL frameworks and report defect if the DTW distance exceeds the preset threshold. It is commonly used in speech recognition [57] to measure the similarity between two time series. Suppose one model executes n rounds and records the average loss value of each round across different DL frameworks. Record two frameworks as X and Y and the two loss series are $X_{loss} = (x_1, x_2, ..., x_n)$, and $Y_{loss} = (y_1, y_2, ..., y_n)$. $D(i, j)$ is the DTW distance between the first i elements of X and the first j elements of Y. The calculation of DTW is shown in the following formula:

$$D(i,j) = d(x_i, y_j) + min(D(i-1,j), D(i,j-1), D(i-1,j-1)) \quad (3)$$

In Formula (3), $d(x_i, y_j)$ represents the euclidean distance between x_i and y_j and We set $D(0,0) = 0$, $D(i,0) = \infty$ and $D(0,j) = \infty$ to ensure the correctness. We calculate $D(n,n)$ as the distance between X_{loss} and Y_{loss}.

Besides, we record the time of the model across different frameworks during model training and inference. If the time of one framework far exceeds the other compared with the preset threshold, DevMuT reports an efficiency defect. We also compare the evaluation metric of the models across different frameworks and report a performance defect when the evaluation metric difference of models across different frameworks exceeds the preset threshold.

Resource Defect. The resources refer to the GPU and memory, and the defects are related to unreasonable scheduling, such as memory leaks. DevMuT records the size of memory and GPU used across different DL frameworks during execution and detects defects from the perspective of the three aspects: (1) the used resources anomalously increase and lead to memory or GPU leak; (2) the maximum used resource exceeds the preset threshold; (3) the cosine similarity calculated by the memory or GPU size of the models across different DL frameworks is lower than the preset threshold.

Accuracy Defect. The accuracy is related to the model outputs in forward propagation and back-propagation. If the model outputs across different frameworks exceed the preset distance threshold, DevMuT reports an accuracy defect. We also adopt Chebyshev distance [4] to measure the output distance, and its calculation is shown in Formula 2. Besides, if the model's output contains outliers on one framework but not on another or contains different outliers, DevMuT also indicates an accuracy defect.

Crash Defect. We detect the crashes exposed in the mutation process and model execution (i.e. training and inference). More specifically, if a model mutates successfully on one framework but fails on another or exposes crashes during the training or inference on one framework but not on another, we report a crash defect.

5 EXPERIMENT DESIGN

5.1 Research Questions

Our study investigates the following three research questions:

- **RQ1: How does DevMuT perform compared to existing work?** We evaluate the performance of DevMuT from the perspective of the diversity of the generated test inputs and detected defects compared to the existing methods.
- **RQ2: What kind of defects are detected by DevMuT?** We analyze the defects detected by DevMuT and show the typical cases in detail, including the types, symptoms, and root causes.
- **RQ3: How does the search strategy contribute to DevMuT?** We perform an ablation experiment to evaluate the double-Q search strategy's effectiveness and its contribution to DevMuT.

5.2 Benchmark

The benchmark used in our study includes DL frameworks, DL models, and the corresponding datasets. The details are as follows.

DL Frameworks. We select the latest versions of three DL frameworks as the test objects, i.e., PyTorch 1.10.1, MindSpore 2.2.0, and JAX 0.4.27. PyTorch is a popular DL framework that supports flexible conversion to different mobile systems and dynamic computing graphs for convenience in development. MindSpore is a new DL framework designed for various running environments with an efficient execution engine to enable users to manage computing resources and optimize task execution efficiently. JAX provides powerful automatic differentiation and GPU/TPU support and is widely used in DL development and scientific computing. Both three DL frameworks have been widely adopted in prior work [29, 30, 58] and have active developer communities in the recent three years.

