DEEPLANCET: Effectively Detecting Deep Learning Library Bugs via LLM-assisted Testcase Generation

Zihao Luo Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
lzhcsu@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—Deep learning libraries such as PyTorch and TensorFlow are essential for building security-critical downstream deep learning applications. Bugs in these libraries compromise their correctness, robustness, and security, thereby undermining the reliability of downstream applications. Unfortunately, effectively detecting bugs in deep learning library remains challenging, as existing approaches often fail to generate effective test cases due to their inability to synthesize complex input constraints that govern deep learning library functions.**

In this paper, we present DEEPLANCET, the first approach for effectively detecting deep learning library bugs by leveraging large language models (LLMs) to systematically parse documentation and thereby assist in the generation of high-quality test cases. Our key observation is that mainstream deep learning libraries typically provide comprehensive and well-structured documentation, which contains detailed descriptions of input constraints that we can effectively synthesize and leverage to generate syntactically valid and semantically correct test cases. Specifically, we first synthesize function constraint descriptions from the documentation by leveraging LLMs. We then generate rigorous constraints which are leveraged to generate test cases through an attribution-based approach in Python. Finally, we employ a differential testing approach on CPU and GPU to detect bugs. We build a software prototype for DEEPLANCET, and our evaluation results demonstrate that DEEPLANCET is effective in uncovering previously unknown real-world bugs: it successfully uncovers 20 bugs in the latest release of PyTorch, including 7 previously unknown ones. Moreover, we compare DEEPLANCET with DocTer, a state-of-the-art technique that also leverages documentation for constraint extraction, and the results indicate that DEEPLANCET can extract more comprehensive constraints, thereby uncovering 3 more bugs that were missed by DocTer.

***Index Terms*—Deep learning libraries, Bug detection, Large Language Models**

I. INTRODUCTION

Deep learning libraries [1] such as PyTorch and TensorFlow are fundamental to modern artificial intelligence systems, powering many security-critical downstream applications including aircraft collision avoidance systems [2], disease diagnosis [3], and autonomous driving [4]. Unfortunately, these libraries inevitably contain bugs [5], due to their large codebase and complex logic. Such bugs not only undermine the security of these deep learning libraries themselves but also propagate to dependent applications, posing severe risks to users' property and personal safety [6]. For instance, a software bug in Uber's self-driving deep learning system led to a pedestrian's death [7]. Therefore, detecting bugs in deep learning libraries is of critical importance.

Fuzz testing is a promising software testing technique [8] to detect software bugs and has been leveraged to test deep learning libraries [9]. The test process typically consists of three interconnected stages: 1) test case generation, 2) execution and monitoring, and 3) anomaly analysis. First, test inputs are generated to explore diverse execution paths in the target program. Second, these test inputs are executed, often in a controlled environment, to continuously monitor and collect runtime behavior such as code coverage to guide the test process. Finally, any anomalies such as memory violations or unexpected crashes are further collected and analyzed to reveal underlying bugs or potential security vulnerabilities. Among these stages, test case generation is crucial, as the quality of generated inputs determines path coverage and directly impacts bug detection effectiveness [10]. To this end, prior studies have proposed various strategies to generate more effective test cases, including grammar-based generation, constraint solving, and feedback-guided mutation [11].

Unfortunately, despite recent progress, generating high-quality test inputs to effectively test deep learning libraries remains a significant challenge. One major reason is that most bugs occur in core function logic and appear only when inputs satisfy strict validity constraints [12]. Consequently,random fuzzing techniques often fail to pass these checks, limiting their ability to explore critical execution paths [6]. To address this challenge, a key idea is to first define input specifications [6] that are conditions the generated test cases should satisfy, then leverage the input specifications to guide the generation of syntactically valid and semantically meaningful test inputs. Although a substantial body of recent studies has achieved notable progress by following this idea [12]–[18], they remain constrained by two major limitations: (1) limited scalability and (2) inadequate comprehension. First, some studies suffer from scalability issues to support large-scale function testing. For instance, Predoo [14] and aNNoTest [15] depend on manually defined constraints, which hinders scalability for large-scale testing. Second, some studies face challenges in generating comprehensive and valid constraints from documentation or source code, as natural language documents are often complex and ambiguous, and source code

* The corresponding author.

is difficult to analyze. For example, DocTer [16] adopts a rule-based approach to extract constraints from function descriptions and documentation, but frequently produces incomplete constraints. Similarly, ACEtest [12] leverages symbolic execution to derive constraints from source code; however, it faces significant challenges when handling highly complex code structures, which often results in incomplete or inaccurate constraints.

In this paper, we propose an LLM-assisted approach to generate valid and meaningful test cases by synthesizing input constraints from documentation.

Our key observation is that, after years of development, mainstream libraries such as PyTorch and TensorFlow provide mature documentation with detailed specifications, constraints, and counterexamples. Automatically extracting constraints from documentation is both beneficial and feasible. However, documentation is typically written in natural language, which is intrinsically ambiguous, making constraint synthesis challenging. To address this challenge, we propose to leverage LLMs, which excel at document understanding and question answering [19], to synthesize constraints. This approach surpasses manual and rule-based methods by enabling more flexible and accurate generation of input constraints from natural language documentation.

With this key observation, we first leverage LLMs to extract semi-structured constraint descriptions in a well-tuned JSON format and prompt from the deep learning library documentation in natural language. Then, following the insight of prior work on structured data generation [20], we synthesize these semi-structured descriptions into structured and semantically equivalent constraint representations, which guide the generation of high-quality input test cases.

Another challenge we must tackle is that LLMs may occasionally produce syntactically incorrect or semantically invalid constraints due to hallucinations [21]. To address this, we introduce a lightweight validation step to ensure structural correctness, removing incomplete or ill-formed parameters, and propose an error-feedback-guided refinement process to correct inaccurate constraints.

We then employ a fuzz testing engine to generate test cases that satisfy the synthesized constraints. Furthermore, we leverage differential testing to compare outputs from CPU and GPU backends, identifying potential bugs when inconsistencies arise.

In realizing the whole process, we address two technical challenges. **C1**: How to rigorously express the function constraints? We propose CONSLANG, a formal language in context-free grammar, to concisely capture and express the constraints that are used in test case generation. **C2**: How to effectively handle constraint errors during testing? We propose an error feedback-guided constraint refinement approach, which iteratively adjusts and improves the previously synthesized constraints by monitoring and analyzing runtime error messages.

