Large Language Models Can Connect the Dots: Exploring Model Optimization Bugs with Domain Knowledge-Aware Prompts

Hao Guan*
University of Queensland
Brisbane, Australia
Southern University of Science and Technology
Shenzhen, China
hao.guan@uq.edu.au

Guangdong Bai†
University of Queensland
Brisbane, Australia
g.bai@uq.edu.au

Yepang Liu†‡
Southern University of Science and Technology
Shenzhen, China
liuyp1@sustech.edu.cn

Abstract

Model optimization, such as pruning and quantization, has become the *de facto* pre-deployment phase when deploying deep learning (DL) models on resource-constrained platforms. However, the complexity of DL models often leads to non-trivial bugs in model optimizers, known as *model optimization bugs* (MOBs). These MOBs are characterized by involving complex data types and layer structures inherent to DL models, causing significant hurdles in detecting them through traditional static analysis and dynamic testing techniques. In this work, we leverage Large Language Models (LLMs) with prompting techniques to generate test cases for MOB detection. We explore how LLMs can draw an understanding of the MOB domain from scattered bug instances and generalize to detect new ones, a paradigm we term as *concentration and diffusion*. We extract MOB domain knowledge from the artifacts of known MOBs, such as their issue reports and fixes, and design knowledge-aware prompts to guide LLMs in generating effective test cases. The domain knowledge of code structure and error description provides precise in-depth depictions of the problem domain, i.e., the *concentration*, and heuristic directions to generate innovative test cases, i.e., the *diffusion*. Our approach is implemented as a tool named YanHui and benchmarked against existing few-shot LLM-based fuzzing techniques. Test cases generated by YanHui demonstrate enhanced capability to find relevant API and data combinations for exposing MOBs, leading to an 11.4% increase in generating syntactically valid code and a 22.3% increase in generating on-target code specific to model optimization. YanHui detects 17 MOBs, and among them, five are deep MOBs that are difficult to reveal without our prompting technique.

*Hao Guan is under the UQ-SUSTech Joint PhD Program.
†The corresponding authors are Yepang Liu and Guangdong Bai.
‡Yepang Liu is affiliated with the Research Institute of Trustworthy Autonomous Systems and Department of Computer Science and Engineering at Southern University of Science and Technology.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680383

CCS Concepts

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

Keywords

Model Optimization, Library Testing, Large Language Model

ACM Reference Format:
Hao Guan, Guangdong Bai, and Yepang Liu. 2024. Large Language Models Can Connect the Dots: Exploring Model Optimization Bugs with Domain Knowledge-Aware Prompts. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680383

1 Introduction

Deep learning models have been extensively applied in various domains, such as autonomous driving [26] and medical diagnosis [2, 37], owing to their remarkable capabilities in classification and generation. To tackle complex tasks and achieve a high level of accuracy, these models are often trained with a large number of parameters and layers. For example, a popular large language model, LLaMA [1] by Meta, contains up to 32 layers and 65 billion parameters. This practice, however, hinders the deployment of models on resource-constrained platforms such as mobile devices and Internet-of-Things (IoT) nodes. As a result, optimization techniques, typically pruning [18, 48, 61] and quantization [3, 25], have to be engaged to reduce the model size and complexity to facilitate model deployment. These techniques have been incorporated by popular DL frameworks, such as TensorFlow and PyTorch, as a module called *model optimizer*.

In contrast to model training and inference stages, where models are operated on in a holistic manner, model optimization entails direct alterations to the model parameters, necessitating careful consideration of the intricate connections among these parameters. This complexity is particularly noteworthy when dealing with realistic models featuring complex architecture or uncommon structures. Consequently, model optimization can become error-prone, giving rise to the *model optimization bugs* (MOBs) [16]. MOBs can significantly undermine model reliability or even hinder the deployment of deep learning applications on end devices, leading to adverse effects on user experience or financial loss [7, 15, 22].