DL Models and Datasets. We collect 17 DL models from real industry scene tasks to ensure the practical meaning of our study. Specifically, VGG16, ResNet50, MobileNetv2, and VIT are four image classification models used in facial recognition. The YoLoV3, YoLoV4, and two SSD models conduct object detection tasks and are widely applied in autonomous driving. The Unet, Unet++, DeeplabV3, and DeeplabV3++ are semantic segmentation models for medical diagnosis. The TextCNN conducts sentiment analysis on movie reviews. Lastly, SSIM-AE, PatchCore, OpenPose, and CRNN handle image-based defects, anomaly detection, key point detection, and scene

Table 1: Statistics of DL Models in Our Study

Scene Task	Model	Parameters	Depth	Width
Image classification	VGG16-1	134,314,186	53	4,096
	ResNet50-1	23,581,642	126	2,048
	MobileNetV2-1	2,270,794	141	1,280
	VIT	87,423,754	150	3,072
	AlexNet*	24,736,682	17	4,096
	LeNet5-1*	107,764	10	120
	LeNet5-2*	218,302	11	120
	ResNet50-2*	123,914,152	126	2,048
	VGG19*	143,661,736	45	4,096
	InceptionV3*	1,254,998,216	26	1,000
	DenseNet121*	7,045,504	248	1,000
	VGG16-2*	138,353,320	39	4,096
	Xception*	29,384,336	153	2,048
	MobileNetV2-2*	4,234,024	57	1,024
Regression Prediction	LSTM-1*	81,301	7	1
	LSTM-2*	3,726	4	1
Object Detection	SSD-resnet50-fpn	33,370,814	232	2,048
	SSD-mobilenetv1	11,329,461	143	1,024
	YoloV3	62,001,757	222	1,024
	YoloV4	65,741,916	334	1024
Semantic Segmentation	DeeplabV3	58,149,077	263	2,048
	DeeplabV3++	59,453,824	272	2,048
	Unet	31,029,698	53	1,024
	Unet++	9,045,280	88	512
Text Classification	TextCNN	859,146	12	96
Anomaly Detection	PatchCore	68,951,464	129	2,048
Defect Detection	SSIM-AE	2,720,768	39	500
Key Point Detection	OpenPose	52,311,446	112	512
Scene recognition	CRNN	4,339,621	29	512

recognition. Besides, we also collect 12 DL models (labeled with "*" in Table 1) from existing work [19, 28, 56] to ensure the fairness of our study. The details are shown in Table 1. The first and second columns show the task and name of the DL models. The third to fifth columns show the parameter scale, model depth, and width.

5.3 Baseline Methods

We select two state-of-the-art DL framework testing methods, i.e., COMET [28] and Muffin [18]. COMET is the latest mutation-based testing method with comprehensive types of mutation operators ranging from weight-mutation, input-mutation, and parameter-mutation to structure-mutation, which shares the same type as DevMuT. It aims to improve the middle layer coverage and generate new model structures with more diversity. Muffin generates new models based on the directed acyclic graph structure and detects defects in the model training stage, including forward calculation, loss calculation, and backward calculation, which shares the most common detection scope with DevMuT.

5.4 Measurements

We evaluate DevMuT based on two measurements, i.e., generated mutants and detected defects. For the first kind of measurement, we adopt three kinds of layer coverage to evaluate the diversity of the generated models adopted in previous work [28]:

Layer Input Coverage (LIC). Record the total number of data types as N_{type}, the total number of dimensions as N_{dim}, and the total number of shapes as N_{shape}. The number of data types covered is n_{type}, the number of dimensions covered is n_{dim}, and the number of shapes covered is n_{shape}. The layer input coverage $Input_{cov}$

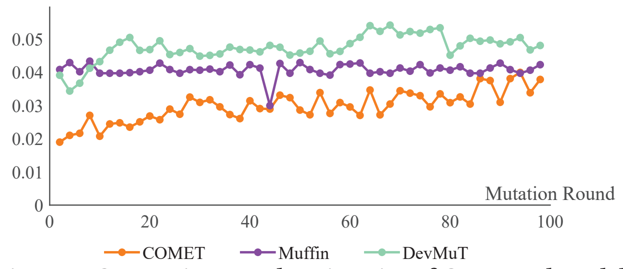


Figure 4: Comparison on the Diversity of Generated Models

is calculated as the result of the sum of N_{type}, N_{dim}, and N_{shape}, divided by the sum of n_{type}, n_{dim}, and n_{shape}.

Layer Parameter Coverage (LPC). Record the number of different middle layers in the current generated models as n_{ut}, and the number of different edges in all the generated models as N_{ut}. The layer parameter coverage is calculated as the result of N_{ut} divided by n_{ut}.

Layer Sequence Coverage (LSC). The layer sequence refers to the groups consisting of two different framework interfaces. Record the total number of all the interface groups that appeared in the execution as AP_{sum} and the number of current generated models as AP_{cur}. The layer sequence coverage is calculated as the result of AP_{sum} divided by AP_{cur}.