We implement a prototype for our approach, dubbed DEEPLANCET, and conduct extensive experiments to evaluate its effectiveness and usability. We first systematically test the latest PyTorch [22], one of the most popular and important deep learning libraries in production and has been well-tested. We successfully uncover 20 bugs in PyTorch, including 7 previously unknown ones. We have reported these issues to the PyTorch development team, and 3 issues have been confirmed as real bugs while the others are still being triaged. We then conduct a comparative study with the state-of-the-art technique DocTer [16]. The evaluation results demonstrate that, under the same setting of generating 1,000 test cases for each of ten randomly selected deep learning functions respectively, our approach DEEPLANCET generates 43.4% more valid test cases than DocTer, and uncovers 3 bugs that have been missed by DocTer, thereby highlighting the effectiveness of our approach.

In summary, this paper makes the following contributions:

- We propose the first LLM-assisted approach to effectively detect test deep learning library bugs, by generating valid test cases through synthesizing input constraints from deep learning library documentation.
- We design and implement a software prototype DEEPLANCET to validate our approach.
- We conduct extensive experiments to demonstrate that DEEPLANCET is effective in uncovering real bugs in deep learning libraries, outperforming state-of-the-art approaches.

The remainder of this paper is organized as follows. Section II introduces the background. Section III outlines the motivations and challenges. Section IV presents our approach. Section V presents the experimental evaluation of DEEPLANCET. Section VI discusses limitations and future directions. Section VII reviews related work, and Section VIII concludes.

II. BACKGROUND

To be self-contained, in this section, we present the necessary background knowledge on deep learning libraries (§ II-A) and the constraints governing deep learning libraries (§ II-B).

A. Deep Learning Libraries

Deep learning libraries such as PyTorch [22] and TensorFlow [23] are essential software infrastructures that are leveraged to build, train, and deploy deep learning models. They encapsulate complex computations, provide automatic differentiation, and support hardware acceleration, allowing neural networks to be implemented without manual low-level coding. These libraries underpin a wide range of applications, including computer vision, natural language processing, and scientific computing [24], and are increasingly used in security-critical domains such as aircraft collision avoidance systems [2], disease diagnosis [3], and autonomous driving [4]. Ensuring their correctness and reliability is therefore critical. However, deep learning libraries are not immune to bugs. Their large codebases, sophisticated optimizations [25] and interaction with heterogeneous hardware backends make numerical inconsistencies, logical errors, and performance anomalies inevitable, potentially undermining system correctness and trustworthiness.

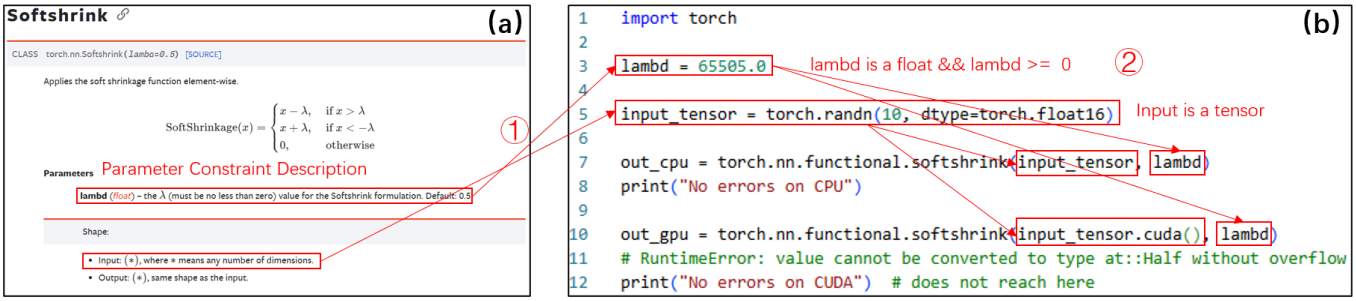

Fig. 1: This figure shows a bug we revealed in the `softshrink` funtion of PyTorch, which we reported to the PyTorch developers who have confirmed and fixed it. Fig. 1(a) shows the documentation of this function, comprising its input constraints. And Fig. 1(b) presents a minimized code snippet to trigger the bug.

B. Input Constraints

Function constraints specify the conditions that the function arguments should satisfy. Specifically, the function constraints in deep learning libraries mainly fall into two categories: 1) data type constraints, and 2) value range constraints. Data type constraints define the argument types, while value range constraints define the allowable numeric intervals. These constraints are crucial for testing, as only test cases that satisfy them can pass validity checks and exercise the core functional logic.

To put the above discussion into perspective, we present in Fig. 1(a) a representative PyTorch function `torch.nn.Softshrink` and its input constraints in the documentation [1]. Specifically, the documentation specifies the data type constraints that the first input parameter `input` should be of type `torch.Tensor` and the second input parameter `lambd` *should* be of type `float`. In addition, the documentation also specifies the value range constraints that the second input parameter `lambd` *must* be no less than `0.0`.

The `torch.nn.Softshrink` function is not unique to have such constraints in documentation, indeed, most functions in PyTorch and other deep learning libraries such as TensorFlow, JAX, PaddlePaddle, and ONNX runtime, also have similar constraints, which are essential guidance for end developers.

III. Motivations and Challenges

In this section, we present our motivation (§ III-A) through a running example, followed by the challenges and our solutions (§ III-B).

A. Motivations

Generating high-quality test cases for the functions in deep learning libraries poses significant challenges, because these functions impose strict input constraints, many specific to the deep learning domain (as evidenced by the example function in Fig. 1(a)). Existing approaches that generate random test cases often produce many invalid inputs, failing to satisfy these constraints. To make this point concrete, we present in Fig. 1(b), a code snippet to differential test the `torch.nn.softshrink` function, for which we have revealed a previously unknown bug which has been confirmed and fixed by the PyTorch developers [2]. Specifically, the arguments `lambd` in the test code for CPU (line 7) and GPU (line 10) should satisfy both the data type constraint (*i.e.*, type `float`) and the value range constraint (*i.e.*, `lambd` ≥ 0). Unfortunately, a randomly generated value for `lambd` may violate either the data type constraint (*e.g.*, `lambd` = "`hello`") or the value range constraint (*e.g.*, `lambd` $= -1$) (as evidenced by our quantitative evaluation results of current approaches' low valid rates in Section V).