Detecting MOBs is a complex task due to various obstacles stemming from the characteristics of model optmizers [9, 16]. The use of hybrid programming languages and diverse hardware/platform

orientation present significant challenges for approaches based on traditional static analysis [30, 31]. Dynamic testing shows promise as a widely used method for bug detection, but the unique nature of models, i.e., the input to model optimizers, impairs the test case generation. Indeed, existing approaches for testing DL libraries primarily focus on either fuzzing DL models [17, 19, 49, 57, 59] or evaluating the usability of library APIs [11, 13, 27, 42]. The research problem of *addressing the involvement of deep learning models as input and complex APIs as configuration* in detecting MOBs remains largely underexplored.

Our work. In this work, we leverage Large Language Models (LLMs) to enhance dynamic MOB detection. LLMs have been shown effective in automatically testing DL libraries by recent studies [9, 10, 53], owing to their exceptional capacity for understanding and memorizing the correct syntax and semantics of the code bases they have seen in their massive training data. Our focus is to refine the capabilities of LLMs, which are initially trained on massive general texts, to tackle domain-specific tasks effectively. To this end, two primary obstacles need to be addressed, as elaborated below.

Obstacle #1. Knowledge gap in model optimization. One significant obstacle in leveraging LLMs lies in their limited knowledge of specific domains, such as model optimization in our work. Model optimization is a relatively newer and smaller component within DL frameworks compared with the mature modules of model training and inference. To illustrate, PyTorch and TensorFlow, the most popular DL frameworks, only introduced their model optimizers in 2020, with PyTorch v1.3 and TensorFlow v2. Within their code repositories, model optimizers are notably sparse in the distribution of code (3.5% in PyTorch and 1.8% in TensorFlow) and test cases (2.2% in PyTorch and 4.0% in TensorFlow). Even though LLMs may have these DL frameworks in their training datasets, the code of model optimizers is a minor portion. As a result, the test code generated by LLMs tends to be biased towards the APIs of model training or inference, which is not our desired target for testing MOBs. Furthermore, due to the limited exposure to model optimization concepts, the generated code may not be accurate. This can lead to misuse of APIs or compatibility issues during testing, reducing the overall efficiency of the testing process.

Obstacle #2. Lack of guidance in test space exploration. Testing with LLM-generated code is essentially a fuzzing process [10], where efficient exploration of the test space poses a common challenge. A valid piece of test code typically consists of several elements, such as model definition, model input data, and optimization API invocation. The intricate nature of these elements and their various combinations can lead to the explosion of the search space. Without proper guidance, the generated test code may be gradually trapped in repetitive or irrelevant patterns [23]. Addressing this obstacle requires LLMs to enhance the diversity of their generated test code, ensuring an expansive exploration of the test space.

We propose a new prompting paradigm termed *concentration and diffusion* to overcome these two obstacles. The *concentration* is designed to transition the capabilities of LLMs from a broad scope to the specific domain of MOBs (for ***Obstacle #1***). Initially, we incorporate domain knowledge distilled from the documentation of model optimizers into the prompts given to LLMs, providing LLMs with a precise and in-depth depiction of model optimization. Within the domain of model optimization, we further design a knowledge concentration strategy that considers the taxonomy of MOBs [16]. This strategy enables flexible prompting by controlling the degree of MOB knowledge concentration.

The *diffusion* boosts innovation and diversity within the MOB domain (for ***Obstacle #2***). This process applies code mutation on historical bugs, a strategy that has been found effective for detecting additional bugs [10, 60]. We enumerate effective and actionable mutation rules grounded in the domain knowledge of MOBs [16]. These mutation rules are embedded as instructions in the prompts, facilitating LLMs to make meaningful modifications and produce innovative test cases, rather than over-imitating existing examples.

We implement our approach as YanHui[1], and compare it with the state-of-the-art LLM-based fuzzing technique FuzzGPT [10]. Our approach shows the significantly enhanced quality of the generated test code. With our prompts, there is an 11.4% increase in generating syntactically valid code and a 22.3% increase in generating relevant code on model optimization. Within the same iterations of test code generation as FuzzGPT, YanHui can detect 17 more bugs. After reviewing the bugs, we find that our approach can detect not only the common model API bugs, but also exclusive types of MOBs involving language features and illogical code, which cannot be found by the baseline.

Contributions. This work makes the following main contributions.