Besides, we further adopt the Scott Knott ESD test [6, 7, 25] to analyze the performance of *DevMuT* and baseline methods on these three metrics. This statistical test method is often used to effectively identify significant differences and categorize the results of different methods into distinct classes.

For the second kind of measurement, we measure the defect detection ability of different testing methods based on the type and count of detected defects. Then we further count the number of reported defects, the confirmed defects, and the fixed defects in detail.

6 RESULT ANALYSIS

6.1 RQ1: Compared with Baseline Methods

We evaluate DevMuT from two aspects: (1) the diversity of generated models and (2) the detected defects compared to baseline methods. For the first aspect, we run DevMuT and baseline methods for 100 rounds followed by the experiment settings of existing work [56] and calculate the *LIC*, *LPC*, and *LSC* metrics. Besides, we also count the illegal models that violate the mutation constraints. For the second aspect, we analyze the defects detected by baselines exposed in our experiment and previous studies with those detected by DevMuT. Besides, COMET cannot execute on DL models collected in our study (models are not labeled with "*" in Table 1) for the following reasons. (1) Its implementations cannot fit the new models since their structures are more complex than those previously used, thus always leading to crashes. (2) Modifying the implementations to accommodate new models may lead to uncontrollable effects on COMET and compromise fairness. Therefore, we only run DevMuT and COMET on those 12 DL models applied in existing work for fairness.

Comparison based on the Generated Models. We show the average results in Figure 4. The x axis represents the mutation rounds while the y axis represents the average value of three diversity metrics of each generated model. Among the three diversity measurements, DevMuT outperforms the other two strategies in