One might assume that satisfying the data type constraint is straightforward because type annotations are syntactic. However, Python is dynamically typed, and most deep learning functions provide no type information for arguments. Although PEP 484 introduces type hints, these are rarely used in deep learning libraries and provide limited help.

To overcome this, we leverage function documentation, which often contains explicit data type and value range constraints. For example, the documentation for the `torch.nn.Softshrink` function in Fig. 1(a) contains sufficient information comprising the data type and value range constraints. By systematically extracting these constraints and generating test cases that conform to them, we can more effectively test deep learning libraries. But synthesizing constraints and converting them into executable code is challenging, as documentation targets developers rather than automated test harnesses. To address this, we propose DeepLancet, a novel approach that uses LLMs to extract constraints from documentation and employs property-based Python annotations to generate test cases for differential testing across CPU and GPU backends. Our approach effectively uncovers bugs by generating high-quality, constraint-compliant test cases. As Fig. 1(b) shows, our approach reveals a bug in the `softshrink` when the argument `lambd` exceeds 65505.0 (line 5), the program executes normally on the CPU backend but raises a `RuntimeError` exception on the GPU backend. We have reported this issue to the PyTorch developers who

[1] https://docs.pytorch.org/docs/stable/generated/torch.nn.Softshrink.html

[2] See our bug report at: https://github.com/pytorch/pytorch/issues/155671

have confirmed this bug and fixed it. As a comparison, the state-of-the-art tool DocTer fails to reveal this bug due to its overlooking of the value range constraint.

B. Challenges

Developing an effective approach to generate more effective test cases by leveraging documentations needs to tackle several technical challenges.

C1: How to rigorously express the constraints? Although documentation provides detailed descriptions of input constraints, it is typically written in natural languages that are informal and ambiguous. The documentation is an ad hoc data source [20] for which useful data analysis and transformation tools are not readily available. As a result, concisely and accurately expressing those complex input constraints remains a significant challenge.

Solution: To address this challenge, we propose CONSLANG, a formal language in context-free grammar, to concisely capture and express the constraints that are used in test case generation.

C2: How to effectively handle constraint errors during testing? The constraints extracted by LLMs may contain inaccuracies due to the well-known hallucination issues. Therefore, effectively identifying and correcting such errors remains a challenge.

Solution: To address this challenge, we propose an error-feedback-guided constraint refinement approach to identify and correct inaccurate constraints generated by LLMs. Our approach leverages runtime feedback to improve the accuracy and reliability of subsequent test case generation.

IV. Approach

In this section, we present our approach for generating more effective test cases for deep learning libraries testing. We begin with an overview of (§ IV-A), then detail the design and implementation of each component (§ IV-B to § IV-E), respectively.

A. Overview

We present an overview of DeepLancet's workflow in Fig. 2, comprising three key components: constraints generation, test case generation, and differential testing. First, Constraint generation (❶) analyzes the target function along with its accompanying documentation that describes the input constraints. It then employs a LLM to automatically extract the relevant input constraints from the documentation. Second, test file generation (❷) utilizes the extracted constraints to generate test cases with constraints as annotations. Third, differential testing (❸) generates concrete input instances based on the provided annotations, and test the target function using these inputs on both GPU and CPU backends. The execution results from both backends are then compared to determine whether any inconsistencies manifest. Finally, the comparison results are fed back into the system to further refine the constraint extraction and test case generation for the next round of test.

B. Constraint Generation

Constraint generation takes as input a function name along with its corresponding documentation that describes the expected input constraints, and extracts input constraints from documentation and transforms the constraints from informal natural language descriptions into a structured internal representation.

Internally, constraint generation leverages LLMs to extract the relevant input constraints of data types and value ranges and convert them into a structured format. Moreover, the constraint generation employs a web search engine that the LLM leverages to retrieve online resources to improve accuracy. The process of constraint extraction comprises four steps of constraint query, description retrieval, implicit constraint generation, and result generation, which are discussed in the following, respectively.

Constraint Query. First, we use the function name to query the LLM regarding the input constraints that the function should satisfy. To this end, we design a prompt to interact with the target LLM. Specifically, we leverage the few-shot Chain of Thought (CoT) prompting technique [26] as Fig. 3 illustrates. In our Few-shot chain-of-thought (CoT), we provide the LLMs with a limited number of examples that illustrate a step-by-step reasoning process, thereby encouraging the model to emulate a logical progression in problem solving. To enhance the correctness of constraint extraction, we build a set of examples to infer constraints step-by-step from the function documentation.

Description Retrieval. In this step, we adopt the Retrieval-Augmented Generation (RAG) technique [27] to leverage constraint information from the function's documentation. Specifically, we leverage RAG to improve the reliability of LLMs responses by retrieving and referencing authoritative external knowledge sources prior to generating an answer. In our context, the documentation describing function constraints serves as an external knowledge base. By consulting this documentation, the system can obtain more accurate and reliable constraint information.

Implicit Constraints Generation. In this step, we drive the LLMs to search for additional information via web search engines, regarding the constraints of the deep learning functions. By conducting real-time online searches, we can collect more comprehensive and constrained information that enhances the reliability of the LLM responses. In particular, our use of web searches allows the model to uncover hidden or implicit constraints in function parameters such as value range constraints that are enforced by the implementation but not explicitly documented. For example, many deep learning functions include parameters regarding input or output sizes (*e.g.*, `kernel_size`, `stride`, and `padding`) that are implicitly required to be positive integers. These implicit constraints are often crucial for correct test case generation, and retrieving them from external sources can significantly improve the quality and accuracy of the inferred constraints.

