Deep Learning Library Testing: Definition, Methods and Challenges

XIAOYU ZHANG, School of Cyber Science and Engineering, Xi'an Jiaotong University, China
WEIPENG JIANG, School of Cyber Science and Engineering, Xi'an Jiaotong University, China
CHAO SHEN*, School of Cyber Science and Engineering, Xi'an Jiaotong University, China
QI LI, Institute for Network Sciences and Cyberspace, Tsinghua University, China
QIAN WANG, School of Cyber Science and Engineering, Wuhan University, China
CHENHAO LIN, School of Cyber Science and Engineering, Xi'an Jiaotong University, China
XIAOHONG GUAN, School of Cyber Science and Engineering, Xi'an Jiaotong University, China

Recently, software systems powered by deep learning (DL) techniques have significantly facilitated people's lives in many aspects. As the backbone of these DL systems, various DL libraries undertake the underlying optimization and computation. However, like traditional software, DL libraries are not immune to bugs. These bugs may be propagated to programs and software developed based on DL libraries, thereby posing serious threats to users' personal property and safety. Studying the characteristics of DL libraries, their associated bugs, and the corresponding testing methods is crucial for enhancing the security of DL systems and advancing the widespread application of DL technology. This paper provides an overview of the testing research on various DL libraries, discusses the strengths and weaknesses of existing methods, and provides guidance and reference for the application of DL library testing methods. This paper first introduces the workflow of DL underlying libraries and the characteristics of three kinds of DL libraries involved, namely DL framework, DL compiler, and DL hardware library. Subsequently, this paper constructs a literature collection pipeline and comprehensively summarizes existing testing methods on these DL libraries to analyze their effectiveness and limitations. It also reports findings and the challenges of existing DL library testing in real-world applications for future research.

CCS Concepts: • **Security and privacy → Software security engineering**; *Systems security*; • **Software and its engineering → Software libraries and repositories**.

Additional Key Words and Phrases: Deep Learning Testing, Deep Learning Library Testing, Deep Learning, Software Testing

ACM Reference Format:
Xiaoyu Zhang, Weipeng Jiang, Chao Shen, Qi Li, Qian Wang, Chenhao Lin, and Xiaohong Guan. 2025. Deep Learning Library Testing: Definition, Methods and Challenges. 1, 1 (February 2025), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the development of deep learning (DL) techniques, the DL systems that are driven by DL models have been applied in many fields, providing societal benefits in areas like image recognition [36], self-driving [29], and natural language processing [67]. As the backbone of DL systems, the security and safety of the underlying DL library have received more and more attention. The DL library (*e.g.*, PyTorch) is responsible for performing specific computations for training

*Corresponding author

Authors' addresses: Xiaoyu Zhang, zxy0927@stu.xjtu.edu.cn, School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China; Weipeng Jiang, lenijwp@stu.xjtu.edu.cn, School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China; Chao Shen, chaoshen@mail.xjtu.edu.cn, School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China; Qi Li, qli01@tsinghua.edu.cn, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China; Qian Wang, qianwang@whu.edu.cn, School of Cyber Science and Engineering, Wuhan University, Wuhan, China; Chenhao Lin, linchenhao@xjtu.edu.cn, School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China; Xiaohong Guan, xhguan@mail.xjtu.edu.cn, School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China.

2025. Manuscript submitted to ACM

Table 1. Comparison Between Our Survey and Related Papers.

Paper	Year	Covered Component					Covered Property		Discussion Across Components
		Framework	**Compiler**	**Hardware Library**	*Models*	*DL-based Software*	**Correctness**	**Efficiency**	
Braik *et al.* [7]	2018	✓			✓	✓	✓	✓	
Zhang *et al.* [141]	2020	✓			✓	✓	✓	✓	
Li *et al.* [70]	2020		✓				✓		
Zhang *et al.* [145]	2022				✓	✓	✓	✓	
Ma *et al.* [89]	2023	✓					✓		
Hu *et al.* [41]	2023				✓		✓		
Ji *et al.* [44]	2023	✓					✓		
Our Survey	2024	✓	✓	✓			✓	✓	✓

or inference DL models and implementing optimized operations on DL hardware. Developers are highly relying on DL libraries to develop software and systems containing DL components. Tesla relies on PyTorch which is one of the most popular DL libraries to solve problems related to the self-driving domain [115]. TensorFlow, which is another popular DL library, undertakes many important business tasks of Google, Intel, and other companies [119].

Similar to traditional software, the DL library also has bugs, which can be propagated to the DL systems developed upon these libraries and cause the systems to make erroneous predictions, generate huge overhead, and even crash [103, 131], thereby jeopardizing user property and personal safety. For example, in recent years, the self-driving systems developed by Tesla and Uber have experienced abnormal behaviors during driving and eventually led to fatal crashes [55, 114], which further arouse people's concerns about the security and safety of DL systems and the underlying libraries.

At present, researchers have proposed a series of tools and methods [21, 103, 131] to discover bugs such as crashes, overflows, and numerical errors on the DL libraries represented by DL frameworks (*e.g.*, TensorFlow, PyTorch), aiming to guarantee the security and usability of the DL system built upon these libraries. How to deeply understand the bugs of the underlying libraries of the DL system and design testing methods for these libraries needs to be solved urgently and is of great significance. Although researchers have proposed various DL library testing methods, there are still many challenges. Firstly, there are various types of DL libraries, including DL frameworks, DL compilers, etc. Different DL libraries undertake different calculation and optimization functions, and there are significant differences between their inputs, outputs, and implementations. As a result, existing testing methods are diverse and highly targeted to specific libraries, leading to a lack of general and systematic testing methods for different DL libraries and evaluation benchmarks for different testing methods. Furthermore, existing research has a limited understanding of DL library bugs and mainly focuses on crashes and numerical errors. They rarely evaluate and identify other bugs in DL libraries (*e.g.*, performance bugs), which limits the effectiveness of these methods. Therefore, it is significant to conduct induction, analysis, and discussion on the existing research in the field of DL library testing to find limitations and provide guidance for subsequent research directions in related fields.

However, existing surveys on the DL library testing are limited. We have compared related surveys in Table 1 and observed that existing works mainly focus on testing and repairing DL models or software built upon the models [7, 28, 41, 91, 145] (italic columns in Table 1). Even though some works involve testing DL libraries, they typically only cover certain components of various libraries and fail to provide a detailed introduction, analysis, and discussion across testing methods for different DL library components. For example, Zhang *et al.* [141] and Ji *et al.* [44] discussed testing methods for DL frameworks, while Li *et al.* [70] focused on the workflow and performance of DL compilers. To fill this gap, this paper focuses on the bugs that affect the functionality correctness and efficiency of DL libraries and comprehensively summarizes corresponding testing methods for three DL library components, namely ❶ **DL frameworks** that execute

the DL program and construct DL model, ❷ **DL compilers** that compile and translate DL models into optimized operators, and ❸ **DL hardware libraries** that map operators to the hardware to perform calculations. This paper further analyzes the advantages and limitations of existing methods and delves into the challenges and future research opportunities in DL library testing. Our goal is to provide a set of practical findings to promote the development of DL library testing research, thereby ensuring the security and reliability of DL systems built on these libraries in real-world scenarios. Note that the scope of this paper includes the testing methods on three DL library components, but does not include the DL model and DL program testing. The main contribution can be summarized as follows:

- We present the first comprehensive and detailed survey of testing methods for various libraries in the DL workflow, including DL frameworks, compilers, and underlying hardware libraries. Our work expands and enhances existing DL testing surveys, which only focus on the specific DL component (*e.g.*, DL framework) and lack an in-depth analysis and discussion of bugs and testing methods for different DL library components.
- We propose a novel taxonomy in three test components to provide an accessible overview of works that focus on libraries at different stages in the DL workflow, namely DL framework, DL compiler, and DL hardware library, as shown in Fig. 1. For each stage, we comprehensively summarize and present existing work according to their testing techniques and provide an in-depth analysis at the end to characterize some critical problems.
- We provide a set of practical findings based on the literature survey and outline the main challenges and future research directions of DL library testing, aiming to promote the development of DL software security and safety.

This paper is organized as follows. §2 describes the workflow and three key components of the DL underlying libraries and other preliminary knowledge. §3 explains the methodology of our paper, the review questions to be answered, and the literature collection process. Based on the review questions, §4, §5, §6 and §7 introduce various testing methods for different DL library components, and discuss the advantages and limitations of these methods. Then §8 provides practical findings from our survey and discusses existing challenges in DL library testing, and §9 concludes this paper.

2 PRELIMINARY

2.1 DL Model

A machine learning (ML) model is a parameterized function $F : X \mapsto Y$, where $x \in X$ is an $m-$dimensional input and $y \in Y$ is the corresponding output label. As a family of the ML model, the DL model is a neural network typically composed of several connected layers. An n-layered DL model can be represented as $F_\theta = l_1 \circ l_2 \circ \cdots \circ l_n$, where l represents a layer and θ indicates the learnable parameters in the DL model. The developers first need to train the DL model on the given data and update the model parameters θ in the training progress. Then, in the inference process, the trained DL model can predict the output for the given input (*e.g.*, an image, or a sentence).

The training process of a DL model consists of the *forward propagation* stage and the *backward propagation* stage, and the model inference process only uses the former. The forward propagation stage calculates model output $F_\theta(x_i)$ based on the input tensor x_i and initialized model parameter θ. The backward propagation stage evaluates the difference between $F_\theta(x_i)$ and the ground truth label y_i by a loss function $\mathcal{L}(F_\theta(x), y)$ and updates model parameters θ to minimize the value of $\mathcal{L}$. The forward propagation and backward propagation stages will be repeated until the training reaches the predetermined stopping criteria. In the DL program, developers call the APIs provided by the DL framework to build and train a DL model and each layer l_i in the model can be directly constructed by one or several DL framework APIs.

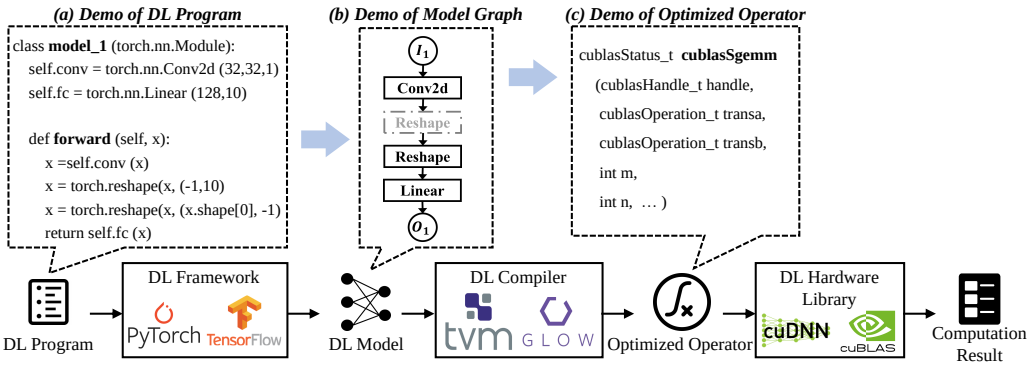

Fig. 1. Overarching Workflow of the DL system and DL program

2.2 DL Library Components

DL libraries implement the abstract DL model based on the program and perform specific operations and optimizations on the underlying hardware to obtain computation results of DL model training and inference. Executing the DL program and building a DL model on the DL underlying libraries mainly involves three components, namely the DL framework, DL compiler, and DL hardware library. The three components are closely related, cooperate with each other, and together form the DL workflow, as shown in Fig. 1. The code in dotted boxes presents the demo inputs of each component. Developers first need to call the DL framework APIs to construct a DL program. As shown in Fig. 1 (a), the DL program calls the DL library APIs to define the convolutional layer (`torch.nn.Conv2d`), linear layer (`torch.nn.Linear`), and reshape operation (`torch.reshape`). DL framework then executes the DL program and constructs the corresponding DL model. Fig. 1 (b) shows the corresponding DL model graph. Subsequently, the DL compiler translates the input model into intermediate representations (IRs) and performs optimization, and outputs the optimized operators and code. During the optimization process, the DL compiler eliminates the 'Reshape' in the gray dashed box of Fig. 1 (b) due to functional redundancy. It also reformulates 'Conv2d' as the matrix multiplication operator 'cublasSgemm' in Fig. 1 (c). Finally, the DL hardware library (*e.g.*, cuDNN) accepts the output of the compiler and maps calculations to the DL hardware to perform calculations and obtain the results.

2.2.1 DL Framework. DL frameworks are high-level DL libraries that provide a user-friendly interface for users to conveniently design, train, and deploy DL models. Industry and academia have proposed various DL frameworks, including TensorFlow [4], PyTorch [101], etc. TensorFlow supports a variety of program languages (*e.g.*, C++, Python, and Go) and is currently one of the most popular DL frameworks. PyTorch is rewritten and optimized based on the DL framework Torch. Nowadays, PyTorch and TensorFlow have active developer communities and are the most commonly used test objects in DL framework testing research. Note that the libraries that extend the functionality of DL frameworks or empower DL development are not within the scope of this paper, such as ONNX which provides an open-sourced format for DL models [93] and AutoML pipelines that reduce manual efforts in designing models [51].

Like traditional software, DL frameworks provide numerous flexible APIs to call functions, perform operations, and construct neural networks. Taking PyTorch [101] as an example, its API includes performing basic matrix operations (*e.g.*, `torch.mul` for multiply operation), calculating loss functions (*e.g.*, `torch.nn.MSELoss` for measuring mean squared error), and building models layers (*e.g.*, `torch.nn.Conv2d` for convolution layers). After the developer calls these APIs in the program, the DL framework will build the corresponding abstract DL model.

2.2.2 DL Compiler. To reduce the burden of manually optimizing DL models on various DL hardware (*e.g.*, TPU) and hardware libraries (*e.g.*, cuDNN), researchers have developed the DL compiler [70]. It takes the abstract model described by the DL framework as input, and then automatically optimizes and generates operators and codes as output to ensure that the DL hardware library can efficiently execute calculations of the DL model. Therefore, DL compilers are generally closely related to and work together with the DL framework that builds abstract DL models. The currently popular DL compilers include Glow [107], TVM [13], etc. Glow is designed to implement state-of-the-art optimizations and generate code for neural network graphs. TVM provides graph-level and operator-level optimizations for DL models, whose optimized performance on some hardware is competitive with state-of-the-art hand-tuned libraries.