- **A novel prompting paradigm for domain-specific tasks**. Our work underscores the importance of domain knowledge when using LLMs for complex tasks such as generating code for MOB detection. We propose a paradigm of concentration and diffusion to direct the focus of LLMs into specific domains while retaining their generation capabilities within those domains.
- **A step forward in MOB detection**. We develop YanHui, a practical framework that implements the concentration and diffusion prompting paradigm. YanHui represents a valuable step toward MOB detection, an essential but underexplored problem. It is shown to be cost-effective for addressing such a complex problem.
- **A practical study on real-world model optimizers and previously unknown MOBs detected**. We apply YanHui on model optimizers from prominent DL libraries. The results demonstrate its superiority over state-of-the-art LLM-based DL testers. Notably, YanHui identifies 17 MOBs, including five deep MOBs that are difficult to detect.

Availability and Ethical Consideration. We open-source YanHui and the associated MOB artifacts [58], to facilitate further research of MOB detection and the exploration of domain knowledge-aware prompts in other scopes. All identified MOBs have been responsibly disclosed to the respective developers. At the time of submitting the camera-ready version of this paper, eleven MOBs have been confirmed, while discussions are ongoing for the remaining six.

Paper Organization. The rest of this paper is organized as follows. Section 2 reviews the background of model optimization bugs and large language models. Next, Section 3 presents our method with domain knowledge-aware prompts applied in this study. Then in Section 4, we show the detailed evaluation results, and compare

[1]Yan Hui is the favorite disciple of Confucius, earning praise from Confucius that "*when he learns one thing, he gets to understand ten more things*".

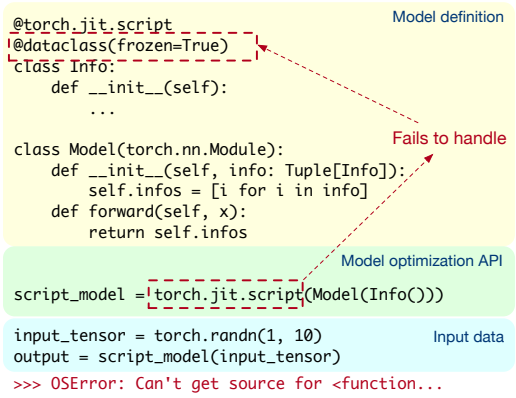

Figure 1: An Example of Model Optimization Bug

our method with the state of the art. In Section 5, we discuss the limitation of this study. Finally, Section 6 surveys related research and Section 7 concludes our work.

2 Preliminaries

In this section, we present a running example of a MOB and the background knowledge to facilitate the understanding of our method.

2.1 Model Optimization Bugs (MOBs)

Contemporary machine learning models, particularly deep neural networks, are designed for tackling complex real-world problems. They typically contain a vast number of parameters and demand significant storage and computational power, exceeding the capacity of resource-constrained devices. Model optimization techniques are crucial for tailoring these pre-trained models before deployment. There are two typical types of model optimization techniques, i.e., *Pruning* [18, 61] that identifies insignificant neural network layers and zeros out the weights to increase the sparsity of models, and *Quantization* [20] that converts model parameters into values of lower precision, such as 16-bit floats to 8-bit integers. Both PyTorch [14] and TensorFlow [21] provide a similar set of APIs in their optimizers.

A typical model optimization program contains three major components.

a) Model definition. The model definition represents the core information of the DL model, including the type of layers, structure and training functions. It determines how the input tensor will be converted to the output.

b) Optimization API invocation. The invocation of optimization APIs will apply specific operations with options and parameters on the defined model.

c) Input data. This component defines the input tensor that fits the model requirements. During the execution of the inputs, the operations of the model are recorded. Optimization may rely on the information of the data and operations.

Figure 1 shows a typical program of model optimization for PyTorch. It defines a model by subclassing `torch.nn.Module`. The calculation in `forward` involves a custom type of data called `Info`,

```
# API: foo
# Title: fails in torch.internal_
(code...)
# API: bar
# Bug description: model parsing failure
(code...)
# API: target_api
# Title: <infill>
```

Figure 2: An Example of Chain-of-Thought Prompts that Contain Steps and Examples

which is optimized with `torch.jit.script`. Finally, the model after the script accepts a tensor as input.