Result Generation. After processing the query and retriev-

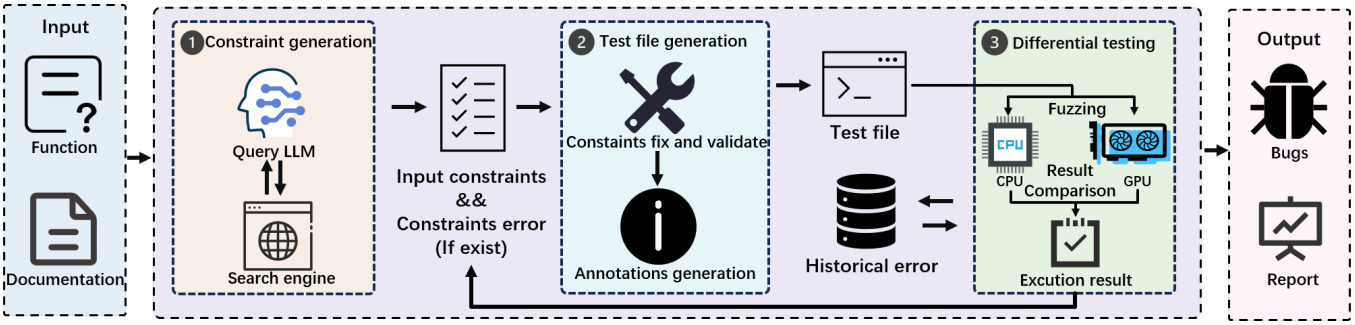

Fig. 2: An overview of DEEPLANCET's workflow.

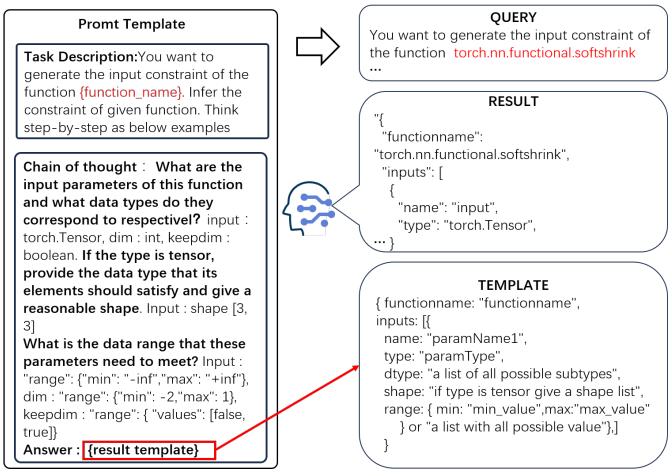

Fig. 3: Prompt and chat of constraints extraction.

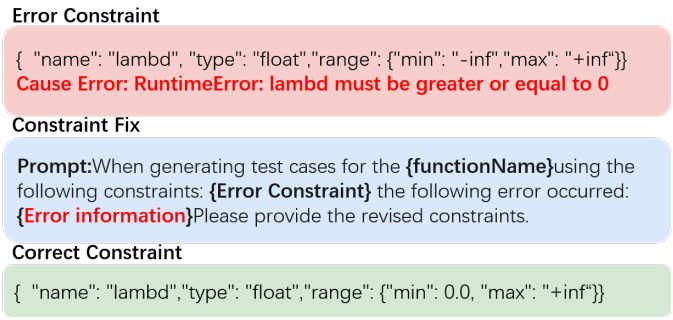

Fig. 4: Prompt and chat of constraints fixing.

ing relevant information from the function's documentation, DEEPLANCET generates the corresponding input constraints for the target function. It is worth noting that to handle these constraints more conveniently in subsequent phases, we have the LLM return the constraints in a dedicated JSON format we design through a given return result template as Fig. 3 shows. Our analysis primarily focuses on two types of constraints that are crucial for test case generation: data types and value ranges, that are reflected by the corresponding JSON fields such as `type`, `dtype`, and `range`.

C. Test Case Generation

Test case generation takes as input the LLM-generated constaints and produces test files augmented with constraint annotations.

First, as our threat model specifies, the constraints returned by the LLMs may not be trustworthy, thus we perform a validation of the input constraints to ensure that their structure is well-formed and that all specified types are supported. To this end, we design a rule-based validator that checks the constaints in JSON format, verifies required fields, and filters out constraints with unsupported data types or function arguments. Furthermore, for any constraint-related error messages encountered during execution, we propose an error-feedback-guided approach to leverage LLMs to rectify erroneous constraints based on the corresponding error messages. For example, as Fig. 4 illustrates, suppose an error were issued in the initial constraint specification for the parameter `lambd` when testing the `torch.nn.functional.softshrink` function, the generated test cases would be rejected by PyTorch's internal parameter validation logic because the parameter `lambd` falls outside the valid range. Fortunately, the resulting error message clearly indicates that the acceptable value for `lambd` must be greater than or equal to zero, so the LLM can leverage this error feedback to rectify the original constraint accordingly to produce a rectified constraint conforming to the expected input specification. Finally, the rectified constaint can be used in the subsequent rounds of testing without triggering the same error.

We then convert the structured constraints returned from the LLM into our specially designed annotation language, CONSLANG, as Fig. 5 shows. We design the CONSLANG language to describe the input constraints of the deep learning library and to write annotations in the test cases. Unlike general-purpose constraint specification languages, CONSLANG can capture key properties commonly found in deep learning functions, such as tensor shapes, data types, and value ranges. Its syntax is given by a syntax-free grammar and is designed to be both human-readable and machine-parsable, facilitating seamless integration with fuzzing engines and constraint validation tools.

Specicifically, an annotation a starts with `@Require` and defines one or more constraints. A constraint c is defined by assigning a parameter name to either an ElementConstraint

Annotation	a	::=	`@Require(`$\overrightarrow{c}$`)`
Constraint	c	::=	`paramName =` e
		|	`paramName =` o
ElementConstraint	e	::=	`integers(`i, j`)`
		|	`floats(`i, j`)`
		|	`booleans()`
OtherConstraint	o	::=	`lists(`$\overrightarrow{c}, s, x$`)`
		|	`one_of(`$\overrightarrow{c}$`)`
		|	`builds(ClassName,` $\overrightarrow{c}$`)`
		|	`just(Value)`
		|	`none()`
		|	`tensor(type=`t`, shape=`p`)`
MinSize	s	::=	`min_size =` n
MaxSize	x	::=	`max_size =` n
MinValue	i	::=	`min_value =` n
MaxValue	j	::=	`max_value =` n
Const	n	::=	`INT | FLOAT | -Inf | +Inf`
Types	u	::=	$t \mid t, u$
Type	t	::=	`f16 | f32 | bf16 | i16 | i32`
		|	`i64 | c64 | c128`
Shape	p	::=	(d)
Dims	d	::=	$n \mid n, d$

Fig. 5: The syntax of ConsLang.