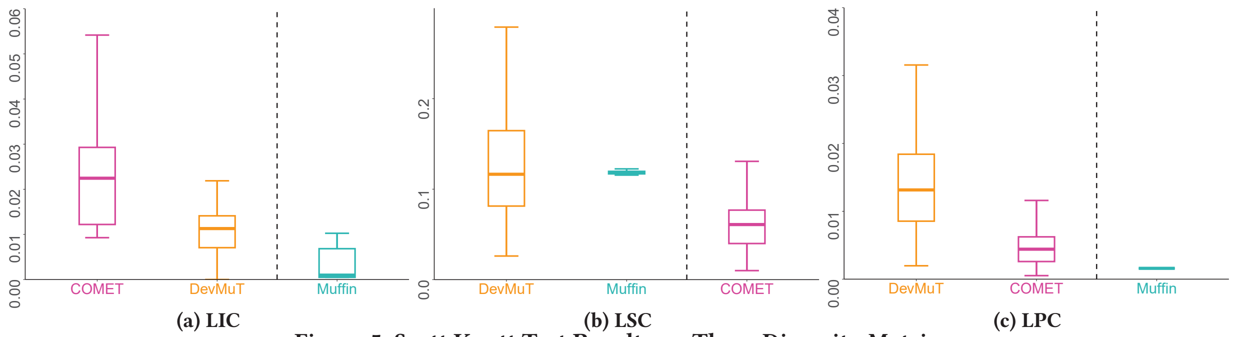


Figure 5: Scott Knott Test Results on Three Diversity Metrics

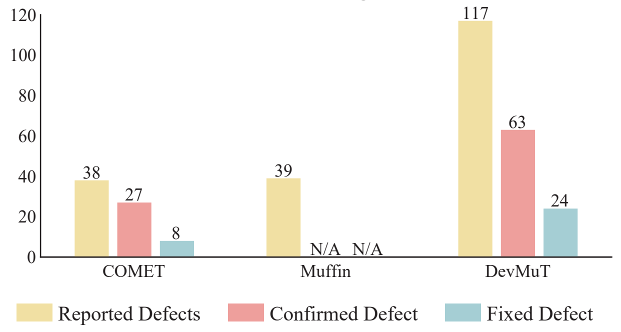


Figure 6: Number of Defects Detected by Different Methods

LSC and ranks second in the other two metrics. However, DevMuT achieves the best performance based on the average of the three indicators as shown in Figure 4. More specifically, DevMuT can introduce richer cascade operators (such as FPN structures) into the original model, thus increasing more diverse layer sequences. Since DevMuT follows the default setting of new structures, the parameter settings and input/output sequence are lower than Muffin and COMET, respectively. COMET combines the three metrics to select the mutation operator and seed model with no constraint, e.g., the value range of parameter mutation, so it can achieve higher *LPC*. However, the generated models of COMET greatly differ from the real-world model, with 36.69% percent (434 illegal models in 1183 models) of models that violate the mutation constraints. DevMuT only generates 226 illegal models and achieves 81.17% legal rates in total. Muffin has a strong shape/dimension matching mechanism and can effectively avoid models with crashes. Meanwhile, it can generate more diverse groups of input/output sequences of middle layers, thus achieving higher *LIC* than the other two methods. However, although the generated model does not violate the mutation constraints, they are only a combination of several DL operators and not actual models that can adapt to industrial tasks. Besides, we conduct Scott Knott ESD test on these three metrics and show results in Figure 5. The x-axis represents different methods, while the y -axis shows the values of the three metrics. The dashed line perpendicular to the x-axis separates methods with significant performance differences; methods on opposite sides show statistically significant differences, whereas those on the same side do not. In this figure, though DevMuT may not always achieve the highest model diversity, it maintains a comparable ability to generate diverse models while ensuring the model's validity and practical significance, and its overall performance is superior to existing methods.

Comparison based on the Detected Defects. We count the defects detected by different methods and show the results in Figure 6. In particular, the x axis represents the method name; the y axis represents the counts of the total detected defects, the confirmed defects, and the fixed defects. Specifically, COMET reports no inconsistency defects, four NAN defects, and six crash defects in our study after manual inspection. Four NAN defects and three crash defects receive confirmation. Besides, COMET previously reports 28 defects, of which developers confirm 20, including five inconsistency defects, three wrong output defects, one NAN defect, seven interface implementation defects, two conversion failure defects that are related to the conversion frameworks, and two "Core Dump" defects related to two DL operators. Muffin does not run on new DL frameworks and reports 17 inconsistency detects, one NAN defect, and 21 crash defects in their previous studies. Since they do not report further confirmed information, we fix "N/A" on the count of confirmed and fixed defects of Muffin in Figure 8. Among all inconsistency defects, over 66.7% are due to operator implementation errors, with only one defect related to the incorrect use of the "BinaryCrossentropy" loss function in TensorFlow. The back-propagation inconsistency defect stems from the improper implementation of the "ReLU" function. One NAN defect is attributed to an implementation error in the "GlobalMaxPooling" function while the remaining 21 crash defects have not been reported or analyzed in detail. DevMuT detects 17 defects during execution, and 14 are confirmed. Except for one inconsistency defect and three NAN defects, DevMuT can detect three new types of defects (i.e., memory leak, efficiency and performance decrease during training, and crashes exposed in training) that cannot be detected by existing work. The number of three types is all two. The remaining six defects are related to the wrong implementations, unreasonable resource scheduling of DL operators, and other framework interfaces.

Answer to RQ1. (1) DevMuT achieves 71.68% diversity average improvement on the generated models and achieves a higher 81.17% legal rate exceeding baselines by 28.20%. (2) DevMuT can detect three new kinds of defects that baselines cannot detect and shows a higher detection ability that meets the expectations of developers on defect detection.

6.2 RQ2: Analysis on Detected Defects

DevMuT detects 117 defects, and 63 are confirmed; eight are labeled with specific tags like "Main", and 24 defects have been fixed. Throughout the DL model lifecycle, which includes construction, execution, and deployment, we count seven defects (e.g., failure

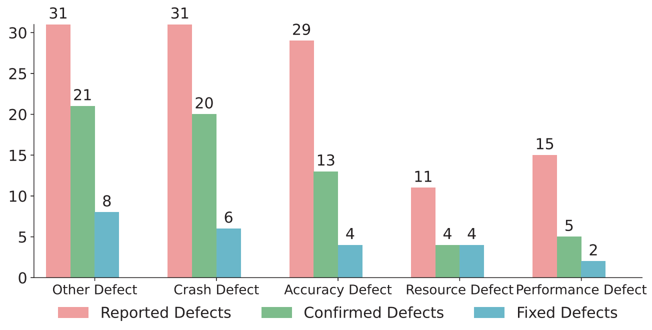


Figure 7: Defects Detected by DevMuT

to initialize the middle layer) in the construction stage, 21 defects (e.g., abnormal loss) in the training stage, 12 defects (e.g., inference performance degradation) in the inference stage and 14 defects (e.g., crashes in pre-trained weight load) in the deployment stage among the 63 confirmed defects (the left 9 defects are document and framework functionality defects). We use Figure 7 to show the defects detected by different test oracles: the x axis represents the names of different types of defects; the y axis represents the number of the total detected defects, the confirmed defects, and the fixed defects. The typical cases of each type are shown in the following part.