Similar to traditional compilers, DL compilers implement a layered design, which mainly consists of the compiler frontend and the compiler backend. The intermediate representation (IR), as the abstract of the program, exists in both the frontend and the backend [70]. The front end converts the DL model from the DL framework into a computation graph and optimizes the graph with various methods (*e.g.*, reducing redundancy). In this process, IR is mainly used to express DL models, construct the control flow and dependencies between operators and data, etc. For the computation graph, the backend performs hardware-specific optimizations by leveraging third-party tools and customizing compilation passes based on prior knowledge. Finally, DL compilers convert the DL model into operators and code which can be used to perform calculations on the DL hardware and hardware libraries, optimizing inference execution speed and memory usage.

2.2.3 DL Hardware Library. Researchers have designed a variety of DL hardwares, such as CPU, GPU, and TPU, to apply DL techniques in different scenarios. To adapt to given DL hardware and map the computation to DL hardware efficiently, researchers have developed a series of DL hardware libraries (*e.g.*, cuDNN, cuBLAS) that implement optimized linear algebras, matrix multiplication, DL operators, etc. For example, cuDNN [16] is a DL accelerate library developed by NVIDIA, which is used to achieve high-performance computing on its developed GPU (*e.g.*, RTX3090). In addition, cuBLAS provides a basic linear algebra library for the computation on GPU.

DL hardware libraries select appropriate algorithms for different hardware and deployment environments, thus they can realize specific calculations on DL hardware. They also call different operators according to the data type to speed up the operation of the model on the given data. For example, cuBLAS implements various matrix-matrix multiplication operators (*e.g.*, cublasHgemm, cublasSgemm) to map the calculations of different data types to the hardware.

2.2.4 Uniqueness, Relationship, and Similarities. **Uniqueness**: The three DL components separately have their unique characteristics. Firstly, different components are in different abstraction levels of DL workflow and the levels decrease from frameworks to hardware libraries. Moreover, to complete different tasks, they also have unique structural designs. DL frameworks typically implement modular interfaces for different stages of model execution (*e.g.*, forward propagation, and backward propagation). DL compilers focus on optimizing and generating operators. Similar to traditional compilers, DL compilers use a layered design, including frontend and backend to generate and optimize IRs. The design of DL hardware libraries is related to their usage scenarios, hardware, and tasks, and there is no general structural design. **Relationship**: Although DL frameworks, DL compilers, and DL hardware libraries have unique designs and abstraction levels, they have close relationships with each other and collaborate to form a DL workflow. Among them, DL frameworks are user-centric and provide high-level abstractions for deep learning tasks. Developers use DL framework APIs to design and construct DL models. The DL framework relies on DL hardware libraries at the lowest level to execute specific computations of DL models. DL hardware libraries provide highly optimized routines for specific hardware architectures. DL compilers at the intermediate level bridge the gap between high-level frameworks and low-level

hardware libraries. It optimizes the DL models generated by DL frameworks to utilize DL hardware libraries for more efficient computation. **Similarities**: In terms of similarity, all three DL library components share the same goal of enabling efficient and convenient DL operations. In addition, with the rapid development of DL techniques, all three DL library components are evolving rapidly, which has laid hidden dangers and bugs in the functional correctness and efficiency of their implementation.

2.3 DL Library Testing

DL library testing aims to discover flaws and errors in the DL libraries. The DL library bug is essentially a kind of software bug. Referring to the prior research [1, 141], we define the behavior that the actual function of the DL library does not meet the requirements and specifications as a DL library bug.

Definition 2.1 (DL Library Bug). A DL library bug refers to any imperfection or deficiency in a DL library that causes the actual function performed by the DL library to fail to meet the expected requirements or specifications.

Based on the above definition of DL library bugs, we define DL library testing as follows.

Definition 2.2 (DL Library Testing). DL library testing refers to any activity designed to discover and identify DL library bugs.

Note that this paper focuses on the test methods that detect functional and performance bugs on DL libraries. In the literature collection, we have observed that existing research mainly tests and discovers bugs, with few works exploring security vulnerabilities and weaknesses. We provide a discussion in §8. In addition, the scope of this paper does not include DL program testing and DL model testing. The former aims to discover and repair errors in DL programs, rather than bugs in the DL library called by the programs [130, 147], while the latter focuses on the security properties (*e.g.*, robustness) and problems of the DL model itself [145].

3 RESEARCH METHOD

In this paper, we followed the methodology of the Systematic Literature Reviews (SLR) to conduct research, which is proposed by Kitchenha [58, 59] and widely adopted in software engineering. Guided by the methodology, we engaged in the SLR through the following three phases:

(1) Planning: In this initial phase, the necessity for conducting a systematic literature review on Deep Learning Library Testing is established, and the goals and review questions of the review are clearly defined.
(2) Conducting: This phase involves the construction of a pipeline for searching relevant literature, followed by a step-by-step process of selection and quality assessment, culminating in the formation of a final pool of papers.
(3) Reporting: A detailed and comprehensive report tailored to the intended audience.

In this section, we will first introduce the specific processes of Planning (in §3.1) and Conducting (in §3.2). The detailed reporting results will be elaborated upon in subsequent sections.

3.1 Planning

To plan the review process, it is necessary to define our research goals and extract the research questions. Our research goals are as follows.

(1) Goal 1: Collect state-of-the-art DL library testing methods and analyze the characteristics of DL library testing.

Table 2. Review Questions (RQs) in Our Survey

Goal 1: Characteristics of DL Library Testing	
RQ 1.1	Which test properties and bugs are focused on in DL library testing?
RQ 1.2	What are the commonly used testing techniques for DL libraries?
RQ 1.3	What are the characteristics and differences between the testing on different DL library components?
Goal 2: Testing Methods on Different Components	
RQ 2.1	How do existing work design testing methods for DL frameworks?
RQ 2.2	How do existing work design testing methods for DL compilers?
RQ 2.3	How do existing work design testing methods for DL hardware libraries?
Goal 3: Comparison Between Testing Methods	
RQ 3.1	What are the strengths and weaknesses of existing DL framework testing methods?
RQ 3.2	What are the strengths and weaknesses of existing DL compiler testing methods?
RQ 3.3	What are the strengths and weaknesses of existing DL hardware library testing methods?
Goal 4: Exploration and Discussion	
RQ 4.1	What are the findings from our survey?
RQ 4.2	What are the challenges in existing DL library testing?

(2) Goal 2: Categorize the testing of DL libraries based on distinct components within the libraries and provide a mapping of the existing testing methods based on the various software testing techniques employed.
(3) Goal 3: Analyze and identify the strengths, weaknesses, and limitations of these different testing methods.
(4) Goal 4: Explore practical implications and findings in real-world scenarios and identify the challenges faced by existing testing methods, thereby delineating potential avenues for future investigation.

We then further developed a set of Review Questions (RQs) for each of the defined goals to analyze three aspects in-depth. The RQs are reported in Table 2.

3.2 Conducting

This section introduces the research methods of this paper. We first constructed a pipeline to search for relevant literature, ensuring comprehensive coverage of advanced related literature. Subsequently, we conducted a step-by-step selection and quality assessment process, culminating in the formation of a final pool of papers. To conduct the review, we took the following steps to search and collect the relevant papers. The overall process is shown in Fig. 3.

• Search String: To encompass the pivotal themes of our review, we meticulously crafted a search string that integrates keywords pertinent to deep learning libraries, compilers, frameworks, and associated bugs. This string is articulated in a syntax compatible with common search engines and databases, ensuring comprehensive coverage of relevant literature. The refined search string is delineated as follows:

```
(("Test" OR "Testing" OR "Fuzzing") AND ("DL Library" OR "Deep Learning Library" OR "DL
    Compiler" OR "Deep Learning Compiler" OR "DL Framework" OR "Deep Learning Framework"
    OR "DL Operator" OR "Deep Learning Operator")) OR "Deep Learning Library Bugs"
```

The search string consists of two major components connected by an 'OR' operator. The first component focuses on testing techniques for DL libraries, which is the primary scope of our survey. We use 'Test OR Testing' as primary keywords since they effectively capture various testing methodologies, including metamorphic testing, mutation testing, and differential testing. We specifically add 'Fuzzing' as an additional keyword because some fuzzing-specific works might not be explicitly labeled with general testing terms. Then we use 'AND' to connect it with keywords such as 'DL Library' and 'DL Compiler' to ensure that these testing methods are relevant to DL library testing. The second

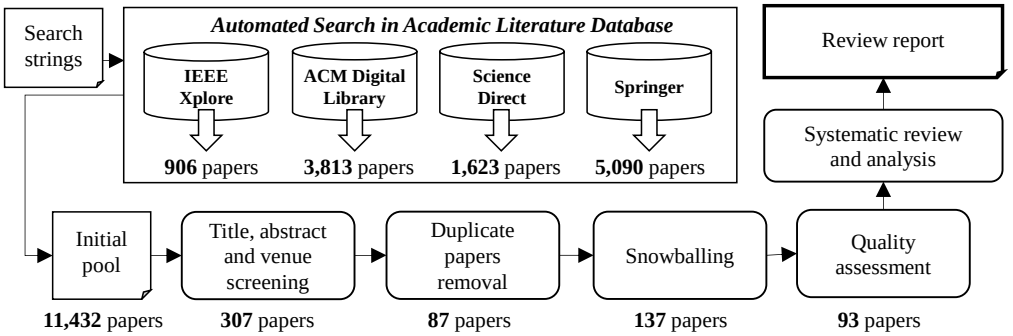

Fig. 3. Process of Literature Selection

component, 'Deep Learning Library Bugs', ensures the inclusion of broader empirical studies and bug analysis papers that might not explicitly mention testing methodologies. Through this carefully designed search strategy, we ensure comprehensive coverage of both testing techniques specific to DL libraries and relevant empirical studies, thereby establishing a solid foundation for our systematic literature review.

- Search Scope: With our search string, we conducted an automated search across four widely recognized databases: IEEE Xplore, ACM Digital Library, Elsevier Science Direct, and Springer. These databases were selected for their comprehensive coverage of published and recent papers in the fields of software engineering and computer science. We collected the papers whose publication date is before September 1, 2024, in search.
- Title, Abstract, and Venue Screening: We conducted a meticulous evaluation of the titles and abstracts to identify papers that were unequivocally relevant to the specific themes of our review. To uphold the quality and credibility of the selected literature, we prioritized papers published in prestigious and highly regarded conferences (e.g., ICSE, ASE, FSE, OOPSLA) and journals (e.g., TOSEM, TSE) within the software engineering and program languages domains.
- Duplicate papers removal: We removed duplicated entries collected from different sources to ensure a clean and unique set of papers.
- Snowballing: Building upon the papers already collected and screened, we employed a forward and backward snowballing approach. This strategy involved tracing the references cited by the initial set of papers (forward snowballing) as well as the papers that cited them (backward snowballing).
- Quality assessment: To mitigate the potential biases introduced by low-quality studies and to guide readers in discerning the reliability of conclusions, we conducted a rigorous quality assessment of the included papers. Specifically, we invited two co-authors with expertise in both the fields of Software Engineering (SE) and Artificial Intelligence (AI) to entirely read the papers and assess the relevance, clarity, validity, and significance of the included papers. For any discrepancies in assessment results, we invited a third co-author to moderate the discussion and solve the differences. This process aimed to ensure that only studies of high methodological quality were incorporated into our review.

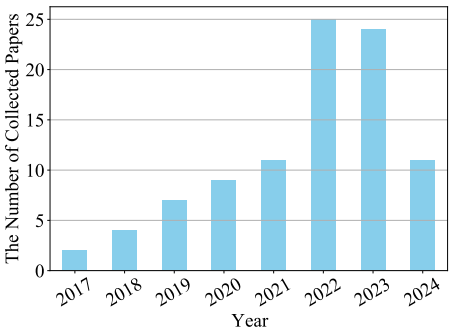

Fig. 2. Number of Relevant Papers Per Year.

3.3 Final Pool of Papers

Employing the above comprehensive search strategy, our initial automated query across multiple academic literature databases yielded a substantial corpus of 11,432 papers, with contributions from IEEE Xplore (906), ACM Digital Library (3,813), Elsevier Science Direct (1,623), and Springer (5,090). Through a meticulous screening process on titles, abstracts, and publication venues, we distilled this collection to 307 papers aligned with our review objectives. By removing duplicates and utilizing snowballing techniques, 137 papers entered our quality assessment, and 93 of them passed the manual assessment and formed our final paper pool. This carefully curated set of 93 papers is the robust and reliable foundation for our comprehensive review analysis. Fig. 2 exhibits the distribution of papers per year in the final paper pool. It reveals a consistent upward trend beginning in 2017, underscoring a burgeoning scholarly interest in the field and the growing recognition of the critical importance of testing DL libraries. The marked escalation in research activity not only reflects the rapid maturation of the field but also signals a heightened awareness among the scientific community of the imperative need for robust, reliable, and secure deep learning frameworks.

In the following sections, we have conducted a review and analysis of DL library testing methods to answer the RQs in Table 2 based on the collected papers.

4 DL LIBRARY TESTING CHARACTERISTICS

4.1 RQ1.1: DL Library Testing Property

The testing property refers to what the DL library testing methods test, which is directly related to the types of detected bugs in tests. It defines where the implementation of the DL library should meet the requirements and expectations. We analyzed 93 collected papers and found that existing DL library testing work mainly focuses on the functionality (*i.e.*, **correctness**) and performance (*i.e.*, **efficiency**) of the DL libraries. They constructed test cases and conducted large-scale experiments to evaluate whether the functionalities of DL libraries are implemented correctly and whether the performance meets expectations. Fig. 4 uses a Venn diagram to show the number of papers involving different testing properties.

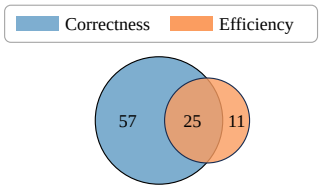

Fig. 4. The Distribution of Test Properties involved in Collected Papers.

Correctness measures the ability of a DL library to correctly perform its functions and complete the given task. Correctness plays a vital role in the application and deployment of DL systems, which ensures the usability and trustworthiness of DL libraries. When the correctness of the DL library is compromised, the intended function cannot be executed, which may cause three types of bugs, namely *status bug*, *numerical bug* and *optimization bug* [10, 110], which could be further exploited to endanger the safety and security of the whole DL system. The status bug refers to the unexpected termination of valid inputs or illegal execution of invalid inputs on the DL library. It includes various crashes, segmentation faults, exceptions, etc. The numerical bug and optimization bug occur when the DL library has incorrect behavior on valid inputs but does not crash. At this time, the DL library will output wrong results and further affect subsequent calculations. The former mainly consists of inconsistent outputs (*i.e.*, inconsistency between expected and actual result) and NaN outputs (*i.e.*, Not A Number, which is caused by overflow in the backend). The latter happens when the DL library (especially the DL compiler) gets erroneous results or middle results in the optimization process, resulting in inequalities between

before and after optimization. Existing testing methods mainly test and validate the correctness of DL frameworks and compilers [21, 103, 136].