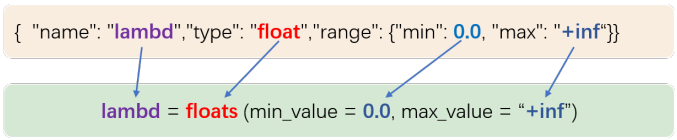

Fig. 6: ConsLang conversion example.

or OtherConstraint. ElementConstraint e specifies primitive values such as `integers` and `floats` with value range bounds, or `booleans`. OtherConstraint o captures complex structures, including `lists` with size bounds, `one_of` for enumerations, `builds` for object construction, `just` for constants, `none` for null values, and `tensor` for tensors with specified types and shapes. `MinSize` s and `MaxSize` x define lower and upper bounds on collection sizes, respectively. `MinValue` i and `MaxValue` j impose numerical constraints on primitive values, respectively. `Const` n denotes constants such as `INT`, `FLOAT`, `-Inf`, or `+Inf`. `Types` u consist of one or more `Type` entities, where each `Type` t denotes a numeric type supported by PyTorch, including common floating-point, integer, and complex formats. `Shape` d specifies tensor dimensionality, with `Dims` d denoting one or more dimensions defined by `Const`. Together, these constructs establish a concise yet expressive grammar for specifying input constraints in deep learning functions.

We build a syntax-directed translation from the constraints' JSON representation to the ConsLang representation. This translation process is largely straightforward, as these two languages have similar structures and fields. We provide an illustration of this translation process in Fig. 6.

D. Differential Testing

Differential testing takes as input the annotated test file generated by the test file generation. We employ a fuzzing engine to generate concrete function inputs that satisfy the specified annotations. We perform differential testing on the target function using these inputs to detect potential bugs. After this round of testing, we incorporate the test results into the system to refine and guide subsequent constraint annotation generation. The differential testing primarily comprises two steps of test execution and error analysis.

Test Execution. We first stochastically generate input cases based on the annotations provided in the test file, supplemented by historical error information. We then execute each input concurrently on both CPU and GPU backends, leveraging deep learning libraries' inherent support for multiple independent backends. Finally, we differentially compare the outputs generated from the two backends, and apply a predefined error tolerance threshold to determine acceptable output differences. If the discrepancy between the outputs exceeds this threshold, we regard it as a potential indicator of a functional inconsistency or a vulnerability in that backend.

Error analysis. We observe that the error messages produced during testing are often highly informative and can be leveraged to further improve the framework. We categorize the error messages into three types of program crashes, runtime result inconsistencies, and input constraint violations, based on the output results. For input constraint violations, we feed the corresponding error messages to the LLM, then leverage the LLM to interpret and refine the previously generated input constraints, thereby ensuring greater completeness. We employ manual inspection to analyze the information associated with program crashes and result inconsistencies to further investigate their underlying root causes. Based on the identified data types and value ranges associated with these errors, we add the corresponding test cases to a historical error database. During test input generation, the DeepLancet leverages this historical vulnerability database to create inputs that are structurally or semantically similar to those that triggered known bugs, thereby enhancing the likelihood of generating new edge test cases.

E. Prototype Implementation

To validate our approach, we design and implement a software prototype for DeepLancet, comprising three components. First, we implement the constraint generation by leveraging GPT-4, a state-of-the-art LLM known for its advanced natural language understanding and reasoning abilities. Second, we implement the test file generation in 1.5k lines of Python code. Third, we implement differential testing on top of Hypothesis [28], a Python-based property-driven testing framework, to generate diverse and constraint-compliant test cases.

V. Evaluation

To evaluate the effectiveness of DeepLancet, we conduct extensive evaluations. Specifically, our evaluation aims to answer the following research questions:

RQ1: Effectiveness. Given that DeepLancet is designed to detect bugs in production deep learning libraries, is DeepLancet effective in achieving this goal?

TABLE I: Number of known / new / all bugs / buggy APIs uncovered by DEEPLANCET.

Category	Known Bugs	New Bugs	All Bugs	Buggy APIs
#No.	13	7	20	55

RQ2: Usefulness. Whether the extracted constraints can indeed improve the quality and relevance of the generated test cases?

RQ3: Ablation Study. How does the component of constraints fixing and validation in DEEPLANCET, contribute to the overall effectiveness?

RQ4: Comparative study. Does DEEPLANCET outperform state-of-the-art approaches for deep learning library testing that also leverage input constraints from documentation?

All experiments and measurements are conducted on a server equipped with an Intel Core i7 CPU (8 physical cores), 16 GB of RAM, and an NVIDIA GeForce RTX 3060 GPU, running Ubuntu 20.04.

A. Datasets

We choose one of the most popular deep learning libraries (PyTorch 2.7.0) as the test subject, because of the following reasons. First, PyTorch provides comprehensive and well-maintained documentation, including detailed descriptions of functions and their input constraints. This makes it a suitable benchmark for evaluating the capability of our system. Second, its widespread adoption in both academia and industry ensures that testing results are representative and practically meaningful. Third, PyTorch has undergone extensive testing and quality assurance [13], [29], [30] , including unit tests, integration tests, and contributions from a large open-source community. As such, it serves as a strong baseline for assessing our system's capability in uncovering real-world but previously overlooked bugs in a mature and rigorously tested framework.

It should be noted that while we focus on testing PyTorch as our first step, our approach (as Fig. 2 shows) in this work does not tie to PyTorch but can be applied to any other deep learning libraries with reasonable documentation.

B. RQ1: Effectiveness

To answer RQ1 by investigating DEEPLANCET's effectiveness in detecting real-world bugs in production deep learning libraries, we first conduct an evaluation of DEEPLANCET on PyTorch (v2.7.0), the latest stable version at the time of our evaluation.

We present the evaluation results in Table. I. In summary, DEEPLANCET successfully uncovers a total of 20 bugs in PyTorch, including 13 previously known bugs and 7 previously unknown ones that have not been reported prior to our study. It is worth noting that the 20 bugs uncovered by DEEPLANCET impact a wider range of 55 functions that contain them. This result demonstrates DEEPLANCET's strong capability in uncovering real-world bugs.