Performance Defect. DevMuT detects 9 loss defects, 3 inference evaluation defects, and 3 efficiency defects. Firstly, we present a defect related [39] to NAN loss. Specifically, DevMuT mutates "TextCNN" for 20 rounds and detects the loss of MindSpore model changes from "NAN" to "inf" during the first train epoch while the loss of PyTorch model changes from 0.171 to 5.969. We find the output distances of the middle layers across different frameworks range from 1e-1 to 1e3 after training on 200 batches, indicating a defect in the framework interfaces related to weight updates. Developers have confirmed the defect report and replied to fix it. Next, we show two examples of the performance defect [40] in model inference and efficiency defect [41], respectively. The inference eval of the "LeNet-5" from MindSpore remains at 11.35% with the loss hovering between 2.0-2.9 in the first 46 epochs. Then, it begins to increase to 30.60% until it rises to around 99% with the loss in 1e-5-1e-7. However, from the 164th epoch, although the accuracy was still around 98.64%, the loss value changed to -inf, and then the loss value remained unchanged at NAN, with the accuracy plummeting to 9.8%. After the analysis, developers find the implementation of the loss function, i.e., the "CrossEntropyLoss" contains defects. It is fixed in the latest version. Another example of an efficiency defect is "Openpose". The generated model from MindSpore is 45.86% slower than that of PyTorch after 50 mutations. After preliminary analysis, we find the calculation time is mainly spent on the "Flatten" operator. Developers confirm that the "Flatten" operator of MindSpore lacks necessary optimization and is not supported in the current version. This defect will be fixed in future updates.

Resource Defect. Among the 11 defects, they can be subdivided into two types: (1) six are memory leaks caused by resource scheduling mechanisms during training, and (2) five are unreasonable memory allocations for single DL operators. For the first type, we list an example [45] of the "LeNet-5-1" model to present. We find the memory usage rate slowly increases during the execution stage. Specifically, memory usage starts at 0.704GB, reaches 5.696GB after one day, and 9.408GB the next day. This is caused by continuous caching and failure to timely recycle unused memory by the MindSpore decorator "ms.jit". Developers label it as a valuable defect report and decide to fix it with their highest priority. The next example [46] shows a typical case of the second defect type. After 50 mutations of "VGG16-1", we find that the MindSpore model threw a memory allocation exception, while the equivalent PyTorch model normally executed because the "Pad" operator within the model cannot process the input data size. The developers confirmed that it is caused by the wrong implementations of the "Pad" operator on secure memory settings.

Accuracy Defect. DevMuT detects 29 accuracy defects, and 13 of them have been confirmed. Among the confirmed defects, nine are related to NAN outputs, and we list one typical example [1] related to the Ascend hardware. Specifically, the output of "VGG16-1" changes to NAN on MindSpore and PyTorch after six mutation rounds. However, when we reproduce it from GPU v100 to Ascend 910, the output changes from NAN to normal values. Developers detect the defect in the output design of Ascend 910 hardware. This defect is labeled with "Main" and achieves the focus of developers.

Crash Defect. DevMuT detects 12 crash defects during mutation and 19 in execution. Defects in mutation arise from the implementation differences of various framework interfaces, leading to inconsistent results. For example, when the kernel size exceeds the input data size, the "Conv2d" operator in MindSpore can execute normally, whereas in PyTorch, it cannot [12]. Defects exposed in execution are often exposed in complex invoking scenarios. One defect [13] is exposed during the forward progress in model training. For example, "Vit" failed to execute forward propagation after mutations. Developers locate the defect of the optimization mechanism in MindSpore. Since we select the lower optimization degree during execution, it cannot trigger "constant folding" and eliminate the redundant layer that leads to the "slice error" crash.

Other Defects. Except for the four kinds of defects detected by our designed test oracles, we detect 31 defects during our experiments. Specifically, we detect nine document defects about the wrong description of the framework interfaces and existing incomplete functionalities of frameworks.