Efficiency evaluates the overhead of time, GPU memory, and other performance indicators of the DL library in executing a given task. It determines the cost of large-scale deployment of DL libraries, which is of great significance for DL systems from the perspectives of performance, economics, and the environment. The problem in DL library efficiency leads to the *performance bug*. This kind of bug not only severely compromises the usability of DL libraries, leading to less responsiveness and waste of computational resources, but also puts pressure on both execution costs and the environment and results in a high carbon footprint [50, 98, 102]. However, due to the limitations of testing methods and test oracles, existing work has paid limited attention to such bugs [33, 63, 148]. Among the 93 papers collected in §3.3, only 38.71% involved performance bugs.

4.2 RQ1.2: DL Library Testing Technique

The testing technique determines how the DL library testing methods test. To effectively test different properties and identify bugs in DL libraries, researchers have employed traditional software testing techniques to design a variety of DL library testing methods. Our study of collected papers indicates that the most widely spread techniques include differential testing, fuzz testing, and metamorphic testing. Fig. 5 uses a Venn diagram to show the number of papers using differential testing, fuzz testing, metamorphic testing, and other test and evaluation techniques.

Differential testing is one of the most classic testing techniques in the field of SE, which usually processes the same input on two or more comparable implementations of a given software, uses the outputs between each other as the pseudo test oracle, and utilizes the difference between outputs to reveal potential bugs [92]. As a simple but effective test technique, differential testing has not only achieved excellent results in traditional software testing and verification tasks [30, 127] but also shined in DL library testing [21, 103] (36 of 93 collected methods used). Different implementations of the same operator in different DL libraries or on different devices greatly facilitate the application of differential testing in DL library testing.

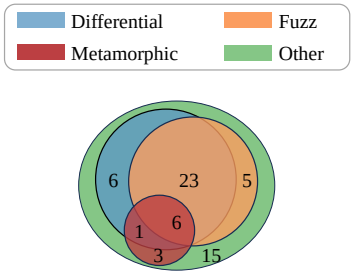

Fig. 5. The Distribution of Test Techniques involved in Collected Papers.

Fuzz testing is a classic software testing technique, which is widely used to automatically detect bugs such as crashes [73, 90]. Fuzz testing typically generates a large number of test inputs and observes whether the target software or system can obtain expected outputs [77]. Therefore, it is usually used to generate valid/invalid test cases in DL library testing, which is currently one of the most popular and effective techniques (34 of 93 collected methods used). According to the test input generation methods, fuzzing can mainly be divided into generation-based fuzzing and mutation-based fuzzing [99]. The former generates test cases and inputs from scratch based on constraints or randomly, while the latter mainly mutates existing inputs to test more potential behaviors of DL libraries. Since fuzz testing can generate a large number of inputs and achieve high API or code coverage in tests, testing methods based on it can often achieve outstanding results in real-world bug detection [19, 137].

Metamorphic testing was proposed by Chen *et al.* [14] in 1998 to construct test oracle. Metamorphic testing constructs a series of metamorphic relations (MRs) that are from the necessary properties of the program under test. An MR describes the expected change in the outputs of a target program when the inputs are changed. Metamorphic testing

can generate a series of test samples and judge whether the functionality of the program is as expected by comparing whether their results conform to the metamorphic relationship [109]. Researchers have designed some DL library testing methods based on the metamorphic testing technique to verify the computation and optimization of DL libraries, and successfully detected real-world bugs in the DL framework and compiler [131, 136] (10 of collected papers).

4.3 RQ1.3: Characteristics of Different DL Library Components Tests

DL library component mentioned in §2.2 points out the application and test objects of various DL library testing methods. In this section, we generally introduce the characteristics of the testing methods on the three DL library components and the differences between them. Note that some libraries (*e.g.*, TensorFlow and PyTorch) in the DL workflow have implemented general functions and APIs that can develop other ML algorithms and models. Most existing testing methods [20, 137] construct test inputs for the various API of the entire DL library to discover bugs. However, some testing methods [34, 103] rely on DL models as test inputs and only cover the library functions called in these DL models. We have provided a detailed introduction and comparison of various testing methods in §5.1.

DL framework testing aims to discover bugs in DL frameworks, which can directly affect the output results and quality of DL models, leading to erroneous results or even crashes of DL systems built on these frameworks. We collected and studied 69 papers on DL framework testing and observed that DL library testing has the following characteristics. ❶ DL framework testing research has the most diverse testing methods and is the most well-developed field in DL library testing. 74.19% of the collected papers in §3.3 studied and tested DL framework bugs. Existing research [10, 43, 48, 140] has comprehensively investigated the characteristics, symptoms, and root causes of DL frameworks bugs through open-source communities (*e.g.*, GitHub and Stack Overflow), providing valuable insights and guidelines for designing testing methods. On this basis, researchers proposed a variety of testing methods and tools based on differential testing, fuzz testing, etc. to discover and identify DL framework bugs [21, 103]. ❷ DL framework testing methods accept manually built or automatically generated DL programs as input. These test cases can be conveniently constructed using the user-friendly API provided by the DL frameworks. ❸ The DL framework testing covers most of the bug types, including status bugs (*e.g.*, crash, segmentation fault), numerical bugs (*e.g.*, inconsistent output, NaN value), and performance bugs (*e.g.*, unexpected time overhead). Early framework testing methods mainly focused on several bugs in the forward and backward propagation stages of DL models [34]. With the development of testing methods, advanced testing methods [20] can cover thousands of APIs of the DL frameworks and discover dozens and even hundreds of bugs, which effectively promotes the development of DL frameworks.

DL compiler testing aims to find DL compiler bugs that cause the DL compiler to generate incorrect code, resulting in unexpected model behaviors [110]. In traditional software, the compiler converts high-level program language (*e.g.*, C++) to low-lever program language (*e.g.*, assembly language) to create and optimize an executable program. SE Researchers have leveraged techniques such as fuzzing [135], metamorphic testing [118], differential testing [152] and machine learning [11, 12] to design a variety of testing methods for these traditional compilers (*e.g.*, GCC, LLVM). Different from traditional compilers, DL compilers convert abstracted DL models into optimized operators and code, which facilitates DL hardware libraries to perform calculations efficiently. We have summarized the following characteristics from the collected DL compiler testing papers. ❶ Similar to DL framework testing, DL compiler testing has also employed techniques such as fuzz testing and mutation testing. However, compared to DL framework testing, the diversity of methods and research in DL compiler testing is less extensive (only collected 12 papers in §3.3). ❷ The DL compiler testing builds models and computation graphs as test input. Unlike DL framework testing that only focuses on parameters and tensor shape in test case generation, DL compiler testing also needs to meet constraints like semantic

specification, posing challenges to efficiently build valid testing cases. ❸ Due to the special architecture of DL compilers, DL compiler testing focuses on three bug-prone stages, namely model loading, high-level IR transformation, and low-level LR transformation stages. The former loads an abstract DL model and transforms it into a computation graph, and the latter two respectively implement optimizations on high-level and low-level IRs. Since specific conversion and transformation are involved, bugs are more likely to occur in the latter two stages [110], therefore, the DL compiler testing methods often pay more attention to optimization bugs in these two stages. Optimization bugs can cause semantic changes and inequality after the compilation process and ultimately incorrect calculation results or middle results [110]. Fig. 6 shows an optimization bug in the Glow compiler, which incorrectly deletes layers (marked in the grey box) during the Dead Code Elimination (DCE) optimization, resulting in optimized operators outputting unexpected results.

DL hardware library testing discovers bugs that can cause the DL model to obtain wrong results and abnormal runtime overhead in the calculation, which is difficult to perceive when executing the DL program or training a DL model. There is relatively little research on DL hardware library testing (13 collected papers). They mainly study and evaluate the functional correctness of DL hardware libraries [120, 124]. Since the DL hardware library is at the bottom of the entire DL workflow, as shown in Fig. 1, it is difficult to generate valid test code to detect potential bugs for these libraries, and little work is done to detect real-world bugs.

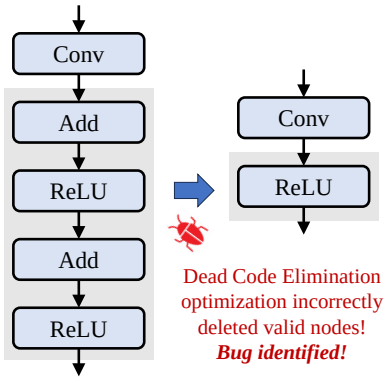

Fig. 6. An Example Optimization Bug [136] on the Glow Compiler.

Summary: DL library testing for different components exhibits significant differences. In terms of testing methods, DL framework testing employs diverse software testing techniques, including differential testing, fuzz testing, etc. DL hardware testing, in contrast, primarily focuses on functional verification, with fewer related tests. In terms of testing inputs, DL framework testing conveniently uses framework APIs to construct DL programs as inputs, while the testing inputs of DL compiler and hardware library testing need to satisfy additional optimization requirements and specifications, posing challenges in effective test case generation. Finally, in terms of detected bugs, DL framework testing covers a variety of status, numerical, and performance bugs in the execution and computation of framework APIs, while DL compilers focus on optimization bugs during the optimization phase, which can lead to incorrect optimization, additional computational overhead, and erroneous results. DL hardware library testing is relatively limited, primarily focusing on the correctness of DL hardware library implementations.

5 DL FRAMEWORK TESTING

In this section, we first introduce the existing empirical studies on bugs and testing of DL frameworks and present DL framework testing methods according to different testing techniques used (RQ2.1). We focus on introducing state-of-the-art and representative methods, which utilize specific testing techniques to break the research limitations at that time, and visually exhibit their descriptions and test results in Table 3. The columns show the methods category, a brief description of each work, the test object, the number of bugs each work reported, and the type of detected bugs. Then, we deeply compare and analyze the advantages and disadvantages of different testing methods and techniques (RQ3.1). Additionally, we conduct experiments to compare five advanced open-source testing methods using different testing techniques in real-world scenarios (PyTorch and TensorFlow frameworks) to support our analysis of different methods.

Table 3. Representative DL Framework Testing Methods

Category	Method Description	Test Object	# Bugs	Bug Type
Empirical Study	Summarized program bug reports and analyzed the challenges of testing and localizing DL bugs [149]	TensorFlow	/	/
	Conducted large-scale study on DL frameworks and summarized bug symptoms and root causes [10]	TensorFlow/DL4J/ PyTorch/MXNet	/	/
Differential Testing	Detected DL framework bugs via inconsistencies between DL frameworks outputs [103]	TensorFlow/ CNTK/Theano	12	status/numerical
	Mutated DL models to explore DL framework behaviors and precisely localized the buggy layer in models [34]	TensorFlow/CNTK/ Theano/PyTorch	26	status/numerical
	Extracted DL equivalence rules from documentation and open source and constructed equivalent graphs to test [125]	TensorFlow/ PyTorch	25	status/numerical
	Leveraged LLMs to generate test code and identified bugs from the different results on different devices [19]	TensorFlow PyTorch	65	status/numerical
Fuzz Testing	Generate test cases for DL framework APIs and fuzz DL framework based on open-sourced data [131]	TensorFlow/ PyTorch	49	status/numerical/ performance
	Extractd constraints from documentations to guide test case generation and fuzz DL framework [137]	TensorFlow/ PyTorch	94	status
	Utilized the behavior of similar APIs as test oracles and fuzz DL framework APIs [21]	TensorFlow/ PyTorch	162	status/numerical
	Designed AI semantics to compute and construct test oracle and generate valid test cases in tests [108]	TensorFlow	14	status
Metamorphic Testing	Designed 11 metamorphic relations to verify the correctness of DL framework functionality [22]	Caffe	/	status/numerical

5.1 RQ2.1: DL Framework Testing Methods

5.1.1 Study on DL Framework Bugs. In the early stages of the development of DL framework testing, developers mainly relied on manual methods to identify and report bugs in DL programs [143, 149]. However, these early studies focused on analyzing programs and projects based on the DL framework and paid limited attention to the bugs in the DL framework itself. To understand the symptoms and characteristics of various bugs, researchers [43] systematically studied over 3,000 bug posts and fixes related to five popular DL frameworks in the open source communities to understand the bug types, root causes, and their effects. Although these studies have summarized the categories of DL framework bugs, they lacked an analysis of the symptoms and root causes of bugs. To fill the gap, Jia *et al.* [47] conducted empirical studies on the TensorFlow framework and found that the most common symptoms of DL framework bugs are functional errors and crashes, which received the most attention from following testing methods [19, 137]. They also found that the most common root causes of bugs are errors related to data processing. Jia *et al.* [48] further analyzed the fix patterns of DL framework bugs and identified 10 fix templates. They have also studied multi-programming language (MPL) bugs in TensorFlow. The code change complexity and communication complexity of MPL bug fixes were usually significantly higher than those of single programming language bug fixes, posing challenges for developers to locate and fix bugs [72]. Their findings have provided valuable insights and guidance for subsequent DL framework testing.

Recently, researchers have conducted more in-depth and subdivided research on DL frameworks and testing methods on the basis of prior work. Some researchers deeply investigated and analyzed the DL framework bug and test oracles on different frameworks and further proposed new testing methods based on their findings [10, 18, 46, 60, 78, 96]. Chen *et al.* [10] conducted a large-scale study on 1,000 bugs on 4 DL frameworks and found 13 types of root causes, including API misuse, numerical issues, etc. Compared to prior studies, they provided a more comprehensive analysis of framework bugs and valuable insights into testing across different frameworks. They further compared the indicators

(*e.g.*, line coverage) of the three existing test methods [34, 103, 129] to evaluate the effect of different testing methods and proposed a preliminary mutation-based testing tools TenFuzz. They took the first step to evaluate the effectiveness of existing DL library testing methods. However, they failed to evaluate more state-of-the-art open-sourced testing tools that cover more framework APIs in tests. The studies of Yang *et al.* [140] and Harzevili *et al.* [35] further included the analysis of DL framework fixing patches. Recently, researchers conducted studies on more subdivided bug types [24, 39, 86, 117, 121, 121]. Tambon *et al.* [117] first focused on silent bugs in Keras and TensorFlow frameworks. Such bugs do not cause crashes or raise exceptions but do affect the DL framework's computation results and performance. They systematically studied the symptoms and root causes of silent bugs and established four levels of impact for silent bugs, with the highest level affecting model output results. Based on their finding, they called for the construction of unit test examples to diagnose silent bugs, providing valuable insights for testing and evaluation of related issues. To fill the research gap of study on performance bugs, Cao *et al.* [8] systematically studied the performance problems in DL frameworks such as TensorFlow and Keras and summarized five types of root causes from the aspects of API usage, model parameter selection, etc. Based on the findings in the empirical study, they proposed and implemented a rule-based static checker, DeepPerf, to detect potential performance problems in DL systems.