Furthermore, we have reported these issues uncovered by DEEPLANCET to PyTorch developers and among them 3 issues have been confirmed as real bugs and have been fixed while the others are still being triaged.

C. RQ2: Usefulness

To answer RQ2 by investigating the practical usefulness of DEEPLANCET-generated constraints to uncover real-world bugs by producing more effective test cases. We conduct a comparative evaluation of DEEPLANCET against DocTer [16], regarding valid test case generation. It should be noted that DocTer, while powerful, does not leverage LLM-assisted constraint generation as this study does.

Proportion of Valid Cases. We first measure the proportion of valid test cases and use it as a metric for evaluating the effectiveness and precision of the constraint generation process. The proportion is calculated as $Proportion = \frac{\#ValidCases}{\#TotalCases}$, where $\#TotalCases$ refers to the total number of test case generated whereas $\#ValidCases$ stands for the valid ones among all the test cases. This metric $Proportion$ reflects how well the generated constraints capture the intended input space. A higher $Proportion$ of valid test cases indicates more precise and comprehensive constraint extraction. For this evaluation, we randomly select functions from PyTorch. For each function, we generate 1,000 random input test cases both for DEEPLANCET and DocTer, respectively. We then record the valid test cases among all the 1,000 test cases generated by DEEPLANCET, and DocTer and compute the $Proportion$ for them, respectively.

We present in Table II the comparison results for 10 representative PyTorch functions. Specifically, the columns **DocTer** and **DEEPLANCET** present the number and proportions of valid test cases for DocTer and our DEEPLANCET, respectively. Overall, our approach, DEEPLANCET, yields a proportion of 100% valid cases for 7 functions whereas DocTer only generates all valid tests for 2 functions. Overall, DEEPLANCET generates a total of 9,193 valid test cases which gives a valid proportion of 91.9% whereas DocTer only generate 4,845 valid test cases giving a valid proportion of 48.5%. Therefore, DEEPLANCET yields a 43.4% improvement regarding the overall proportion of valid test cases, consistently and significantly surpassing DocTer. Moreover, across all evaluated functions, the validity proportion of test cases generated by DEEPLANCET is always equal or higher than that of DocTer. This persistent superiority reflects the enhanced comprehensiveness and granularity of the constraints derived by DEEPLANCET. Consequently, our method facilitates more precise and thorough test case generation, effectively capturing the intricate semantic requirements of deep learning library functions, which in turn contributes to improved robustness and reliability in bug detection.

It is worth noting that DEEPLANCET's validity proportion for the `bincount`, `eye`, and `conv1d` functions are 76.7%, 75.6%, and 67.0%, respectively, and are thus inaccurate. We conduct a manual inspection with the three functions and reveal two key root causes. The first one is that, due to the close relationship between their input parameters and memory allocation, the generated test cases often request

TABLE II: Comparative results of the three types of constraints.

Test Case	Function Name	#Valid tests (Proportion) (DocTer)	#Valid tests (Proportion) (DEEPLANCET$^-$)	#Valid tests (Proportion) (DEEPLANCET)
1	torch.acos	1,000 (100%)	1,000 (100%)	1,000 (100%)
2	torch.as_strided	330 (33.0%)	572 (57.2%)	1,000 (100%)
3	torch.bincount	221 (22.1%)	767 (76.7%)	767 (76.7%)
4	torch.chunk	120 (12.0%)	1,000 (100%)	1,000 (100%)
5	torch.cummax	238 (23.8%)	1,000 (100%)	1,000 (100%)
6	torch.eye	756 (75.6%)	756 (75.6%)	756 (75.6%)
7	torch.nn.functional.softshrink	549 (54.9%)	1,000 (100%)	1,000 (100%)
8	torch.nn.functional.conv1d	116 (11.6%)	227 (22.7%)	670 (67.0%)
9	torch.sin	1,000 (100%)	1,000 (100%)	1,000 (100%)
10	torch.topk	515 (51.5%)	1,000 (100%)	1,000 (100%)
All	-	**4,845 (48.5%)**	**8,322 (83.2%)**	**9,193 (91.9%)**

memory exceeding the system's maximum capacity, resulting in execution errors for the functions `bincount` and `eye`. Moreover, another root cause is the invalid inputs from complex interdependent constraints among its parameters, leading to failures in the `conv1d` function. It worthy note that while DEEPLANCET does not produce 100% valid test cases for these 3 functions, it also exhibits higher validity proportion than DocTer.

Overall, these results indicate that the constraints extracted by DEEPLANCET are more comprehensive and precise compared to those produced by DocTer. Together, these improvements facilitate generating a larger number of valid test cases, thereby demonstrating the superior capability of DEEPLANCET in capturing accurate input constraints and enhancing downstream testing effectiveness.

Case Study. To clearly illustrate the advantages of the constraints synthesized by DEEPLANCET, we present a concrete example in Fig. 7, with the constraints extracted by DocTer in Fig.7(a), the constraints generated by DEEPLANCET in Fig.7(b), and a detailed tabular comparison highlighting the key differences between these two sets of constraints in Fig.7(c). We conduct this comparison using the `torch.nn.functional.softshrink` function.

From the comparison results, we observe that DEEPLANCET eliminates extraneous information unrelated to testing and systematically converts irregular natural language descriptions of value ranges into a normalized, structured format. Notably, value range constraints are particularly important, as they directly determine input validity and ensure that test cases exercise core functional logic rather than being prematurely rejected.

This advantage is further evidenced by the results in Table II, where for the `torch.nn.functional.softshrink` function (Case 7), all test cases generated using constraints synthesized by DEEPLANCET successfully pass the validity checks, achieving a 100% validity rate. In contrast, DocTer attains only 54.9%, reflecting a 45.1% improvement with our approach.