Answer to RQ2. DevMuT is good at detecting the crash defects exposed in complex invoking scenarios while also detecting new types of detects like resource scheduling defects and abnormal model performance defects, which shows better defect detection ability than existing methods.

6.3 RQ3: Ablation Study on Mutation Guideline

The mutation guideline is designed to determine the selection of seed models and mutation operators for DevMuT and can affect the performance. We conduct an ablation study to evaluate the contribution of the double-Q learning strategy to DevMuT. In particular, we implement two baselines named $DevMuT_r$ and $DevMuT_{mcmc}$ that select mutation operators and seed models by the random strategy and the MCMC strategy [3], respectively. These two strategies are widely applied in existing work [28, 56] and show encouraging performance. We run DevMuT, $DevMuT_{mcmc}$, and $DevMuT_r$ for 100 iterations and analyze the generated models from the perspective of model diversity and the number of legal ones.

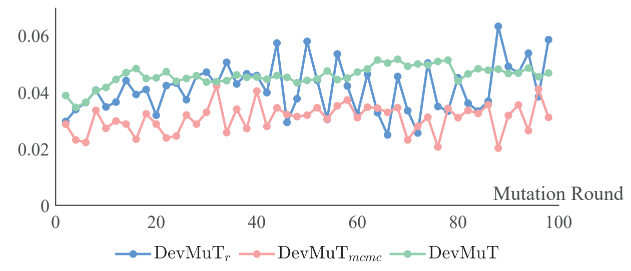


Figure 8: Model Diversity Comparison on Different Strategies

We show the average results for all models in Figure 8. The x axis represents the mutation rounds while the y axis represents the average value of three diversity metrics of each generated model. As shown in Figure 8, $DevMuT_r$ can reach the peak earlier than the other two strategies, but as the mutation deepens (after 20 rounds of mutation), it is prone to falling into "local optima" and often produces illegal models, and the performance is no longer stable; The mutation process of $DevMuT_{mcmc}$ is relatively stable, with the variance on three metrics being the smallest among the three strategies. The generated model with maximum value on three metrics is, on average 15.04% and 14.86% lower than $DevMuT_r$ and $DevMuT_{mcmc}$, respectively. The model generated by DevMuT gradually increases with execution, reaching a peak between 60-80 rounds before slowly decreasing. Compared with the other two strategies, the model generated by DevMuT has better average values of LSC, but its stability is slightly lower than $DevMuT_{mcmc}$. Besides, DevMuT achieves the highest legal rate, 81.17%, and outperforms $DevMutT_{mcmc}$ and $DevMutT_r$ with 73.83% and 51.92% among all the generated models across three strategies, respectively. The average time of DevMuT, $DevMuT_{mcmc}$, and $DevMuT_r$ to generate each model is 95.08s, 97.92s, and 244.917s since $DevMuT_r$ often generate illegal models and need to spend extra time on backtrack to explore previously test input space.

Answer to RQ3. Results show that double-Q learning performs better than the random and MCMC strategy regarding the diversity and legitimacy of generated models and execution efficiency.

7 DISCUSSION

This section discusses DevMuT's performance in efficiency and false positives. Its computational cost includes (1) model mutation and (2) defect detection. For model mutation, the average times for DevMuT, COMET, and Muffin are 95.08, 98.18, and 14.45 seconds, respectively. DevMuT takes longer than Muffin due to its enhanced validation process, which improves legal rates of generated models by 28.20%, improving test effectiveness. For defect detection, DevMuT requires around 45 minutes per model, compared to a few seconds for baselines, but it identifies four new defect types: abnormal loss, low training efficiency, training crashes, and unreasonable resource scheduling that are valuable to developers with eight confirmed as "Serious" by developers. Baseline methods, which focus on accuracy defects during inference, fail to detect these defects. We argue that the execution time of DevMuT is not a concern in practice, especially in industry: (1) the critical nature of detected defects justifies the testing, as undetected defects may cause longer delays; (2) the process is fully automated and can run in backgrounds; (3) advancements in AI accelerators (e.g. GPU [17], NPU [37], and TPU [52]) can rapidly improve efficiency.