Researchers have also paid attention to the DL framework bugs in subdivided deployment scenarios and environments such as distributed systems and JS systems [15, 84, 105] and provided practical insights for DL framework deployment. Aach *et al.* [3] focused on the distributed DL frameworks and studied the performance of ResNet models on PyTorch, Horovod, and DeepSpeed frameworks and different data loaders. They found that using a suitable data loader can significantly accelerate the computation of ResNet models on these DL frameworks. DeltaNN [87] studied the impact of environmental factors such as DL frameworks on the performance of the image recognition model. They have observed that the discrepancy in output labels of the same model on different DL frameworks can reach up to 72% due to the noise introduced into the model weights by the conversion between the different frameworks.

Summary and Analysis. Existing research leveraged interviews and empirical studies to summarize and analyze software bugs in DL frameworks and pointed out potential directions and challenges for designing testing methods for DL frameworks. In the early stage, studies primarily focused on the symptoms and characteristics of bugs in individual popular DL frameworks like TensorFlow. Recent research has devoted additional effort to a detailed analysis of the root causes of bugs on multiple DL frameworks or subdivided software problems such as silent bugs and has provided more detailed and timely observations and findings for tests than earlier studies. Some researchers even further proposed new test tools based on their findings [10, 35]. Although researchers have a deep understanding of various DL framework bugs, there is still a lack of effective and diverse DL framework bug datasets or benchmarks to help developers evaluate and compare existing testing methods.

5.1.2 Differential Testing on DL Framework. To construct test oracles and identify DL framework bugs, researchers leveraged different implementations and devices to construct testing oracles based on the concept of differential testing. [33, 34, 63, 83, 103]. Depending on the generated test cases, the existing differential testing methods can be mainly divided into model-level testing and API-level testing [21].

Model-level Differential Testing. The model-level differential testing usually leverages the different results of a widely-used DL model (*e.g.*, ResNet-50) on different platforms or frameworks to detect bugs [68, 129]. CRADLE [103] is one of the first tools to detect and identify bugs based on the concept of differential testing. Based on Keras [56] which can build and train models on different DL frameworks as backends, CRADLE conducted differential testing on three frameworks(*i.e.*, TensorFlow, CNTK, and Theano) and finally detected 12 bugs. Fig. 7 shows two trigger figures of the

inconsistencies that cause one model to have different prediction results and accuracy on different DL frameworks. CRADLE compared the model layer outputs between multiple DL frameworks and detected such inconsistencies. However, the inconsistent outputs of one layer may further lead to inconsistencies in subsequent layers, therefore this localization method is prone to false positives (FPs) and false negatives (FNs). To break these limitations, Guo *et al.* [34] mutated the parameters of model layers to explore more DL model behaviors and leveraged a causal-testing-based technique to localize buggy layers and reduce the FPs. Although the above testing methods effectively detected bugs in popular DL frameworks like TensorFlow, Theano, and CNTK, they only focused on the model inference process and could not detect bugs in the backward propagation stage. To fill the research gap, Gu *et al.* [32] proposed Muffin, which creatively generated structure information and layer information of the DL model to thoroughly explore possible abnormal behaviors of the model. Then it conducted differential testing between DL frameworks for model training and inference processes and finally detected 39 new bugs. Although model-level differential testing methods obtain outstanding test results, they still have limitations in practice. The major limitation is that they can only explore limited DL model-related APIs and implementations in the framework For example, existing research [131] indicated that LEMON [129] only covered 35 TensorFlow APIs. As a result, these methods cannot comprehensively test the entire framework (including APIs that can be used to develop ML algorithms), limiting the effectiveness of model-level differential testing methods.

API-level Differential Testing. Different from the DL model with dozens of layers generated by model-level methods, the test cases in the API-level method are simple calls or combinations of DL framework APIs. Some test cases even merely call one operator or transformation APIs to calculate or process a set of randomly generated input data. API-level testing eliminates the need to build and mutate DL models, which liberates the test cases from model shape constraints and enables the test to detect potential bugs on various framework APIs. The existing API-level differential testing mainly used three approaches to construct test oracles. ❶ In the early stage, researchers compared APIs implemented by different frameworks to identify bugs on DL frameworks such as TensorFlow and Theano [31, 104]. However, these methods can hardly determine whether the different behaviors in tests come from implementation differences or real bugs in the DL frameworks, which limits the effectiveness of their testing. ❷ Another simple but effective approach is to execute and compare DL APIs and operators on different devices (*e.g.*, GPU and CPU) and observe the differences between the behaviors [19, 20, 144, 146]. Zhang *et al.* [146] tested the precision errors by comparing the behaviors of seven DL framework operators on CPU and GPU. However, their testing method can only construct test cases and explore abnormal behaviors for several DL APIs. To efficiently and automatically generate test code at scale, Wei *et al.* [131] proposed FreeFuzz, the first DL libraries testing method via mining from open source. FreeFuzz first collected code that calls DL framework APIs from API documentation, DL framework test cases, and open-source models. Then, it tracked and extracted the input and parameter constraints of each API from the execution of the collected code and constructed new test cases. In the test, based on the concept of differential testing, FreeFuzz detected bugs by comparing the performance of test cases on different devices (*i.e.*, CPU and GPU). ❸ Differential testing using different frameworks and services only cover limited APIs that have different implementations on different frameworks and devices. To break these limitations and detect more complex DL framework bugs, researchers tried to construct and leverage the equivalence relationship between APIs to design differential tests [21, 138, 139]. Wang

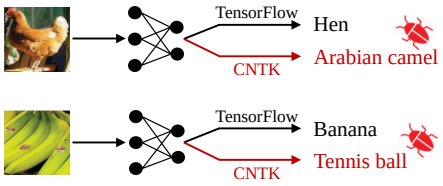

Fig. 7. Trigger Inputs for Inconsistencies Between DL Frameworks

et al. [125] proposed EAGLE, which created equivalent graphs to test DL frameworks. They extracted 16 new DL equivalence rules from DL framework API documentation and non-crash issues in open-source communities and designed elaborate equivalent graphs that use different DL framework APIs, data types, or optimizations to produce identical output given the same input. EAGLE focused on the numerical bug (*i.e.*, inconsistencies) and finally detected 25 bugs on TensorFlow and PyTorch. Deng *et al.* [21] went one step further on the prior work and designed two elaborated equivalence (*i.e.*, status equivalence and value equivalence) and matched thousands of API pairs based on these equivalence relations. It considered the output values and status of APIs in a pair as test oracles for each other and effectively detected a total of 162 status and numerical bugs.

Summary and Analysis. In the early stage of DL framework differential testing, researchers usually compare the output results of DL models or APIs on multiple frameworks to detect potential bugs in the model layers (*e.g.*, LEMON and Muffin). However, these methods have two main limitations. Firstly, they can only cover limited APIs in the framework, which need to be implemented on multiple frameworks and can be triggered by models or other test cases, which reduces the practical value of these testing methods. For example, LEMON only covered 35 TensorFlow APIs [131]. Secondly, they have a high false positive rate in tests. Existing work [79] has pointed out that the false positive rate of these model-level differential testing methods can reach 58%. Since test oracles rely on the implementation of multiple frameworks, it is difficult for developers to confirm whether the different results found in the tests are bugs or merely implementation differences, which affects the effectiveness of the test [34, 103]. To effectively explore more potential bugs in DL frameworks, researchers use equivalent APIs or different devices to construct differential testing scenarios and achieve outstanding results with a much lower false positive rate. However, some advanced methods still reported a high false positive rate of over 30% [19], which required a significant amount of manual effort to verify test results, increasing software maintenance costs. How to reduce false positives while exploring various bugs is a challenge for differential testing research.

5.1.3 Fuzz Testing on DL Framework. According to the methods of generating test input, fuzzing can mainly be divided into generation-based fuzzing and mutation-based fuzzing [99].

Generation-based Fuzz Testing. Generation-based fuzzing generates test inputs randomly or based on the specifications of test inputs. As the test input of DL framework testing, DL programs usually have complex specifications (*e.g.*, specific value ranges of API parameters, input sizes, and dimensions), and the input violating specification will lead to termination in execution, therefore existing work usually follow the specification to generate test inputs. Existing testing proposed various methods to obtain specifications and constraints to guide test case generation [54, 108, 137]. ❶ One direct approach is to directly extract and leverage API constraints from documentation or source code to guide testing [113, 137]. Xie *et al.* [137] designed DocTer that analyzed documentation and extracted DL framework API constraints and further automatically built test cases. The test case generation in DocTer generated both valid and invalid cases according to the constraints and specifications to comprehensively evaluate whether the DL framework has unexpected behaviors. DocTer provided a DL API constraint extraction method and an effective test input generation tool for DL framework testing, which facilitates and promotes the development of other methods [125]. Unfortunately, only part of the DL framework APIs have detailed documentation that could provide constraint information, which limits DocTer's testing of some less commonly used APIs. To overcome the limitations, Shi *et al.* [113] proposed ACETest, which collected DL operators' information from the source code and extracted input validation constraints by analyzing the execution path, thus they could build valid test cases to uncover crashes in DL frameworks. ❷ Researchers also proposed the ML-based method to solve the constraints problem in tests [54, 81]. SkipFuzz [54] used active learning to

learn the input constraints of different library APIs and generated valid test inputs for TensorFlow and PyTorch. Deng *et al.* [20] observed that prior methods typically generated valid inputs and were difficult to help tests cover edge DL library behaviors,. To comprehensively explore the potential behaviors of DL frameworks, they designed zero-shot and few-shot learning to prime LLMs to generate edge cases while ensuring the semantic validity of generated test cases. LLMs' knowledge of DL API calls enabled FuzzGPT to skip the step of collecting API constraints and directly generate effective test code for over 3,000 APIs and detect 108 bugs.

Mutation-based Fuzz Testing. Mutation-based fuzzing usually applies various mutation strategies to introduce small changes to the valid test inputs, so as to explore potential bugs of DL frameworks while ensuring the validity of test cases as much as possible. Some model-level testing methods and tools [10, 34, 68, 94, 97, 112, 129, 134, 144, 146] mentioned above have implemented mutation operators to further explore potential behaviors in DL frameworks. To increase the API coverage of the mutated DL models and explore DL framework behaviors, Zou *et al.* [155] leveraged a mutation-based hierarchical method to generate new models and effectively detected bugs on three frameworks. They implemented two mutation modes, namely random mutation and heuristic mutation. The former randomly modified the layers and operators of the model, and the latter was based on the results of the previously generated model and tended to select mutation operators that can increase the error of the model.

Researchers also proposed various API-level fuzz testing methods [69, 131]. FreeFuzz [131] collected code snippets from open source and implemented 3 categories of 15 mutation strategies on the data type and value to conduct fuzz testing. The mutation strategies include mutating the dimension and datatype of a tensor, mutating the value of a tensor, etc. Based on these strategies, FreeFuzz generated variants of a given test case from open source and further tested and revealed status, numerical and performance bugs for 1158 APIs on DL frameworks. Fig. 8 shows a invalid input on `torch.nn.MaxUnpool2d` that only leads to a crash on the CPU[1]. FreeFuzz generated the invalid input by mutating the value of the input tensor and identified this status bug. Currently, developers have fixed this bug by adding a check for abnormal input values. Researchers have implemented other mutation-based fuzz testing methods based on FreeFuzz [21, 75, 139]. However, limited by the design of mutation strategies, the above methods usually mutate a limited part of the test inputs (*e.g.*, dimension) to generate valid mutated inputs, which cannot explore diverse code structures and potential framework behaviors. To break the limitation, TitanFuzz [19] utilized the ability of LLMs in code generation to empower fuzz testing. It leveraged a Codex model to generate test seeds for a given API and implemented four well-designed mutation operators. These mutation operators masked the parameters of the API, the suffix and prefix of the test code, and the method under test in the seed, respectively, and then used the InCoder model to populate the mask and generate variants at scale. TitanFuzz is currently one of advanced the testing tools, which covers a total of 3,544 APIs on PyTorch and TensorFlow frameworks in experiments.

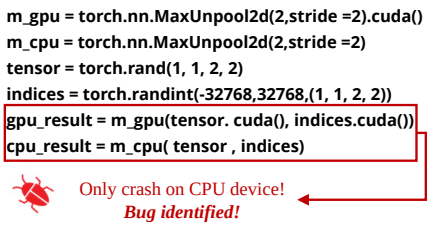

Fig. 8. Invalid Input for `MaxUnpool2d` to Trigger Crash on CPU.

Summary and Analysis. Fuzz methods in the early stage primarily utilize predefined rules and processes to extract constraints or perform mutations from code and documentation, exploring the behaviors of DL framework APIs (*e.g.*, DocTer). However, their test effects are determined by documentation information and mutation operators, limiting the

[1]https://github.com/pytorch/pytorch/issues/68727

exploration of the testing space and making it difficult to deeply uncover bugs in DL frameworks. With the development of ML technologies, advanced ML techniques like LLMs have empowered fuzz testing, enabling tests to generate diverse test cases for thousands of DL framework samples, and exploring potential abnormal behaviors. We have observed that ML-based testing methods (*e.g.*, TitanFuzz) typically achieved better API coverage and test effects than predefined rule-based methods (*e.g.*, DocTer), which have been demonstrated by our experiment results in Fig. 9.

5.1.4 Metamorphic Testing on DL Framework. Metamorphic testing has been widely used to detect problems in DL models and systems [5, 151]. Despite the success of metamorphic testing in DL testing, researchers have paid limited attention to the metamorphic testing of DL frameworks. Ding *et al.* [22] constructed 11 MRS and validated AlexNet on the Caffe DL Framework. Although they were the first to try to design MRs to validate a DL framework, they focused on the DL model accuracy in tests and paid less attention to DL framework bugs. To address the limitation of localizing DL framework bugs at the parameter and tensor level, Chen *et al.* [9] proposed a testing framework with 18 MRs to test 10 common DL operators. However, their manually designed MRs only cover popular DL operators like Conv2d and lack support for the thousands of other DL framework APIs.