MOBs occur when model optimizers modify pre-trained models, including data types, layers operations and metadata. For instance, the original model in Figure 1 can work well, but an `OSError` will occur when applying optimization on the model because the framework fails to handle the `dataclass` decorator.

2.2 LLMs and Chain-of-Thought Prompting

Various powerful Large Language Models (LLMs) [12, 32, 36, 43–45] have been trained with a large amount of text to accomplish natural language processing, such as text classification, summary and generation. They have demonstrated strong capabilities on these tasks and are evolving rapidly by accepting extensive data, with billions of parameters. Some variants of LLMs have been introduced to accomplish code-specific tasks. Typical code-specific LLMs include CodeT5 [47], Codex [6], and CodeLlama [39]. The variant models can be produced by adding code as training data, or fine-tuning the original model with code-related contexts.

LLMs accomplish tasks provided in the form of prompts [38, 52], in a flexible manner that can avoid model retraining. To enhance the reasoning capability of LLMs, various prompting techniques have been proposed. The chain-of-thought [51] represents one of the prominent prompting techniques. Its core idea is to describe a complex task with small steps, and prompts are separated into several intermediate segments that include the reasoning process. Such prompts have been demonstrated to be processed more accurately by LLMs [8, 10, 24, 46, 56]. For example, the segments may include the instructions on how to process the previous results, or what formats or restrictions should be satisfied when generating the proceeding contents. LLMs can follow this pattern and produce similar instructions or constraints during the generation. As a result, the final outputs are more likely to meet the expectations.

In general, chain-of-thought prompting is applied with few-shot examples to accomplish tasks of code generation. For instance, FuzzGPT [10] provides the information of API, bug description, and finally code snippet as the steps of thought. The core generation process is demonstrated in Figure 2. The prompts first include several examples with the format of API, bug description, and code snippet. Then the targeted API is mentioned, and LLMs will infill the remaining part of the bug description and code snippet. LLMs will complete the remaining parts.

Given the efficacy of the chain-of-thought prompting, YanHui applies it for the construction of prompts. YanHui incorporates domain knowledge of model optimization and MOBs into the prompts, to provide reasoning information based on the characteristics of