To deal with false positives, DevMuT employs nine mutation constraints to avoid and filter illegal models, supplemented by manual inspections for the remaining cases. The first six constraints are designed to guide mutation with the development experience of developers. The second three leverage tactics in identifying and filtering illegal models, reducing the false positive rate to 22.22%. Remaining false positives often include accuracy defects like loss inconsistency or crashes due to incorrect parameters, migration failures, or unreasonable configurations. For loss defects, if DevMuT reports loss inconsistency that does not affect inference, developers reject it. Crashes caused by missing settings are addressed by experienced developers, while other false positives are filtered via manual inspection. Experiments show these measures effectively handle false positives in DevMuT's defect reports.

8 THREATS TO VALIDITY

Internal Validity. It mainly comes from the correctness of implementation. To address this, three authors alternately conduct cross-validation to ensure the implementation is correct and compare the original DL models' equivalence across different frameworks.

External Validity. It mainly comes from the benchmark: DL frameworks and models may affect the generality of our study. To address this, we plan to encompass other frameworks like Jittor [26] and TensorFlow [50] to enhance our study's breadth. Besides, future research will expand to new model tasks, such as audio recognition.

Construct Validity. It mainly comes from the experiment parameter and interview guideline settings in the experiments and may affect the reliability of experimental results. For the parameter settings, we follow the recommendation of previous studies [18, 28] when running baseline methods. Besides, we set the threshold of the test oracles and the parameters of DevMuT based on the experimental results to reduce the false positives and achieve better performance. For the interview, we send the collected interview results to the interviewees for confirmation.

Conclusion Validity. It mainly comes from the rationality of evaluation metrics and may affect the effectiveness of our conclusions in real scenes. To address this, we follow previous studies' metrics for evaluation and conduct manual inspections to exclude false positives and enhance robustness.

9 CONCLUSION

In this study, we propose a mutation-based DL framework testing method DevMuT based on developers' expertise. To detect defects that are more relevant to real scenes, DevMuT designs seven mutation operators and nine extra constraints to simulate the common operations of developers on DL models and expand the detection scope to more stages of the model lifecycle (i.e., model training and inference). DevMuT further utilizes the double-Q learning strategy to guide the mutation. Our comprehensive evaluation shows the effectiveness and efficiency of our proposed method.

ACKNOWLEGEMENT

We would like to thank the anonymous reviewers for their insightful comments. This work is supported partially by the National Natural Science Foundation of China (61932012, 62372228, 62141215), Science, Technology, and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

REFERENCES

[1] accuracy defect1 Accessed: 2023. accuracy defect1. https://gitee.com/mindspore/mindspore/issues/I6B308.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem.

[3] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng (Eds.). 2011. *Handbook of Markov Chain Monte Carlo.* Chapman and Hall/CRC.

[4] Cyrus D Cantrell. 2000. *Modern mathematical methods for physicists and engineers.* Cambridge University Press.

[5] 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.

[6] Xiang Chen, Yanzhou Mu, Yubin Qu, Chao Ni, Meng Liu, Tong He, and Shangqing Liu. 2020. Do different cross-project defect prediction methods identify the same defective modules? *Journal of Software: Evolution and Process* 32, 5 (2020), e2234.

[7] Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, and Chao Ni. 2019. Software defect number prediction: Unsupervised vs supervised methods. *Information and Software Technology* 106 (2019), 161–181.

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016). https://api.semanticscholar.org/CorpusID:1335572

[9] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 793–804.

[10] CNTK Accessed: 2023. CNTK. https://www.tutorialspoint.com/microsoft_cognitive_toolkit/index.htm.

[11] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games.* https://api.semanticscholar.org/CorpusID:16724115

[12] crash defect1 Accessed: 2023. crash defect1. https://gitee.com/mindspore/mindspore/issues/I6NS9V.

[13] crash defect2 Accessed: 2023. crash defect2. https://gitee.com/mindspore/mindspore/issues/I7HZT8.

[14] devmuttest Accessed: 2023. DevMuttest. https://e.gitee.com/mind_spore/repos/mind_spore/net-sv/sources.

[15] William S. Faught. 1986. Applications of AI in Engineering. *Computer* 19, 07 (1986), 17–27.

[16] Mohamed Fayad and Douglas C Schmidt. 1997. Object-oriented application frameworks. *Commun. ACM* 40, 10 (1997), 32–38.

[17] GPU Accessed: 2024. NVIDIA GPU. https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/.