A significant challenge in metamorphic testing lies in the design of MRs. MRs typically require expert knowledge and manual design, making it difficult to cover diverse DL framework APIs. To improve the testing effect, researchers considered constructing MRs that apply to multiple APIs and combine them with other testing methods. Liu *et al.* [79] designed 15 MRs to construct equivalent inputs for different DL frameworks and reduce the false positive rate in differential testing (*i.e.*, 3.1%). Wei *et al.* [131] combined fuzzing with metamorphic testing to detect performance bugs in DL frameworks by comparing the execution time of test cases on diverse APIs under the two data types of `float16` and `float32`. The MR they designed is that programs carrying less precision information should execute faster. It is undeniable that they have taken an important step in detecting performance bugs. However, their MR can only provide a qualitative evaluation of API performance and cannot accurately and quantitatively detect and identify unexcepted runtime overhead.

Summary and Analysis. The major limitation of metamorphic testing methods is that the quality of MRs directly affects the results of metamorphic testing. Simple MRs in early research (*e.g.*, the impact of extending datasets [22]) can hardly uncover bugs in DL frameworks and have limited test effects, while complex MRs (*e.g.*, the impact of transposing the input and kernel of convolution operator [9]) can verify the functionality of operators in-depth, it is difficult to cover diverse APIs. To overcome these limitations, recent research has combined MRs with other testing techniques (*e.g.*, fuzz testing) to provide test oracles for more framework APIs and comprehensively verify DL framework behaviors [131].

5.1.5 Other Testing on DL Framework. In addition to the aforementioned testing techniques, researchers have used various software testing techniques such as smoke testing and just-in-time defect prediction to verify the functional correctness and efficiency of the DL frameworks [27, 38, 45, 132, 148]. Their methods have provided valuable insights and new perspectives for DL framework testing. However, these methods have rarely detected real-world bugs in DL frameworks, which limits their effectiveness. Jia *et al.* [45] conducted mutation testing to validate the quality of the unit test cases of three DL frameworks. Different from the aforementioned mutation-based fuzz testing, mutation testing observes whether unit test cases can identify the modified DL frameworks and kill them. They designed 13 categories of mutation operations to mutate the DL framework to generate mutants. Their experiment showed that more than 60% mutants were not detected by the unit test cases in the framework, which points out the ineffectiveness of these test cases and possible research opportunities. To promote the development of the testing methods for DL frameworks and systems, Kim *et al.* [57] proposed an open-source DL bug dataset, covering 4,577 bugs in 8 categories of DL

software, including DL frameworks, platforms, compilers, etc. Their dataset mainly included bug reports and statistical information like buggy entities. How to automatically reproduce these DL framework bugs on the corresponding environment to facilitate the evaluation of existing testing methods remains a challenge.

5.2 RQ3.1: Comparision and Analysis

Comparison Experiment and Results. To compare the effectiveness of testing methods based on different testing techniques and principles, we conducted experiment with five state-of-the-art and representative open-source testing methods, covering three popular testing techniques and both model-level and API-level methods, namely Muffin [32], FreeFuzz [131], DocTer with conforming (CI) and violating inputs (VI) [137], DeepREL [19], and TitanFuzz [20]. We have followed the instructions in their source code and repeatedly executed their source code for 48 hours in the same environment to separately detect bugs on TensorFlow and PyTorch frameworks. Note that the final outputs of existing testing methods were thousands of bug candidates, which requires dozens of man-months to validate these candidates, and how to validate and filter out invalid bug candidates automatically is out of our scope. Therefore, we directly recorded ❶ the API coverage and ❷ the number of bug candidates generated by each method in experiments. More details of the experiment results and settings are in our repository [2]. The experimental results are shown in Fig. 9, where the X-axis represents time and the Y-axis represents API coverage and the number of generated bug candidates in scientific notation. In experiments, Muffin covered the fewest APIs (52 TensorFlow APIs) and generated 233 bug candidates within 48 hours. TitanFuzz achieved the largest API coverage with a total of 3,413 APIs on two frameworks. Note that, FreeFuzz and DeepREL have generated a large number of bug candidates, with 737,070 and 379,478 for the two frameworks respectively, which is 49.67 times and 25.57 times the detected buggy candidates of TitanFuzz. However, we have not found an efficient way to automatically verify these bug candidates. So many bug candidates require a significant amount of manual verification and screening, which poses a challenge for comparing the testing effects of different testing methods in real-world scenarios.

Analysis. Based on the collected literature and experiment results in Fig. 9, we analyzed the advantages and drawbacks of the testing methods based on different testing techniques and principles as follows:

• *Differential Testing*. Existing differential testing can provide general test oracles for various DL framework APIs (*e.g.*, comparisons between different devices and different DL framework implementations), and effectively identify the status and numerical bugs. However, existing differential testing methods have two major limitations. ❶ Compared to metamorphic testing whose MRs are manually designed based on expected behaviors, the test oracles in differential testing are not accurate. Similar APIs or different framework implementations used in differential testing may have inconsistent output results, which leads to false positives in testing (*e.g.*, DeepREL's false positive rate exceeds 30%), requiring manual effort to validate the buggy cases. Such inconsistency also makes it difficult to detect performance bugs. ❷ The test oracle design could affect the API coverage in differential testing. Muffin, which utilized mutated models to conduct differential testing across multiple frameworks, can only cover APIs that are called by the models and have corresponding implementations in different frameworks, ultimately covering 52 APIs in our experiments. In contrast, DeepREL, leveraging the status and output results of equivalent APIs to construct test oracles, effectively explored 2,607 APIs within 48 hours. Such an observation of different is consistent with our analysis in §5.1.2.

• *Fuzz Testing*. Existing fuzz testing methods have generated thousands of valid test inputs by extracting constraints, mutating seed inputs, etc., covering a variety of DL framework APIs and comprehensively exploring framework behaviors. The development of ML techniques (*e.g.*, LLMs) has further improved the effect of fuzz testing in test case generation. LLM-guided fuzzing method TitanFuzz covered 3,413 APIs within 48 hours, the most among all methods

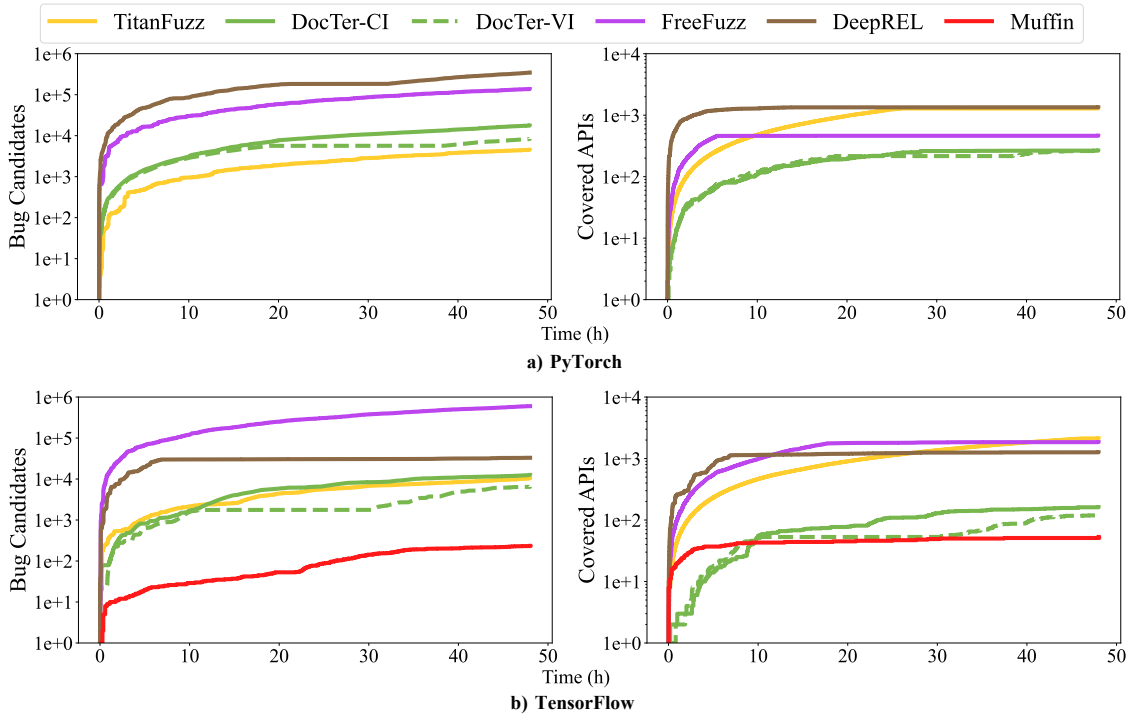

Fig. 9. Comparison of Testing Methods on the PyTorch and TensorFlow Frameworks

in our experiments. Existing fuzz testing methods have two major limitations. ❶ The effectiveness of fuzz testing is correlated with the way they generate test cases (*e.g.*, document-based, ML-based). DocTer extracted API constraints from documentation to generate test cases, therefore it can only test APIs with detailed documentation descriptions. The two kinds of inputs in DocTer finally covered 470 APIs in 48 hours of testing. In contrast, TitanFuzz leveraged LLM to generate and mutate test cases and covered 3,413 APIs in our experiments. ❷ Fuzzing cannot provide test oracles in tests. Merely using fuzz testing can only detect simple status bugs such as crashes [113]. Therefore, existing fuzzing methods typically utilized differential testing techniques to construct test oracles, effectively identifying the status and numerical bugs [21, 137]. Titanfuzz used differential testing to construct test oracles and effectively discovered buggy cases by identifying the difference between DL framework behaviors on GPU and CPU. FreeFuzz, further combined with metamorphic testing, can further detect performance bugs.

• *Metamorphic Testing.* Compared with other techniques, existing DL framework research using metamorphic testing is relatively limited. The MRs in metamorphic testing are abstracted from the expected functionalities of DL frameworks, providing accurate test oracles for identifying status, numerical, and performance bugs. It still has two limitations. ❶ The design of MRs requires the manual effort of experts with domain knowledge. Compared to test oracles of differential testing that are general for most APIs (*e.g.*, results on different devices), MRs are much more complex and typically designed for a specific operator (*e.g.*, the impact of transporting the input and kernel of the specific operator). ❷ Metamorphic testing methods can only verify whether the target framework is implemented as expected in aspects related to MRs, but cannot discover potential bugs in other aspects, making it difficult to explore various

Table 4. Representative DL Compiler Testing Methods

Category	Method Description	Test Object	#Bugs	Bug Type
Empirical Study	Studied and analyzed DL compiler bugs and summarized the bug symptoms and root causes [110]	TVM, Glow, and nGraph	/	/
Fuzz Testing	Generated valid test cases based on constraints and conducted differential testing on DL compilers [80]	TVM, ONNXRuntime, TensorRT, and PyTorch	72	status/numerical
	Mutated low-level IR of DL compilers and conducted coverage guided fuzz testing [82]	TVM	49	status/numerical/ performance
	Used domain-specific languages to describe constraints and generate valid test inputs [128]	TVM	16	status/numerical

buggy behaviors of DL frameworks. Therefore, current research (*e.g.*, FreeFuzz) mainly combines it with other testing techniques to provide test oracles with fewer false positives [131].

6 DL COMPILER TESTING

Compared with DL frameworks, DL compilers require higher professional knowledge requirements and are often used to solve task-specific problems of the underlying optimization of DL models, which makes it difficult to design testing methods for DL compilers. As a result, DL compiler testing has received less attention and are still in its infancy in recent years. Similar to §5, we introduce the existing testing methods for DL compilers according to the testing techniques used (RQ2.2), and then discuss the advantages and limitations of these methods (RQ3.2). Table 4 present representative DL compiler testing methods using different testing techniques. In addition, we conduct experiments on the TVM compiler, which is one of the most popular DL compilers, to compare and analyze the testing effects of three advanced open-source testing methods.

6.1 RQ2.2: DL Compiler Testing Methods

6.1.1 Study on DL Compiler Bugs. Researchers have conducted empirical studies on DL compilers bugs, analyzing types and root causes of bugs on popular DL compilers, including TVM, Glow, PlaidML, etc [25, 110]. They observed that crashes and wrong code (*i.e.*, optimization bugs) are the most common bugs in the DL compiler and the IR transformation stages were the most buggy stages, which provided insights for DL compiler testing methods. To fill the gap in prior studies on optimizing efficiency, Verma *et al.* [122, 123] focused on the optimization performance of DL compilers such as TensorFlow Lite and TensorRT (TF-TRT) in edge inference scenarios. Their large-scale evaluation revealed that integrated TF-TRT consistently performed better at high-precision floating-point operations but could not hold at low precision. Although the above work discussed the performance of DL compilers, they mainly focused on optimization performance, failing to delve into the potential bugs behind low performance or provide insights for detecting performance bugs.

Summary and Analysis. Existing empirical studies on DL compilers collect bug reports from the open-source community, analyze their symptoms and root causes, and further provide valuable suggestions for the development, application, and testing of DL compilers. However, they still lack investigation and research on some DL compilers (*e.g.*, XLA) and fix patterns of DL compiler bugs, which can promote the research on testing and repair of DL compiler bugs. Furthermore, researchers mainly focus on compiler bugs related to functional correctness, such as crashes and optimization bugs, only Shen *et al.* [110] have studied the bad performance and root causes of DL compilers.

6.1.2 Fuzz Testing on DL Compiler. As we have discussed in §4.3, DL compiler testing faces the challenge of generating valid test cases and exploring the potential behaviors of DL compilers. To overcome the challenge, researchers have leveraged the fuzz testing techniques to generate test cases based on generation and mutation.

Generation-based Fuzz Testing. Researchers have proposed a series of generation methods that follow semantical constraints [80, 106]. Liu *et al.* [80] proposed NNSmith that leveraged the operator constraints provided by users to generate valid graphs for DL compiler testing and performed cross-validation on multiple DL compilers. Recently, researchers have observed that previous methods, while generating test cases that conform to constraints, have difficulty carrying enough effective information (*e.g.*, operators and attributes) to explore DL compiler behavior. Moreover, they often require heavy manual effort to write constraints for each operator in tests. To overcome these limitations, researchers proposed GenCoG which used domain-specific languages to describe the constraints of operators and automatically generate valid graphs based on feasible combinations of input tensor types and attributes to carry useful information [128]. It finally covered 62 operators and detected 14 bugs on TVM. Unfortunately, their implementation only supported the TVM compiler. Generalizing to various DL compilers can increase practical value.