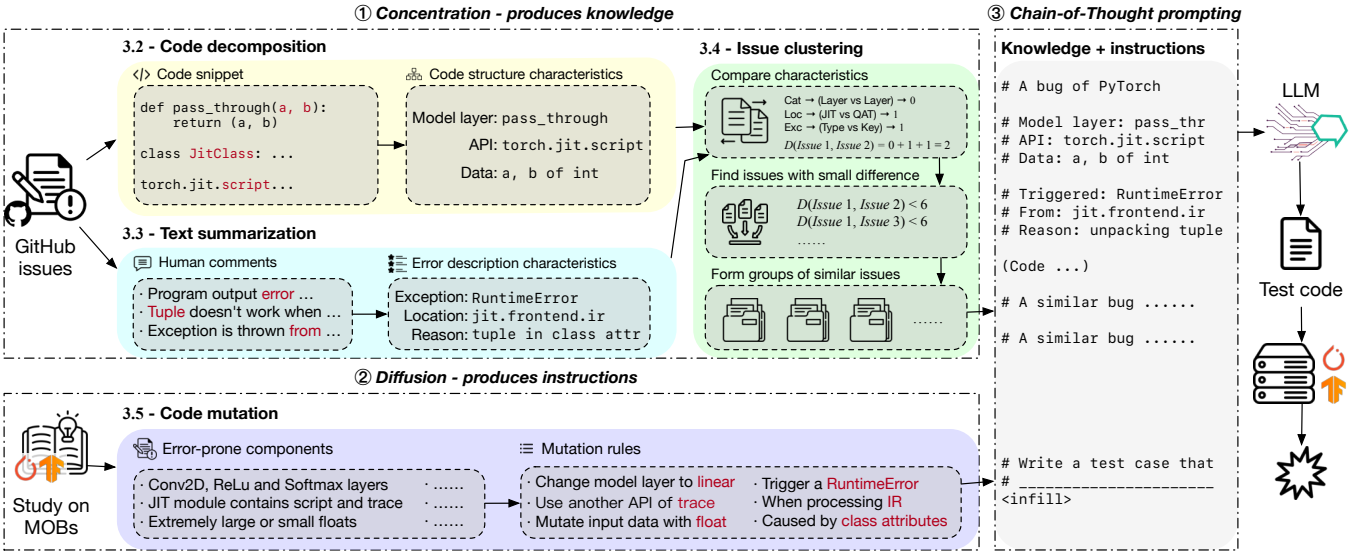

Figure 3: Overview of YanHui

the buggy code snippets, including API, exception, data type, and error reason. This is further detailed in Section 3.

3 Methodology

YanHui takes advantage of domain knowledge to generate effective test cases for detecting MOBs. In this section, we detail its approach. We start with an overview of its chain-of-thought prompting with the concentration and diffusion paradigm (Section 3.1), and then present each of its internal components (Section 3.2 to 3.5).

3.1 Overview

Figure 3 demonstrates the overall workflow of YanHui, using our example (Figure 1) as an illustration. YanHui takes the artifacts of historical MOBs as input and applies the concentration and diffusion paradigm to produce prompts that mainly consist of *knowledge* and *instructions*. The knowledge distilled by concentration (the box ① in Figure 3) provides LLMs with accurate and concise domain knowledge to facilitate LLMs in generating valid and relevant code on model optimization. The instructions constructed by diffusion (the box ② of Figure 3) provide LLMs with an effective and specific guide to make LLMs generate tests targeting error-prone components. Both the knowledge and instructions are formatted as specific steps so that LLMs can generate effective test code after chain-of-thought prompting (the box ③ of Figure 3). The generated code is then tested with model optimizers for MOB detection.

Concentration. In the concentration process, YanHui targets to group similar MOB instances so as to enhance the specificity of the distilled knowledge and reduce irrelevant contexts. To measure this similarity, we use six features to characterize a MOB. They are derived based on the optimization program paradigm (see Section 2.1) and the MOB taxonomy from an existing study [16]. They are organized with the following two aspects.

- **Code structure**. Code can be a summary of the functionality of the machine learning program, and can facilitate LLMs in understanding code. The features in this aspect are the model optimization components discussed in Section 2.1, including
  - *model definition*, which describes what layers and operations are included,
  - *optimization API invocation*, which represents the optimization technique that modifies the model, and
  - *input data*, which specifies the type and value passed to the model.
- **Error description**. Error description contains the textual data in relation to MOBs, and it has been shown effective in analyzing the root cause of MOBs [16]. The features that are included by error description include
  - *exception type*, which is the type of error that arises when running the code,
  - *error location*, which is the location of the buggy code in terms of the file and line inside optimizers, and
  - *error reason*, which contains developers' or programmers' explanation of the triggering condition and the root cause of the bug.

YanHui identifies these features from each bug issue of historical MOBs. The artifacts are processed using different strategies due to the various formats. For structured code snippets, YanHui applies decomposition (Section 3.2), and for unstructured natural language, YanHui applies summarization (Section 3.3). With the identified features, historical MOBs can be clustered to control YanHui's concentration degree. The similarity among the issues is measured with the difference in MOB category, API and data type, which are detailed in Section 3.4.

Diffusion. As has been shown in recent studies [41, 54], code generated by LLMs can easily fall into the trap of repeating the given samples and their patterns without proper guidance. Therefore, in the diffusion process, YanHui provides explicit instructions so that LLMs can mutate the code of given examples in meaningful ways and generate innovative and diversified code that involves new

APIs and data, for example, the code containing uncommon model types (e.g., `@dataclass` in Figure 1) and optimization APIs (e.g., `torch.jit.script`).

In addition, Guan et al. [16] have revealed that the error-prone components and poorly tested modules of model optimizers are highly likely to lead to MOBs. For instance, an extremely large or small floating-point number has a high risk of error. A model may cause a crash if it contains a complex data structure instead of a simple scalar. Therefore, we design the mutation rules based on these heuristics to instruct LLMs to diffuse towards error-prone modules inside model optimizers (Section 3.5).