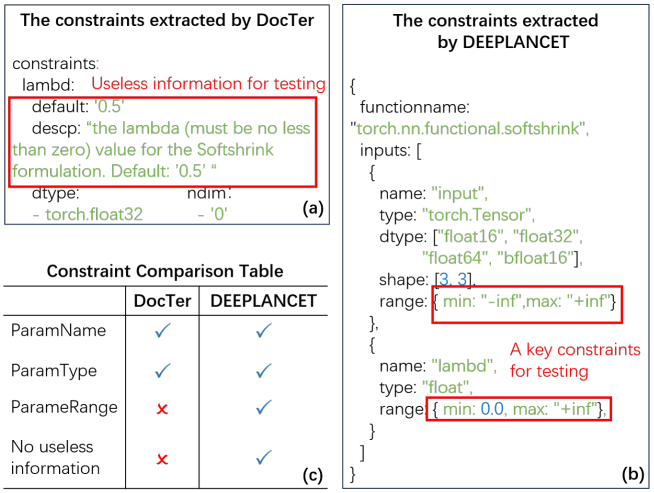

Fig. 7: Comparison of constraints extracted by DEEPLANCET and DocTer.

D. RQ3: Ablation Study

To answer RQ3 by investigating the contribution of DEEPLANCET's constraints fixing and validation components to the overall effectiveness, we conduct an ablation study by comparing the effectiveness of test cases generated with and without these components, thereby validating the usefulness of our technique. Specifically, we refer to the system without the constraints fixing and validation components as DEEPLANCET$^-$. For this experiment, we reuse the same experiment setting as in RQ2 to generate 1,000 test cases for each function and calculate the validity proportions for DEEPLANCET and DEEPLANCET$^-$, respectively.

We present the experiment results in the fourth column **DEEPLANCET$^-$** of Table. II. Without constraint fix, DEEPLANCET$^-$ generates 8,322 valid test cases giving a validity proportion of 83.2%, which are significantly less than those of DEEPLANCET. We conduct a manual inspection of the root causes and reveals that this degradation is due to the less precise constraints generated by DEEPLANCET$^-$. For example, in the function `torch.as_strided`, the valid range for its `stride` parameter should be $(0, +\infty)$ according to its documentation. Without constraint fix, the LLMs incorrectly infer this range as $(-\infty, +\infty)$, leading to invalid test cases.

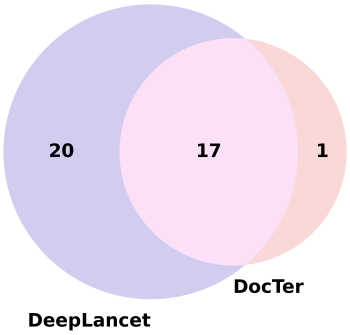

Fig. 8: Comparative analysis of bugs detection capability of DEEPLANCET against DocTer.

E. RQ4: Comparative Study

To answer the RQ4 by further assessing the technical advantages of DEEPLANCET, we compare it with the state-of-the-art deep learning library testing tool, DocTer [16], which also extracts input constraints from documentation to facilitate testing but without LLM-assisted constraint generation employed in this study.

We demonstrate the superiority of DEEPLANCET by conducting an end-to-end comparison of the number of bugs detected by generating an equal number of test cases. To be fair, we follow DocTer's evaluation setup to generate 2,000 test cases for each deep learning function by: 1,000 valid test cases that conform to the extracted constraints, and 1,000 invalid test cases that deliberately violate them.

We present the experimental results in Fig. 8. Among the 18 known bugs in the PyTorch, as listed by DocTer, DEEPLANCET successfully rediscovered 17. The only missing bug case involves the `torch.cuda.comm.scatter` function, which DEEPLANCET fails to test due to its current lack of support for the `torch.cuda.Stream` data type.

Furthermore, our approach demonstrates obvious advantages over DocTer in multiple aspects. First, by leveraging LLMs for constraint extraction, DEEPLANCET generates more precise and comprehensive input constraints, including detailed data types and value ranges, which DocTer often misses or extracts incompletely.

Second, DEEPLANCET's capability to convert informal natural language descriptions into structured annotations enables more effective and targeted test case generation. Moreover, DEEPLANCET is capable of handling documentation written in diverse styles and formats, which demonstrates its strong adaptability and scalability when applied to different deep learning libraries or evolving function specifications.

Third, DEEPLANCET incorporates a differential testing mechanism that systematically compares the execution results across heterogeneous backends including CPUs and GPUs detecting subtle inconsistencies that might otherwise remain unnoticed, thereby uncovering a broader range of hidden bugs and enhancing overall testing effectiveness.

VI. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work.

Generalizability. DEEPLANCET provides a general deep learning library constraint extraction techniques and introduces a unified annotation design for constraint representation to test deep learning libraries. This design allows easy extension to other frameworks. In this work, we primarily focus on evaluating PyTorch, however, our design (see Fig. 2) does not tie to PyTorch but is general. In our future work, we plan to employ DEEPLANCET to assess other deep learning libraries.

Complex constraints. DEEPLANCET currently supports constraints that manifest in the documentation. While current production deep learning libraries all have mature documentation to facilitate constraint extraction, they do miss some complex constraints involving indirect dependencies on the parameters. Nevertheless, these cases are rare, representing only 4.8% of parameters in our sample.. Therefore, the impact of this limitation on the overall effectiveness of DEEPLANCET is minimal and thus acceptable.

Incomplete Documentation. Although documentation-based constraint extraction generally provides sufficient guidance for generating high-quality test cases, occasional incompleteness or minor inaccuracies in documentation may occur. Fortunately, such cases are relatively uncommon and have limited impact on the overall effectiveness of DEEPLANCET. In future work, we think integrating source code level constraint extraction could supplement documentation, mitigating rare gaps and broadening DEEPLANCET's applicability.

VII. RELATED WORK

Related work can be classified into two categories: model-level testing and API-level testing.

Model-level Testing. Many existing approaches test deep learning libraries by generating new models or mutating existing ones.. CRADLE [31] is among the earliest tools to detect and localize bugs using open-source models. Luo et al. [32] proposes a graph-based fuzz testing approach that introduces six distinct mutation strategies by systematically exploring combinations of model architectures, hyperparameters, and input data. Although model-level testing has demonstrated promising results, it has limitations [33], particularly in achieving comprehensive coverage across all API functions. To address this, our work focuses on generating effective, fine-grained test cases for individual API functions.