Mutation-based Fuzz Testing. In addition to generation-based fuzzing methods, researchers also pay attention to mutation-based fuzz testing [76, 88]. Based on the findings from their empirical study, Shen *et al.* [110] designed the testing tool TVMfuzz for the TVM compiler. It conducted fuzz testing based on the directed graph built by TVM APIs and mutated the shape and type of tensors to construct new unit tests. It finally detected 8 crash bugs by performing differential testing on the two versions of TVM. However, TVMFuzz can only detect obvious bugs such as crashes, but cannot catch numerical bugs. To break through the limitation, DeepDiff [76] implemented priority-guided fuzz testing that efficiently identifies logic errors by mutating IR seeds to maximize the difference between results on different TVM versions and detect 9 new bugs. Liu *et al.* [82] further proposed the first coverage-guided fuzz testing tool for testing the tensor compiler (*i.e.*, TVM), Tzer. They designed six kinds of mutation operators (*e.g.*, inserting loops, replacing operations) for the low-level IR of DL compilers to trigger more potential behaviors, and added the mutated IR to the seed pool when it covered more code. It finally detected 49 new bugs on the TVM compiler, surpassing prior tools (*e.g.*, TVMfuzz) in terms of coverage and bug detection. However, the above fuzzing methods mainly focus on the IR transformation stages which are related to optimization, and ignore the model loading stage. To fill the research gap and cover more DL compiler operators, Shen *et al.* [111] transferred the knowledge of DL framework fuzzers (*e.g.*, DocTer and DeepRel) to generate effective test cases for diverse DL compiler operators to detect crashes and inconsistencies.

Summary and Analysis. Fuzz testing can effectively generate test inputs for DL compilers and explore potential behaviors and has already helped developers identify hundreds of bugs on these compilers. In the early stages, DL compiler fuzzing methods utilized simple random generation or mutation on tensor shapes to generate test cases [106, 110]. With the advancement of technology, researchers have proposed a series of methods such as coverage-guided fuzzing and priority-guided fuzzing to efficiently generate valid test cases and deeply explore DL compiler behaviors. These testing methods can cover more DL compiler operators and be combined with differential testing to detect more types of bugs. Despite this, it can be observed that existing DL compiler fuzzing methods still rely on predefined rules and constraints to construct and mutate test cases. Compared to state-of-the-art DL framework fuzzing methods using ML techniques [20, 71], these DL compiler fuzzing methods still have a lot of room for improvement.

6.1.3 Other Testing on DL Compiler. The DL compiler fuzzers mainly construct test oracles by leveraging the concept of differential testing (*e.g.*, results between different devices), enabling the detection of state and numerical bugs, but they cannot effectively identify complex optimizations or performance bugs. To construct test oracles to discover more

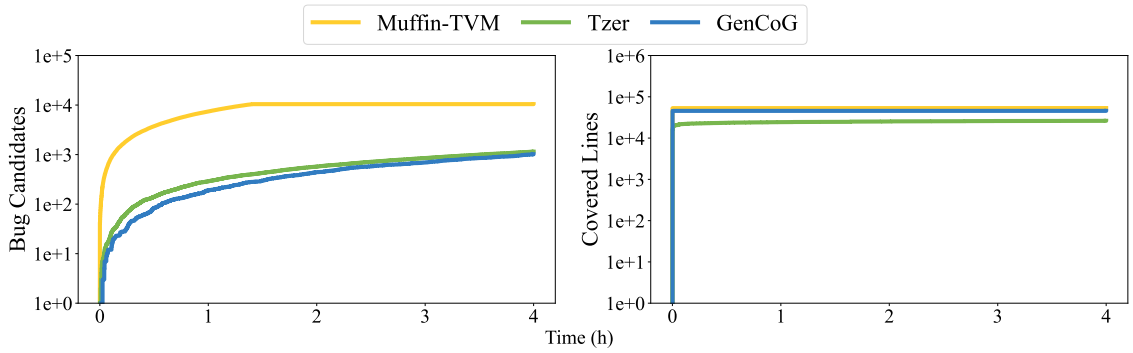

Fig. 10. Comparison of Testing Methods on the TVM Compiler

complex optimization bugs, researchers have delved into metamorphic testing. Xiao *et al.* [136] designed elaborated metamorphic relations to test DL compilers. For example, they introduced a series of operators whose final result is 0 in the input, and then in tests and then observed the change of the DL compiler result to detect potential bugs. They have detected 4 DL compiler bugs on three compilers (*i.e.*, TVM, Glow, and XLA), and Fig. 6 provides one example bug. This bug causes the DCE optimization in Glow to mistakenly delete active and valid nodes in the computation graph, resulting in the generated optimized operators outputting unexpected results. In addition, to alleviate the false positives introduced by differential testing on different compilers, Zhou *et al.* [153] proposed a method that builds tensor graphs with the same semantics and different syntaxes as test oracles to identify mis-compilation bugs in DL compilers. The concept of constructing equivalent graphs/programs has shown potential in both DL framework testing and compiler testing [125, 153].

6.2 RQ3.2: Comparision and Analysis

Comparison Experiment and Results: We have compared three advanced open-sourced DL compiler testing methods to study their test effects, namely Tzer [82], Muffin-TVM and GenCog [80], where Muffin-TVM is a reimplementation of Muffin [32] by researchers, aiming to detect status bugs by converting DL models to relay IR with TVM's Keras frontend. We have followed their recommendation to repeatedly execute their source code for 4 hours in the same environment to separately detect bugs on TVM [82]. Similar to the setting in §5.2, in the experiments, we collected ❶ the code line coverage and ❷ the number of bug candidates generated by each method. More details are in our repository [2] The experimental results are shown in Fig. 10, where the X-axis represents time and the Y-axis represents line coverage and the number of bug candidates in scientific notation. We can observe that Tzer has the lowest line coverage, covering only 26.173 lines after four hours of execution, while the other two methods separately covered 45,610 lines and 53,309 lines within the first hour. Particularly, Muffin-TVM covered more than 50,000 lines of TVM code and generated 10436 bug candidates in the experiment, significantly higher than the results of the other two methods. Our analysis showed that such results could be related to its test case generation methods. Muffin-TVM built diverse DL models using advanced framework testing methods and converted them to TVM for testing. The model graphs carried rich information (*e.g.*, operators) and conformed to the compiler's semantical constraints, thus effectively exploring DL compiler code lines and discovering bug candidates.

Analysis. Existing DL compiler testing methods mainly employ fuzzing techniques and generate test cases based on the compiler's semantical constraints. These methods effectively explore various DL compiler behaviors while

Table 5. Representative DL Hardware Library Testing Methods

Category	Method Description	Test Libraries
Testing on Functionality	Generated test patterns for DL accelerator to ensure and reliability of its functional implementation [37]	NVDLA
	Designed several MRs and conducted metamorphic testing for the operators of DL accelerators [124]	HiAI/SNPE
Testing on Performance	Conducted large-scale experiments to evaluate the performance of the convolution operators in cuDNN [52]	cuDNN
	Compared and evaluated the performance of the fixed CNN architectures on three DL hardware libraries [95]	cuBLAS, cuDNN, and TensorRT

carrying useful information (*e.g.*, operators and tensors) without violating the syntax of DL compilers, covering dozens of thousands of code lines in tests, as shown in Fig. 10. In addition, both existing research [111] and our experiment results in Fig. 10 demonstrate the potential of leveraging DL framework testing methods to cover a large amount of compiler code and trigger buggy cases in compiler testing. However, existing methods have limitations in constructing test oracles, and can only detect limited bug types. They mainly rely on the state of program execution and different results between multiple DL compilers to identify crashes and inconsistency bugs, but can hardly identify optimization bugs and performance bugs in compilers. For example, the Muffin-TVM can only detect crashes during TVM execution. Although some researchers have proposed using metamorphic rules and equivalent graphs to construct test oracles, their methods are usually specific to limited operators and bug types and cannot cover diverse DL compiler operators and bugs, which limits the practical value of existing testing methods.

7 DL HARDWARE LIBRARY TESTING

In this section, we mainly focus on how existing work design testing methods for the DL hardware-related libraries (RQ2.3) and analyze their strengths and weaknesses (RQ3.3). We have observed that existing research on DL hardware library testing mainly focuses on verifying and evaluating the correctness and efficiency, and rarely delves into bug detection with different testing techniques. Therefore, we do not introduce existing work according to their testing techniques, but introduce the research on functional correctness and efficiency of DL hardware libraries respectively, and comparatively analyze the advantages and disadvantages of these studies in §7.2. We introduce representative testing methods in Table 5 and explain how they overcome the above challenges in the following section. Since we cannot find a specific description of the number and type of bugs, we do not show the relevant information in the table.

7.1 RQ2.3: DL Hardware Library Testing Methods

7.1.1 Testing on DL Hardware Library Functional Correctness. In terms of empirical research, Huang *et al.* [42] investigated and understood the dependency bug in the DL stack. They investigated the symptoms and root causes of a total of 326 bugs in DL libraries which include DL applications, DL frameworks, DL accelerators, and DL hardware. They found that violating the constraints among dependencies is the main root cause of bugs and then suggested that developers should receive systematic training to fully understand the DL stack and life cycle to reduce the occurrence of bugs. Wang *et al.* [126] systematically studied the compatibility issues in the underlying DL libraries of DL systems that can affect the deployment of DL systems and lead to degradation of execution performance, including their symptoms and fixing patterns, providing insights into the testing and repairing of DL hardware libraries. Their work promoted the following research on DL hardware-related libraries research and DL stack.

Researchers have also developed a series of methods to verify the correctness of the implementation of DL hardware libraries [37, 120]. He *et al.* [37] designed a set of functional testing methods for the compute units and control units of NVIDIA Deep Learning Accelerator (NVDLA) and achieved high test coverage and low test time overhead in tests. However, their work relies on test coverage to verify the reliability of the library and did not construct test oracle to identify real-world bugs. To construct effective test oracles in testing, Wang *et al.* [124] conducted metamorphic testing and designed a series of MRs on the convolution and softmax operators of DL accelerators HiAI and Snapdragon Neural Processing Engine (SNPE) to verify the accuracy of accelerators and explore potential accuracy defects. Their results showed that HiAI has better accuracy performance than SNPE on the float16 data type. To explore the abnormal behavior of hardware libraries, researchers have leveraged fuzz testing to generate a variety of test cases. CUDAsmith [49] has been designed as a test case generation tool for the underlying NVCC and Clang library of the DL computing platform CUDA. It implemented a generation-based kernel function generator to create valid test inputs that were adapted for the CUDA context and constructed test oracles based on random differential testing to identify bugs. In their experiments, they mainly detected the build failures and timeout failures caused by wrong code. All aforementioned test methods did not report identified real-world bugs in experiments.

7.1.2 Testing on DL Hardware Library Efficiency. Existing research mainly evaluates the efficiency of DL hardware libraries on specific operators or model architectures [52, 95]. Dongarra *et al.* [23] conducted a large-scale evaluation on batched GEMM operation implemented in four DL hardware libraries, namely, MKL, OpenMP, CuBLAS, and MAGMA. They found that the operation implemented in these libraries used simple memory layouts, leading to suboptimal performance. Based on their findings, they proposed optimization that improves the performance of MKL and CuBLAS operations by up to 6 times. Sun *et al.* [116] observed that existing work lacks attention to DL libraries on the ROC platform described by AMD. To fill this gap and systematically evaluate the performance of DL libraries rocBLAS and MIOpen, Sun *et al.*constructed four benchmarks, including K-Nearest Neighbors (KNN). Their experiment results showed that MIOpen can perform well in both model training and inference processes. Recently, Öz *et al.* [100] conducted a detailed experimental analysis and evaluation of two BLAS libraries, cuBLAS and MAGMA on the basis of prior work. The experiment results showed that CuBLAS and MAGMA did not exhibit significant differences in operations with large computational loads. However, cuBLAS offered higher performance in terms of GFlops and achieved higher streaming multiprocessor utilization in most executions. Beyond these DL libraries designed for computation and acceleration on hardware, Li *et al.* [64, 65] conducted a systematic evaluation on four DL hardware libraries, PCIe, NVLink, NV-SLI, NVSwitch, and GPUDirect, which provide optimizations for GPUs interconnection. They constructed empirical evaluations and observed that different GPU combinations have a significant impact on GPU communication performance. For example, GPUDirect can exhibit the best performance on the supercomputer 'Summit'. Although the evaluation results of the aforementioned methods have provided valuable observations and insights on the efficiency of various DL hardware libraries, they still lacked testing case generation and further real-world bug detection.

7.2 RQ3.3: Comparision and Analysis

Compared with the DL framework and compiler, existing testing methods on DL hardware libraries are relatively limited. Researchers mainly built large-scale experiments to evaluate and verify the correctness and efficiency of specific DL hardware library functions. In terms of efficiency, researchers have established large-scale experiments to evaluate the performance of DL hardware libraries in scenarios such as executing GEMM operators and inter-GPU communication. However, existing performance evaluation lacks in-depth analysis and understanding of the root causes

for poor performance. Furthermore, there is no systematic study on the performance bug symptoms, root causes, and fix patterns of various DL hardware libraries. In terms of correctness, although researchers have leveraged fuzzing and metamorphic testing to construct test cases and test oracles to verify the correctness of their functions, they can only evaluate and assess specific DL hardware libraries (*e.g.*, CUDA accelerators), and have limited generalization ability, and also lack the ability to detect and identify real-world bugs.

8 FUTURE GAZING

Although the testing research on the DL library has made considerable progress, this field is still in its nascent stage. There are still many challenges to be addressed. In this section, we first conclude the findings from the prior literature study, aiming to explore practical implications and provide meaningful conclusions. Subsequently, based on these findings, we list the challenges faced by existing methods and potential research opportunities, aiming to provide insights and a clear set of guidelines for future research in DL library testing.

8.1 RQ4.1: Findings

• **Finding 1:** *Conducting empirical studies can promote the design of effective DL library testing methods.* Empirical studies can provide valuable findings of DL library bugs, root causes, and fix patterns and help the following research construct effective testing methods, which have been shown in related literature [10, 110]. In addition, with the continuous evolution of DL libraries, it is meaningful to study and summarize the distribution and fix patterns of bugs in their latest releases. For example, early studies on DL frameworks focused on bugs in programs using a specific framework and lacked analysis of the fix patterns [149]. Recent researchers have analyzed thousands of bugs in four frameworks, as well as their root causes and fix patterns, providing more comprehensive and detailed findings on DL framework testing than earlier studies [10]. However, existing research lacks of study on bugs of various DL hardware libraries (*e.g.*, MKL). Systematically studying bugs in various DL hardware libraries and pointing out their root causes is of practical implication for developing new DL library testing methods.

• **Finding 2:** *DL library testing research mainly focuses on DL frameworks at the high abstraction level and pays less attention to the underlying DL compiler and hardware libraries.* DL compilers and hardware libraries optimize the abstract DL models and execute corresponding operators on specific hardware. The bugs in these libraries can directly affect the calculation results of DL models. Even worse, they can hardly be detected in higher-level tests (*e.g.*, tests on DL systems and DL frameworks). However, researchers have paid insufficient attention to testing existing DL compilers and hardware libraries. Among the papers we collected, only 12.90% and 13.98% are related to DL compilers and DL hardware library testing, respectively. Especially for DL hardware libraries, the related literature lacked systematic and comprehensive testing methods to identify real-world bugs [95, 100], which limits the practical value of these methods.