3.2 Code Decomposition

The code snippets are usually well structured in bug issues, making them easily recognizable. However, there is a lack of techniques that split a code snippet into semantical fine-grained steps that chain-of-thought prompting demands. To address this, we develop a decomposition method. We use the paradigm of model optimization programs presented in Section 2.1 and Figure 1. YanHui matches the code with API keywords and syntactic structures to retrieve the program components automatically, as briefed below.

a) Model definition. A model can be defined with either class API or functional API [14, 21]. In PyTorch and TensorFlow, the model can be recognized from the following patterns.

- PyTorch
  - Class models: inherit from `torch.nn.Module` and contain the method of `forward`.
  - Functional models: use the API from `torch.nn.functional`.
- TensorFlow
  - Class models: inherit from `tf.keras.Model` and define `call` method.
  - Functional models: use the API from `tf.keras.layers`.

Inside the model definition statements, YanHui matches the layer APIs from the library documentation. Finally, it produces a formatted model description, i.e., the type and parameters of input, processing layers, and output.

b) Input data. PyTorch accepts `torch.Tensor` as the input, whereas TensorFlow models accept `tf.Tensor`. The input data can be identified by finding the tensor object that is passed as the argument to the model.

c) Model optimization API. Both PyTorch and TensorFlow put the optimization APIs into separated modules and name the modules after the optimization techniques [21, 34]. YanHui identifies the APIs that are provided by the following library modules:

- PyTorch
  - `torch.quantization`
  - `torch.nn.utils.prune`
  - `torch.jit`
- TensorFlow
  - `tensorflow_model_optimization`
  - `tfmot`

With these patterns, the three components of the model optimization program are extracted, and the API used in each part is also identified. The formatted parts are the core of model optimization, and can describe the functionalities of the code clearly, which would facilitate LLMs to parse the example code. They are also

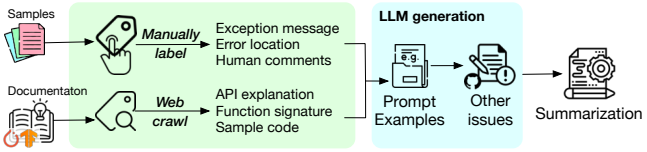

Figure 4: The Process of Text Summarization

the templates of code to generate new test cases from historical bugs (to be discussed soon in Section 3.5).

MOB-specific code pool of YanHui. YanHui maintains a pool of MOB-specific code collected from the GitHub repositories of PyTorch and TensorFlow, the most popular DL frameworks. We follow the filtering queries of Guan et al. [16] to select the issues or pull requests that are in the scope of model optimization. From the collected issues or pull requests, we fetch runnable code from the markdown code blocks. Overall, YanHui has collected 392 code snippets.

3.3 Text Summarization

Besides code snippets, the issues and pull requests contain natural language texts, including titles, developer comments, commit messages, and other metadata. YanHui mines the three types of error features from these texts, including exception type, error location and error reason (Section 3.1), and uses them as heuristic steps that direct LLMs to comprehend the MOB and produce targeted test code [10]. Figure 4 presents the overall process of the text summarization.

It is challenging to extract such features automatically because the error description can be expressed in various ways. This requires precise comprehension of natural language and sufficient background knowledge of model optimization. We thus apply a semi-automated process assisted by LLMs. We first sample 20 issues, four from each of the following five MOB categories, i.e., wrong type, unexpected shapes, missing supporting data types, missing supporting layer operations, and metadata conversion errors. These take up 5.1% of the total MOB issues we have collected. Then, we apply the summarization with the following steps.

a) Exception and location from error messages. We first manually review the error messages from the exceptions raised by Python, which usually contain some type of `RuntimeError`, file and line number, or `Traceback` data. Rich information about the MOB can be found in the error message. For instance, the error reason is placed after the exception name. With the file and line number, the module and function can be retrieved. Figure 5 demonstrates an example of extracted features.