API-level testing. API-level testing focuses on individual functions rather than complex inter-function relationships, enabling more flexible, fine-grained testing and effectively exposing potential errors. DocTer [16] is the first approach that automatically synthesizes constraints from documentation, but rule-based extraction is hard to scale and often misses information. ACETest [12] extracts constraints via symbolic execution, which suffers from path explosion when analyzing complex code. DeepConstr [17] derives constraints from error messages, but this fails if the message does not reflect the root

cause. In contrast, we leverage LLMs to extract input validation constraints from documentation, providing an effective approach for testing deep learning libraries.

VIII. Conclusion

In this work, we propose an effective approach for detecting bugs in deep learning libraries. Our method generates input test cases by extracting API input constraints from documentation using LLMs, enabling more effective testing of API functions. We first extract input attribute constraints from documentation, then generate test files using these constraints and historical error information, and finally run the tests to detect bugs in the deep learning libraries. We implement a software prototype called DeepLancet and conduct experiments to evaluate its effectiveness, usefulness, and overhead. DeepLancet identifies 20 bugs in PyTorch, demonstrating that the constraints extracted from documentation with LLMs can significantly enhance the effectiveness of fuzz testing.

References

[1] B. J. Erickson, P. Korfiatis, Z. Akkus, T. Kline, and K. Philbrick, "Toolkits and libraries for deep learning," *Journal of Digital Imaging*, vol. 30, no. 4, pp. 400–405, Aug. 2017.

[2] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. Sacramento, CA, USA: IEEE, Sep. 2016, pp. 1–10.

[3] S. Liu, S. Liu, W. Cai, S. Pujol, R. Kikinis, and D. Feng, "Early diagnosis of alzheimer's disease with deep learning," in *2014 IEEE 11th International Symposium on Biomedical Imaging (ISBI)*. Beijing, China: IEEE, Apr. 2014, pp. 1015–1018.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*. Santiago, Chile: IEEE, Dec. 2015, pp. 2722–2730.

[5] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, Jul. 2021.

[6] X. Zhang, W. Jiang, C. Shen, Q. Li, Q. Wang, C. Lin, and X. Guan, "Deep learning library testing: Definition, methods and challenges," *ACM Computing Surveys*, vol. 57, no. 7, pp. 1–37, Jul. 2025.

[7] *Enabling Pedestrian Safety Using Computer Vision Techniques: A Case Study of the 2018 Uber Inc. Self-Driving Car Crash*. Cham: Springer International Publishing, 2020, pp. 261–279.

[8] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 Fourth International Conference on Multimedia Information Networking and Security*. Nanjing, China: IEEE, Nov. 2012, pp. 152–156.

[9] J. Ji, W. Kong, J. Tian, T. Gu, Y. Nie, and X. Kuang, "Survey on fuzzing techniques in deep learning libraries," in *2023 8th International Conference on Data Science in Cyberspace (DSC)*. Hefei, China: IEEE, Aug. 2023, pp. 461–467.

[10] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, Dec. 2018.

[11] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, "A systematic review of fuzzing," *Soft Computing*, vol. 28, no. 6, pp. 5493–5522, Mar. 2024.

[12] J. Shi, Y. Xiao, Y. Li, Y. Li, D. Yu, C. Yu, H. Su, Y. Chen, and W. Huo, "Acetest: Automated constraint extraction for testing deep learning operators," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023, pp. 690–702.

[13] Y. Deng, C. Yang, A. Wei, and L. Zhang, "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*. Singapore Singapore: ACM, Nov. 2022, pp. 44–56.

[14] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen, "Predoo: Precision testing of deep learning operators," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Denmark: ACM, Jul. 2021, pp. 400–412.

[15] M. Rezaalipour and C. A. Furia, "annotest: An annotation-based test generation tool for neural network programs," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bogotá, Colombia: IEEE, Oct. 2023, pp. 574–579.

[16] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "Docter: Documentation-guided fuzzing for testing deep learning api functions," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual South Korea: ACM, Jul. 2022, pp. 176–188.

[17] G. Go, C. Zhou, Q. Zhang, X. Zou, H. Shi, and Y. Jiang, "Towards more complete constraints for deep learning library testing via complementary set guided refinement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vienna Austria: ACM, Sep. 2024, pp. 1338–1350.

[18] M. Li, J. Cao, Y. Tian, T. O. Li, M. Wen, and S.-C. Cheung, "Comet: Coverage-guided model generation for deep learning library testing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–34, Sep. 2023.

[19] Y. Wu, H. Iso, P. Pezeshkpour, N. Bhutani, and E. Hruschka, "Less is more for long document summary evaluation by llms," 2023.

[20] K. Fisher, D. Walker, K. Q. Zhu, and P. White, "From dirt to shovels: Fully automatic tool generation from ad hoc data," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 421–434, Jan. 2008.

[21] S. M. T. I. Tonmoy, S. M. M. Zaman, V. Jain, A. Rani, V. Rawte, A. Chadha, and A. Das, "A comprehensive survey of hallucination mitigation techniques in large language models," 2024.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[23] TensorFlow Developers, "Tensorflow," Zenodo, May 2021.

[24] R. K. Das and M. S. U. Islam, "Application of artificial intelligence and machine learning in libraries: A systematic review," 2021.

[25] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Virtual Canada: ACM, Jun. 2021, pp. 883–898.

[26] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837.

[27] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," Mar. 2024.

[28] D. R. MacIver, Z. Hatfield-Dodds, and many other contributors, "Hypothesis: A new approach to property-based testing," Nov. 2019.

[29] M. Li, D. Li, J. Liu, J. Cao, Y. Tian, and S.-C. Cheung, "Enhancing differential testing with llms for testing deep learning libraries," 2024.

[30] J. Liu, J. Lin, F. Ruffy, T. Cheng, J. Li, A. Panda, and L. Zhang, "Asplos2023 artifact for "nnsmith: Generating diverse and valid test cases for deep learning compilers"," Oct. 2022.

[31] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 1027–1038.

[32] W. Luo, D. Chai, X. Ruan, J. Wang, C. Fang, and Z. Chen, "Graph-based fuzz testing for deep learning inference engines," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, May 2021, pp. 288–299.

[33] J. Li, S. Li, J. Wu, L. Luo, Y. Bai, and H. Yu, "Mmos: Multi-staged mutation operator scheduling for deep learning library testing," in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. Rio de Janeiro, Brazil: IEEE, Dec. 2022, pp. 6103–6108.