• **Finding 3:** *Although researchers have proposed various DL framework testing methods, it is difficult to compare their test effects in real-world scenarios.* Based on the collected literature, we have observed that, in the recent five years, dozens of DL framework testing methods and tools have been proposed, which conducted testing for different bugs such as crashes, inconsistent output, NaN, performance problems, and documentation bugs [34, 131, 137]. However, it is difficult to compare the effectiveness of these methods in real-world scenarios. On the one hand, different testing methods discover different types of bugs (*e.g.*, performance bugs and crashes) on different library releases, making it difficult to quantitatively compare these testing methods. On the other hand, existing methods often output thousands of bug candidates (Fig. 9). Verifying and determining real bugs requires a significant amount of manual effort. Although researchers have compared several test tools based on metrics like line coverage [10], and we have also conducted

comparative experiments for five state-of-the-art methods in §5.2, there is still a lack of comprehensive benchmarks to compare and evaluate the effectiveness of existing DL framework testing methods. Such benchmarks are essential not only to create standardized baselines for future research but also to encourage broader deployment and practical application of various testing methods in real-world scenarios.

• **Finding 4:** *Existing testing and evaluation on DL libraries mainly focuses on functionality correctness (e.g., crashes) and pays less attention to performance bugs.* Performance bugs can introduce significant runtime and resource overhead to DL libraries and systems built upon them, resulting in energy waste and environmental concerns in real-world deployment [61, 98]. However, in our literature study, only 38.71% of collected papers paid attention to performance bugs, and most of them only conducted empirical studies on performance bugs. Even though several methods claimed to be able to detect performance bugs, they usually involved identifying one or two cases of memory leaks or performance degradation, lacking systematic testing methods [131, 148]. There is an urgent need to discover performance bugs and ensure the efficiency of DL libraries and even DL systems in real-world scenarios.

• **Finding 5:** *Combining and leveraging the advantages of multiple testing techniques has the potential to bring better testing results to DL library testing.* Using a single testing technique can only cover specific bug types (*e.g.*, crashes [113]), while combining multiple techniques has the potential to detect diverse bug types and cover a variety of DL library APIs, which has been demonstrated in related literature [20, 131]. In contrast, DL compiler and hardware library research typically relied on simple testing and evaluation methods, focusing on specific bugs like mis-compilation [136]. We suggest that future developers can combine various techniques (*e.g.*, metamorphic testing and ML techniques) to generate valid test cases and construct accurate test oracles, thereby effectively identifying various bugs in real-world scenarios.

• **Finding 6:** *The DL library testing method mainly detects bugs and reports them to developers, rarely discovering security vulnerabilities.* In the literature collection, we have observed that most existing works focus on DL library bugs that degrade the functionality and performance of DL libraries. Only a small number of research tests and discovers security vulnerabilities and weaknesses in DL libraries [17, 62, 154]. The vulnerabilities in DL libraries can be exploited by attackers, leading to memory exhaustion, kernel crashes, etc. that compromise the security of DL-based software and systems. With the widespread application of DL technology, we suggest that industry and academia pay more attention to the security vulnerabilities and weaknesses in DL libraries.

• **Finding 7:** *The continuous evolution of DL techniques poses challenges for the maintenance and testing of DL libraries.* Existing work on DL library testing often focuses on identifying bugs in specific versions of the library. However, libraries continue to have new versions released, which may fix some inherent bugs while potentially introducing new ones. Regression bugs are also huge pitfalls in DL libraries[2], where previously functional functions in older versions become problematic in the new version. In addition, DL libraries that emerged in the era of LLMs (*e.g.*, HuggingFace Transformers [133]) may also introduce risks to DL workflow and systems. However, existing research [45] revealed that the unit test cases in the DL libraries are not sufficient to discover potential bugs. To effectively detect bugs in the ever-changing DL library, existing work [138] has proposed continuous testing methods to test different releases of libraries, but they focus on testing different releases directly with existing techniques and lack attention to the characterization of the evolution of the frameworks. Building a lightweight bug benchmark for various DL libraries is of practical significance in effectively identifying bugs during software evolution, which can reduce the cost of maintaining and developing DL libraries.

[2]https://github.com/pytorch/pytorch/issues/95432

8.2 RQ4.2: Challenges and Opportunities

Based on the above findings, we summarize the main challenges and the research opportunities as follows.

• **Challenge 1: Collecting bug reports to study various DL Hardware libraries.** The major challenge in conducting empirical research for DL hardware library bugs is collecting various bug cases. Different from open-sourced DL frameworks and compilers, DL hardware libraries are usually designed for specific hardware and do not provide available code or bug (issue) lists. In addition, there are various DL hardware libraries covering different scenarios such as computing and hardware communication. Researchers could first collect buggy cases or projects of a specific hardware library from open-source communities such as Stack Overflow, proposing a taxonomy for these bugs. Then they could extend the research to different libraries, providing valuable findings for DL hardware library testing.

• **Challenge 2: Constructing test cases and oracles for DL compiler and hardware library.** There are two major challenges in testing and identifying bugs in DL compilers and hardware libraries The first challenge lies in building valid test cases. The inputs of DL compilers and hardware libraries often consist of specific programming languages and operators, which makes it difficult to find enough usable reference code and test cases in the open-source community and extract input constraints. Building test oracles for these libraries is another challenge. Especially for DL hardware libraries designed for specific hardware, it is difficult to build test oracles for them using different devices or implementations. Constructing equivalent graphs or generating test cases based on LLM techniques which have shown strong capabilities in traditional SE testing tasks (*e.g.*, guiding differential and fuzz testing [20, 26, 40]), could be a potential solution to overcoming these challenges. In addition, our experiment results in Fig. 10 and existing research [111] show that using advanced DL framework testing methods to effectively generate diverse test cases and convert them into underlying libraries is also a promising research direction.

• **Challenge 3: Conducting a comprehensive evaluation of DL framework testing methods.** Firstly, a major challenge in constructing benchmarks for existing testing methods is to collect a comprehensive dataset including various bugs on different DL frameworks. DL frameworks evolve rapidly, and various bugs typically exist in the different releases and environments. Researchers could refer to existing software testing datasets and benchmarks [6, 53, 66, 74] to build a dataset that covers a large number of APIs and various types of bugs and use virtual environment management tools like Docker, thus supporting bug datasets for multiple DL frameworks and releases. Another challenge lies in automated processing output results of existing testing methods (*i.e.*, bug candidates) to obtain the real bugs they detected. Existing research often reports thousands of bug candidates (as discussed in §5.2), and evaluating and verifying the detection results of detection methods requires a significant amount of manual effort. Automatically merging bug candidates based on ML techniques is a potential research direction.

• **Challenge 4: Identifying performance bugs on DL libraries.** The major challenge in identifying performance bugs is designing effective test oracles to estimate the expected performance of a given DL program on the hardware. Existing research primarily leverages metamorphic relations or prior bug reports to construct test oracles and detect performance bugs. For example, researchers compare the executions of the same function with different data types and observe whether the memory or time overhead has increased or decreased as expected, which could provide a qualitative test oracle [131]. However, such a method can not precisely predict the performance of the given API or operator quantitatively, and can only identify the performance bugs related to data types.

• **Challenge 5: Designing effective DL library testing methods with multiple testing techniques.** How to combine the advantages of different testing techniques to design new testing methods is a challenge. As we analyzed in §5.2, different testing techniques have their strengths and limitations. For example, differential testing can provide

general test oracles for diverse APIs, but it may lead to false positives in tests. Fuzzy testing can efficiently generate test samples. Researchers can refer to previous methods [20, 131], using fuzzing and ML techniques to efficiently generate test cases to cover behaviors of different DL libraries and designing some general MRs and equivalent relationships to construct accurate test oracles for various bugs.

• **Challenge 6: Discovering security vulnerabilities of DL libraries.** Discovering vulnerabilities in DL workflow and building reasonable threat models is the major challenge. On the one hand, researchers can generate test cases based on the security properties summarized in existing machine learning security research [142, 145] to evaluate and identify problems in the DL library. On the other hand, researchers can also explore potential weaknesses and vulnerabilities based on common attack scenarios in AI security and software security, combined with the characteristics of different DL library components, such as using DL library APIs to perform unauthorized operations, illegally modifying system files of host, and implanting backdoors into DL software and systems.

• **Challenge 7: Designing benchmarks to test DL libraries during evolution.** The major challenge in designing bug benchmarks for DL libraries is how to efficiently design test cases for evolving libraries, thereby covering the diverse functionalities of the libraries. As new optimization algorithms and model architectures are continuously proposed and implemented [85, 150], updating and modifying test cases in the benchmark requires a lot of manual effort. A promising approach is to leverage code similarity metrics and Software Component Analysis (SCA) to identify existing components in the DL library that are functionally similar to newly released components and migrate the test cases of the former to the latter, thereby reducing the effort of maintaining the benchmark.

9 CONCLUSION

The rapid development and widespread deployment of DL-driven systems have attracted researchers in academia and industry to investigate and study the DL underlying library that supports DL systems. Existing research has achieved fruitful results in testing and validating DL library bugs. However, with the development and iteration of DL techniques and software, there is still room for improvement in these testing methods. To comprehensively summarize the testing research of DL underlying libraries, understand their effectiveness and limitations, and discuss challenges and directions for future research, this paper first describes the definitions of DL library bugs and testing. Then, based on four proposed review questions, it reviews the existing testing research on three components of DL libraries (*i.e.*, DL frameworks, DL compilers, and DL hardware libraries). Finally, this paper summarizes the findings of the literature survey and discusses the challenges of DL library testing, aiming to promote further development and real-world application of DL library testing research.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions. This work is supported partially by the National Key Research and Development Program of China (2023YFB3107400), the National Natural Science Foundation of China (62006181, 62132011, 62161160337, 62206217, U20A20177, U21B2018), and the Shaanxi Province Key Industry Innovation Program (2021ZDLGY01-02 and 2023-ZDLGY-38). Thanks to the New Cornerstone Science Foundation and the Xplorer Prize.

REFERENCES

[1] 2010. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061

[2] 2024. Our repository with more experiment details. https://github.com/shiningrain/CSUR_DL_library_survey.

[3] Marcel Aach, Eray Inanc, Rakesh Sarma, Morris Riedel, and Andreas Lintermann. 2023. Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks. *Journal of Big Data* 10, 1 (2023), 1–23.

[4] Martín Abadi. 2016. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN international conference on functional programming*. 1–1.

[5] Aitor Arrieta. 2022. Multi-objective metamorphic follow-up test case selection for deep learning systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1327–1335.

[6] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 66–77.

[7] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542.

[8] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 357–369.

[9] Jinyin Chen, Chengyu Jia, Yunjie Yan, Jie Ge, Haibin Zheng, and Yao Cheng. 2024. A Miss Is as Good as A Mile: Metamorphic Testing for Deep Learning Operators. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2005–2027.

[10] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–31.

[11] Junjie Chen and Chenyao Suo. 2022. Boosting compiler testing via compiler optimization exploration. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–33.

[12] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler test-program generation via memoized configuration search. In *Proc. 45th International Conference on Software Engineering*.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[14] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. *technical report hkust-cs98-01* (1998).

[15] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 750–762.

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[17] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. 2023. IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2383–2400.

[18] Di Cui, Xingyu Li, Feiyang Liu, Siqi Wang, Jie Dai, Lu Wang, and Qingshan Li. 2022. Towards Demystifying the Impact of Dependency Structures on Bug Locations in Deep Learning Libraries. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 249–260.

[19] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[20] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 70, 13 pages. https://doi.org/10.1145/3597503.3623343

[21] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56.

[22] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a deep learning framework by metamorphic testing. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 28–34.

[23] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science* 108 (2017), 495–504.

[24] Xiaoting Du, Yulei Sui, Zhihao Liu, and Jun Ai. 2022. An empirical study of fault triggers in deep learning frameworks. *IEEE Transactions on Dependable and Secure Computing* (2022).

[25] Xiaoting Du, Zheng Zheng, Lei Ma, and Jianjun Zhao. 2021. An Empirical Study on Common Bugs in Deep Learning Compilers. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 184–195.

[26] Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. 2024. Mokav: Execution-driven Differential Testing with LLMs. *arXiv preprint arXiv:2406.10375* (2024).

[27] Jian Ge, Huiqun Yu, Guisheng Fan, Jianhao Tang, and Zijie Huang. 2023. Just-In-Time Defect Prediction for Intellignet Computing Frameworks (in Chinese). *Journal of Software* 34, 9 (2023), 0–0.

[28] Bahar Gezici and Ayça Kolukısa Tarhan. 2022. Systematic literature review on software quality for AI-based software. *Empirical Software Engineering* 27, 3 (2022), 66.

[29] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

[30] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 621–631.

[31] Diandian Gu, Yining Shi, Haozhe Liu, Ge Wu, Haiou Jiang, Yaoshuai Zhao, and Yun Ma. 2022. Defect Detection for Deep Learning Frameworks Based on Meta Operators (in Chinese). *Chinese Journal Of Computers* 45, 2 (2022), 240–255.

[32] 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.

[33] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 810–822.

[34] 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.

[35] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. 2023. Characterizing and understanding software security vulnerabilities in machine learning libraries. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 27–38.

[36] 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.

[37] Yi He, Takumi Uezono, and Yanjing Li. 2021. Efficient functional in-field self-test for deep learning accelerators. In *2021 IEEE International Test Conference (ITC)*. IEEE, 93–102.

[38] Steffen Herbold and Tobias Haar. 2022. Smoke testing for machine learning: simple tests to discover severe bugs. *Empirical Software Engineering* 27, 2 (2022), 45.

[39] Shuo Hong, Hailong Sun, Xiang Gao, and Shin Hwei Tan. 2024. Investigating and Detecting Silent Bugs in PyTorch Programs. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 272–283.

[40] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and Strong Test Oracle Generation with LLMs. *arXiv preprint arXiv:2405.03786* (2024).

[41] Qianchao Hu, Feng Wang, Binglin Liu, and Haitian Liu. 2023. Research on Deep Neural Network Testing Techniques. In *Proceedings of the 2023 4th International Conference on Machine Learning and Computer Application*. 113–119.

[42] Kaifeng Huang, Bihuan Chen, Susheng Wu, Junming Cao, Lei Ma, and Xin Peng. 2023. Demystifying dependency bugs in deep learning stack. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 450–462.

[43] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[44] Jiahe Ji, Wei Kong, Jianwen Tian, Taotao Gu, Yuanping Nie, and Xiaohui Kuang. 2023. Survey on Fuzzing Techniques in Deep Learning Libraries. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*. IEEE, 461–467.