In addition to the error messages collected from bug issues, we execute the code snippets to enrich the error information. Although the error messages are generally structured, they may include irrelevant exceptions and backtraces. In this process, manual effort is involved to select relevant errors and extract concise information. Overall, we have collected the error features of 392 historical MOBs.

b) Error reasons from comments From the natural language texts in the issues, we extract the errors and root causes. We search keywords representing symptoms such as "*crash*" and "*error*", root causes such as "*typing*" and "*missing supporting*", Python exceptions

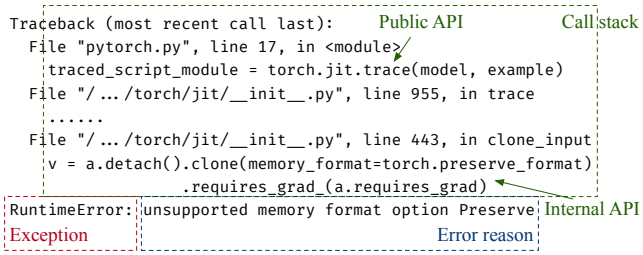

Figure 5: Features Extracted from Error Messages

```
1  # Title: REGR: Accessing dict in JITed code in 1.11
2  # Exception: KeyError: meta_y_hat
3  # Trace: File "test.py", line 36, in forward
4  # x, meta = self.activation(x, meta)
5  # Version: PyTorch version: 1.11.0+cu102
6  # Labels: oncall:jit
7  # PR Title: [JIT] fix common_expression_hoisting
8  (code...)
9  # Exception: KeyError on model with custom activation
10 # Location: TorchScript in JIT module
11 # Reason: accessing dict the property of Dict using
           torch.jit.script
```

Figure 6: Sample Labeling of Pytorch#74056

such as `TypeError` and `AssertionError`, and the APIs listed in PyTorch documentation [14]. In addition, DL framework developers and maintainers usually mark issues with labels after they triage the problems. Therefore, we can extract the location information if the label mentions the module inside the optimizers. For the resolved issues, one or more pull requests can be cross-referenced to the issue. If a pull request is referenced by the issue, we extract the error reasons or reproducing situations by rephrasing the patch title and the contents after "*fix*" in the commit message.

Then, LLMs are involved in generating the summarization for the rest of the issues in our dataset. As the example shown in Figure 6, the comments above the code (title, exception, error reason, API call stack, version, labels, and pull requests) are the automatically extracted features. The comments under the code (API and error description) are summarized manually based on the characterization of MOBs. The labeled error description reveals the root cause of this bug, which cannot be directly found from the code and trace information. Instead, it should be mined from the title, pull request and the deeper implementation of code.

With a small set of human-labeled examples, we construct a context for labeling the code snippets automatically. An unlabeled issue information and code snippet are placed after the examples. Then, the LLMs are prompted to complete the text with API and error descriptions. Specifically, for a collected issue like Figure 6, the contents are masked after Line 8, and we use LLMs to fill in the remaining contents. LLMs can follow the patterns of summarizing the API and error description from the provided information step by step, producing more accurate results.

As this contains both manual effort and LLM generation, we involve a cross-validation process to control the quality in Section 5.1.

3.4 Issue Clustering

We develop a measurement of the similarity between MOBs based on the categories of MOBs [16], and the structure of optimizers [21, 34]. Given two MOBs, we denote their features with a vector C. Let i be the issues, API, Typ represent the API and data type extracted from the code information, and Cat be the exceptions and error category of the MOB. The feature vector is defined as $C_i = \langle Cat, API_1, \cdots, API_n, Typ_1, \cdots, Typ_m \rangle$. The distance between two MOBs a and b is measured as $D_{a,b} = \sum |C_a - C_b|$, where the distance in each feature is defined with the following rules.

- **Category** ranges from the five root causes of MOB [16]. If the root causes are the same, the difference is 0, otherwise the difference is 1.
- **API** is the function called in the Python code, including the layers in the model and the optimization techniques. Each API has its module hierarchy inside the code base. For example, `script` and `trace` are both in `torch.jit`, whereas `quantization` is directly under `torch.quantization`. The distance is measured using the directory distance of the file that defines the API. For example, given two files `fake_quantize.py` and `quantized.py`, we compare the difference on the full paths of these two files, i.e., `torch/ao/quantization/experimental` and `torch/jit`. There are 3 levels of difference after `torch`, therefore the distance between the two APIs is 3.
- **Type** is the Python class of data or exception. The types have a hierarchy of subclass inheritance, such as `ArithmeticError` contains `ZeroDivisionError` and `Dtype` contains `bool` and `float`. For the same type, the distance is 0. If the types are different, the distance will be added by 1, and the types of their parent classes will be compared recursively. The comparison repeats until there is no parent for either type.