[18] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering.* 1418–1430.

[19] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 486–498.

[20] Hado Hasselt. 2010. Double Q-learning. *Advances in neural information processing systems* 23 (2010).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[22] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1158–1161.

[23] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. Deepcrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 67–78.

[24] JAX Accessed: 2018. JAX. https://jax.readthedocs.io/en/latest/.

[25] Enio G Jelihovschi, José Cláudio Faria, and Ivan Bezerra Allaman. 2014. ScottKnott: a package for performing the Scott-Knott clustering algorithm in R. *TEMA (São Carlos)* 15 (2014), 3–17.

[26] Jittor Accessed: 2022. Jittor. https://cg.cs.tsinghua.edu.cn/jittor/.

[27] Keras Accessed: 2020. Keras. http://keras.io/.

[28] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen*, and Shing-Chi Cheung*. 2023. COMET: Coverage-guided Model Generation For Deep Learning Library Testing. *ACM Transactions on Software Engineering and Methodology* (2023).

[29] Jiawei Liu, Yuheng Huang, Zhijie Wang, Lei Ma, Chunrong Fang, Mingzheng Gu, Xufan Zhang, and Zhenyu Chen. 2023. Generation-based Differential Fuzzing for Deep Learning Libraries. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–28.

[30] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Neuri: Diversifying dnn generation via inductive rule inference. *arXiv preprint arXiv:2302.02261* (2023).

[31] Weisi Luo, Dong Chai, Xiaoyue Ruan, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based fuzz testing for deep learning inference engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 288–299.

[32] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE).* IEEE, 100–111.

[33] MindSpore Accessed: 2020. MindSpore. https://www.mindspore.cn/.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[35] Motivating Example Accessed: 2023. Motivating Example. https://github.com/pytorch/pytorch/issues/93357.

[36] Meinard Müller. 2007. Dynamic time warping. In *Information retrieval for music and motion.* Springer, 69–84.

[37] NPU Accessed: 2024. Apple Neural Engine. https://www.apple.com.

[38] Ziad Obermeyer and Ezekiel J Emanuel. 2016. Predicting the future—big data, machine learning, and clinical medicine. *The New England journal of medicine* 375, 13 (2016), 1216.

[39] Performance defect1 Accessed: 2023. Performance defect1. https://gitee.com/mindspore/mindspore/issues/I8ICVU.

[40] Performance defect2 Accessed: 2023. Performance defect2. https://gitee.com/mindspore/mindspore/issues/I6AQUE.

[41] Performance defect3 Accessed: 2023. Performance defect3. https://gitee.com/mindspore/mindspore/issues/I73B3G.

[42] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 1027–1038.

[43] PyTorch Accessed: 2019. Pytorch. https://pytorch.org/.

[44] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015).

[45] Resource defect1 Accessed: 2023. Resource defect1. https://gitee.com/mindspore/mindspore/issues/I6AQUE.

[46] Resource defect2 Accessed: 2023. Resource defect2. https://gitee.com/mindspore/mindspore/issues/I73Z8B.

[47] sharelink Accessed: 2024. Data Avaiable. https://github.com/Wayne-coding/DevMuT.

[48] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[49] Min Sun, Ali Farhadi, and Steve Seitz. 2014. Ranking domain-specific highlights by analyzing edited videos. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13.* Springer, 787–802.

[50] Tensorflow Accessed: 2019. Tensorflow. https://www.tensorflow.org/.

[51] teslanews Accessed: 2023. teslanews. https://www.tesladeaths.com/.

[52] TPU Accessed: 2024-04-27. Google TPU. https://cloud.google.com/tpu.

[53] Petra Vidnerová and Roman Neruda. 2016. Evolutionary generation of adversarial examples for deep and shallow machine learning models. In *Proceedings of the The 3rd Multidisciplinary International Social Networks Conference on SocialInformatics 2016, Data Science 2016.* 1–7.

[54] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. 2018. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience* 2018 (2018).

[55] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering.* 798–810.

[56] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 788–799.

[57] XIAO Xiong. 2023. Fundamentals of speech recognition. (2023).

[58] Yinglong Zou, Haofeng Sun, Chunrong Fang, Jiawei Liu, and Zhenping Zhang. 2023. Deep learning framework testing via hierarchical and heuristic model generation. *Journal of Systems and Software* 201 (2023), 111681.