[45] Li Jia, Hao Zhong, and Linpeng Huang. 2021. The unit test quality of deep learning libraries: A mutation analysis. In *2021 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 47–57.

[46] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Zexuan Li. 2022. How Do Injected Bugs Affect Deep Learning?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 793–804.

[47] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 604–620.

[48] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.

[49] Bo Jiang, Xiaoyan Wang, Wing Kwong Chan, TH Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. 2020. Cudasmith: A fuzzer for CUDA compilers. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 861–871.

[50] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.

[51] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1946–1956.

[52] Marc Jorda, Pedro Valero-Lara, and Antonio J Pena. 2019. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access* 7 (2019), 70461–70473.

[53] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[54] Hong Jin Kang, Pattarakrit Rattanukul, Stefanus Agus Haryono, Truong Giang Nguyen, Chaiyong Ragkhitwetsagul, Corina Pasareanu, and David Lo. 2022. SkipFuzz: Active Learning-based Input Selection for Fuzzing Deep Learning Libraries. *arXiv preprint arXiv:2212.04038* (2022).

[55] K Ken. [n. d.]. *Exclusive: surveillance footage of tesla crash on sf's bay bridge hours after elon musk announces "self-driving" feature.* https://theintercept.com/2023/01/10/tesla-crash-footage-autopilot/

[56] Nikhil Ketkar and Nikhil Ketkar. 2017. Introduction to keras. *Deep learning with python: a hands-on introduction* (2017), 97–111.

[57] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. 2021. Denchmark: A bug benchmark of deep learning-related software. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 540–544.

[58] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.

[59] Barbara Kitchenham, Lech Madeyski, and David Budgen. 2022. SEGRESS: Software engineering guidelines for reporting secondary studies. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1273–1298.

[60] Eliska Kloberdanz, Kyle G Kloberdanz, and Wei Le. 2022. DeepStability: A study of unstable numerical methods and their solutions in deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 586–597.

[61] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the Carbon Emissions of Machine Learning. *arXiv preprint arXiv:1910.09700* (2019).

[62] Zhongzheng Lai, Huaming Chen, Ruoxi Sun, Yu Zhang, Minhui Xue, and Dong Yuan. 2024. On Security Weaknesses and Vulnerabilities in Deep Learning Systems. *arXiv preprint arXiv:2406.08688* (2024).

[63] Maksim Levental and Elena Orlova. 2020. Comparing the costs of abstraction for dl frameworks. *arXiv preprint arXiv:2012.07163* (2020).

[64] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.

[65] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: evaluating modern GPU interconnect via a multi-GPU benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 191–202.

[66] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. 2020. Benanza: Automatic μBenchmark Generation to Compute" Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 440–450.

[67] Hang Li. 2018. Deep learning for natural language processing: advantages and challenges. *National Science Review* 5, 1 (2018), 24–26.

[68] Junqiang Li, Senyi Li, Jiawei Wu, Long Luo, Yang Bai, and Hongfang Yu. 2022. MMOS: Multi-Staged Mutation Operator Scheduling for Deep Learning Library Testing. In *GLOBECOM 2022-2022 IEEE Global Communications Conference*. IEEE, 6103–6108.

[69] 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* 32, 5 (2023), 1–34.

[70] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[71] Xiaoting Li, Xiao Liu, Lingwei Chen, Rupesh Prajapati, and Dinghao Wu. 2022. ALPHAPROG: reinforcement generation of valid programs for compiler fuzzing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 12559–12565.

[72] Zengyang Li, Sicheng Wang, Wenshuo Wang, Peng Liang, Ran Mo, and Bing Li. 2023. Understanding bugs in multi-language deep learning frameworks. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 328–338.

[73] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 562–566.

[74] Yunkai Liang, Yun Lin, Xuezhi Song, Jun Sun, Zhiyong Feng, and Jin Song Dong. 2022. gDefects4DL: a dataset of general real-world deep learning program defects. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 90–94.

[75] Shuyan Liao and Chun Shan. 2024. A PSO-based Method to Test Deep Learning Library at API Level. In *Proceedings of the 3rd International Conference on Computer, Artificial Intelligence and Control Engineering*. 117–130.

[76] Kuiliang Lin, Xiangpu Song, Yingpei Zeng, and Shanqing Guo. 2023. DeepDiffer: Find Deep Learning Compiler Bugs via Priority-guided Differential Fuzzing. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 616–627.

[77] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*. IEEE, 152–156.

[78] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*. 1–10.

[79] 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.

[80] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[81] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Neuri: Diversifying dnn generation via inductive rule inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 657–669.

[82] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26.

[83] Ling Liu, Yanzhao Wu, Wenqi Wei, Wenqi Cao, Semih Sahin, and Qi Zhang. 2018. Benchmarking deep learning frameworks: Design considerations, metrics and beyond. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1258–1269.

[84] Xuanzhe Liu, Diandian Gu, Zhenpeng Chen, Jinfeng Wen, Zili Zhang, Yun Ma, Haoyu Wang, and Xin Jin. 2023. Rise of Distributed Deep Learning Training in the Big Model Era: From a Software Engineering Perspective. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–26.

[85] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. [n. d.]. MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases. In *Forty-first International Conference on Machine Learning*.

[86] Zhihao Liu, Yang Zheng, Xiaoting Du, Zheng Hu, Wenjie Ding, Yanming Miao, and Zheng Zheng. 2022. Taxonomy of Aging-related Bugs in Deep Learning Libraries. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 423–434.

[87] Nikolaos Louloudakis, Perry Gibson, José Cano, and Ajitha Rajan. 2023. DeltaNN: Assessing the impact of computational environment parameters on the performance of image recognition models. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 414–424.

[88] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 248–260.

[89] Xiangyue Ma, Xiaoting Du, Qing Cai, Yang Zheng, Jing Hu, and Zheng Zheng. 2023. A Survey on Testing of Deep Learning Frameworks (in Chinese). *Journal of Software* (2023).

[90] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[91] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software engineering for AI-based systems: a survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–59.

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

[93] Microsoft. 2023. *ONNX Github repository*. https://github.com/onnx/onnx

[94] Yanzhou Mu, Juan Zhai, Chunrong Fang, Xiang Chen, Zhixiang Cao, Peiran Yang, Yinglong Zou, Tao Zheng, and Zhenyu Chen. 2024. DevMuT: Testing Deep Learning Framework via Developer Expertise-Based Mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1533–1544. https://doi.org/10.1145/3691620.3695523

[95] Zhumakhan Nazir, Vladislav Yarovenko, and Jurn-Gyu Park. 2023. Interpretable ML enhanced CNN Performance Analysis of cuBLAS, cuDNN and TensorRT. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 1260–1265.

[96] Mahdi Nejadgholi and Jinqiu Yang. 2019. A study of oracle approximations in testing deep learning libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 785–796.

[97] Yuanping Nie, Xiong Xiao, Bing Yang, Hanqing Li, Long Luo, Hongfang Yu, and Gang Sun. 2024. Python Coverage Guided Fuzzing for Deep Learning Framework. In *2024 International Conference on Electronic Engineering and Information Systems (EEISS)*. IEEE, 1–6.

[98] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 902–912.

[99] Peter Oehlert. 2005. Violating assumptions with fuzzing. *IEEE Security & Privacy* 3, 2 (2005), 58–62.

[100] Işıl Öz. 2024. Quantitative Performance Analysis of BLAS Libraries on GPU Architectures. *Dokuz Eylül Üniversitesi Mühendislik Fakültesi Fen ve Mühendislik Dergisi* 26, 76 (2024), 40–48.

[101] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[102] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350* (2021).

[103] 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.

[104] Alexander Prochnow and Jinqiu Yang. 2022. DiffWatch: watch out for the evolving differential testing in deep learning libraries. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 46–50.

[105] Lili Quan, Qianyu Guo, Xiaofei Xie, Sen Chen, Xiaohong Li, and Yang Liu. 2022. Towards understanding the faults of javascript-based deep learning systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[106] Luyao Ren, ZiHeng Wang, Yingfei Xiong, Li Zhang, Guoyue Jiang, and Tao Xie. 2023. Effective Random Test Generation for Deep Learning Compilers. *arXiv preprint arXiv:2302.00842* (2023).

[107] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[108] Richard Schumi and Jun Sun. 2022. ExAIS: executable AI semantics. In *Proceedings of the 44th International Conference on Software Engineering*. 859–870.

[109] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[110] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

[111] Qingchao Shen, Yongqiang Tian, Haoyang Ma, Junjie Chen, Lili Huang, Ruifeng Fu, Shing-Chi Cheung, and Zan Wang. 2024. A Tale of Two DL Cities: When Library Tests Meet Compiler. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 305–316.

[112] Xiangzhong Shen, Jieyi Zhang, Xiaonan Wang, Hongfang Yu, and Gang Sun. 2021. Deep learning framework fuzzing based on model mutation. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 375–380.

[113] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. Acetest: Automated constraint extraction for testing deep learning operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 690–702.

[114] ABC7.com staff. [n. d.]. *Uber gives up testing of self-driving cars in California in wake of fatal Arizona crash.* https://abc7.com/self-driving-uber-crash-video-pedestrian-hit-by-car-autonomous-vehicles/3269690/

[115] Y Sun. 2020. *Tesla and PyTorch: PyTorch Developer Conference Highlights.* https://medium.com/data-science-bootcamp/tesla-and-pytorch-pytorch-developer-conference-highlights-part-3ed36f2c9d5e

[116] Yifan Sun, Saoni Mukherjee, Trinayan Baruah, Shi Dong, Julian Gutierrez, Prannoy Mohan, and David Kaeli. 2018. Evaluating performance tradeoffs on the radeon open compute platform. In *2018 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 209–218.

[117] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2024. Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow. *Empirical Software Engineering* 29, 1 (2024), 10.

[118] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference*. IEEE, 270–279.

[119] TensorFlow. 2020. *Learn how TensorFlow solves real, everyday machine learning problems.* https://www.tensorflow.org/about/case-studies

[120] Takumi Uezono, Yi He, and Yanjing Li. 2022. Achieving automotive safety requirements through functional in-field self-test for deep learning accelerators. In *2022 IEEE International Test Conference (ITC)*. IEEE, 465–473.

[121] Tatiana Castro Vélez, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja. 2022. Challenges in migrating imperative deep learning programs to graph execution: an empirical study. In *Proceedings of the 19th international conference on mining software repositories*. 469–481.

[122] Gaurav Verma, Swetang Finviya, Abid M Malik, Murali Emani, and Barbara Chapman. 2022. Towards neural architecture-aware exploration of compiler optimizations in a deep learning {graph} compiler. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*. 244–250.

[123] Gaurav Verma, Yashi Gupta, Abid M Malik, and Barbara Chapman. 2021. Performance evaluation of deep learning compilers for edge inference. In *2021 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 858–865.

[124] Chaojin Wang, Jian Shen, Chunrong Fang, Xiangsheng Guan, Kaitao Wu, and Jiang Wang. 2020. Accuracy measurement of deep neural network accelerator via metamorphic testing. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 55–61.

[125] 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.

[126] Jun Wang, Guanping Xiao, Shuai Zhang, Huashan Lei, Yepang Liu, and Yulei Sui. 2023. Compatibility issues in deep learning systems: Problems and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 476–488.

[127] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. Qdiff: Differential testing of quantum software stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 692–704.

[128] Zihan Wang, Pengbo Nie, Xinyuan Miao, Yuting Chen, Chengcheng Wan, Lei Bu, and Jianjun Zhao. 2023. GenCoG: A DSL-Based Approach to Generating Computation Graphs for TVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 904–916.

[129] 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.

[130] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: fault localization for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 251–262.

[131] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[132] Moshi Wei, Nima Shiri Harzevili, YueKai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2024. Demystifying and Detecting Misuses of Deep Learning APIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[133] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[134] Jiawei Wu, Senyi Li, Junqiang Li, Long Luo, Hongfang Yu, and Gang Sun. 2022. DeepCov: Coverage Guided Deep Learning Framework Fuzzing. In *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*. IEEE, 399–404.

[135] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 56–68.

[136] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28.

[137] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188.

[138] Danning Xie, Jiannan Wang, Hung Viet Pham, Lin Tan, Yu Guo, Adnan Aziz, and Erik Meijer. 2024. CEDAR: Continuous Testing of Deep Learning Libraries. In *International Conference on Software Analysis, Evolution, and Reengineering,*. IEEE.

[139] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing automatic differentiation in deep-learning libraries. *arXiv preprint arXiv:2302.04351* (2023).

[140] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. 2022. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology* 151 (2022), 107004.

[141] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.

[142] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1159–1170.

[143] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–115.

[144] Xufan Zhang, Jiawei Liu, Ning Sun, Chunrong Fang, Jia Liu, Jiang Wang, Dong Chai, and Zhenyu Chen. 2021. Duo: Differential fuzzing for deep learning operators. *IEEE Transactions on Reliability* 70, 4 (2021), 1671–1685.

[145] Xiaoyu Zhang, Chao Shen, Chenhao Lin, Qian Li, Qian Wang, Qi Li, and Xiaohong Guan. 2022. The Testing and Repairing Methods for Machine Learning Model Security. *ACTA ELECTONICA SINICA* 50, 12 (2022), 2884.

[146] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–412.

[147] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. Autotrainer: An automatic dnn training problem detection and repair system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 359–371.

[148] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Shiwei Wang, and Chao Shen. 2024. CITADEL: Context Similarity Based Deep Learning Framework Bug Finding. *arXiv preprint arXiv:2406.12196* (2024).

[149] 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.

[150] Yihua Zhang, Pingzhi Li, Junyuan Hong, Jiaxiang Li, Yimeng Zhang, Wenqing Zheng, Pin-Yu Chen, Jason D Lee, Wotao Yin, Mingyi Hong, et al. [n. d.]. Revisiting Zeroth-Order Optimization for Memory-Efficient LLM Fine-Tuning: A Benchmark. In *Forty-first International Conference on Machine Learning*.

[151] Zhiyi Zhang, Pu Wang, Hongjing Guo, Ziyuan Wang, Yuqian Zhou, and Zhiqiu Huang. 2021. Deepbackground: Metamorphic testing for deep-learning-driven image recognition systems accompanied by background-relevance. *Information and Software Technology* 140 (2021), 106701.

[152] Hao Zhong. 2022. Enriching compiler testing with real program from bug report. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[153] Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 317 (Oct. 2024), 27 pages. https://doi.org/10.1145/3689757

[154] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Xiaofei Xie, and Rui Chang. 2025. My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P)*. IEEE, IEEE.

[155] 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.