With a smaller distance threshold for the clusters, the code would be more centralized, but there would be fewer examples inside each cluster. After evaluating our dataset, we choose 6 as the distance threshold which can include 2-5 examples in each cluster. The quantity of examples is neither too few for chain-of-thought prompting nor so abundant that it exceeds the context size of LLMs. Overall, YanHui produces 198 clusters from 392 issues.

3.5 Mutation Rules

To guide the generation of testing code, we apply diffusion by providing instructions in the prompt. Each instruction specifies a mutation rule, which is associated with the knowledge summarized in the previous sections. These mutation rules are designed based on the error-prone or untested APIs, types and values, which are aligned with the root causes of MOBs [16].

1) Model definition. A deep learning model is usually constructed with a model class, which defines the layers and functions. The instruction explicitly asks the model to change the layer from the given sample code, for example, to replace the `Linear` layer with other layers supported by the tested optimizer. As a result, the generated code includes models with various layers so as to test the wrong type and metadata conversion errors in depth.

2) Optimization API. The instruction explicitly asks the model to use an alternative API to optimize the model, for example, to switch the conversion method from `script` to `trace`. Different

APIs can vary in configuration and operation, which can increase the combinations of the optimization process. The misalignment of model and API can commonly trigger MOBs of missing supporting layer operations [16].

3) Input data. Input data is used during the tracing process of deep learning models. The computational graph can be constructed from the tracing results. However, some data with special values or types may trigger edge cases, and the results cannot represent the original model. Mutating input data based on historical bugs, such as very large values or list formats, can help expose MOBs of unexpected shapes or missing supporting data types [16].

4) Triggered exception. The exception can usually clearly represent a bug. For example, `IndexError` can be caused by accessing the list elements, and `KeyError` can be caused by dict operations. By enumerating the exception types of Python, LLMs can be directed to explore the different components in the code and detect potential bugs after scanning the list, dictionary, class or other parts.

5) Bug location. The location in historical bugs can reveal a common failure during the model optimization process. Since the optimization may be implemented differently for each operation and platform, a historical missing supporting bug [16] may still exist in another similar part of the code base. Enumerating the place of mistakes may improve the probability of finding such potential problems.

6) Bug reason. In historical bugs, the information on the root cause can point out the direction to generate new test cases. The bug reason may mention some data type, API and language features. We do not directly mutate the reason content but ask LLMs to generate tests that may trigger problems in a similar way. The reason can make LLM generate code that reflects a deeper analysis of the bug.

An example of a complete prompt with both knowledge and instructions is shown in the box ③ of Figure 3. In the prompt, code from historical bugs is included, and the knowledgeable description is organized systematically. Following the examples, explicit instructions are given to guide the generation process. Then LLMs continue to complete the text and fulfill the mutation as expected, which will run for testing crash.

Among the 198 clusters, all of them are mutated in model definition and input data. 126 clusters are suitable for mutating the optimization API according to the documentation. Furthermore, 73 clusters show variations in exceptions or locations amenable to mutation. In total, mutating each factor once results in the generation of 595 test cases.

4 Evaluation

In this section, we evaluate the effectiveness of our prompting strategy. We analyze our approach by answering the following three main research questions.

- ***RQ1: Does YanHui outperform the state-of-the-art approach of LLM code generation in testing MOBs?*** This RQ aims to investigate the quality of the testing code generated with YanHui. In the comparison, we take the few-shot version of FuzzGPT [10] as the baseline.
- ***RQ2: Is the knowledge-aware prompting useful to find bugs in model optimization?*** This RQ focuses on the MOBs found with the knowledge-aware prompting technique, and reveals the advantages of using LLM generation for MOB detection.
- ***RQ3: Can YanHui generalize to different LLMs? How do the steps of concentration and diffusion contribute to the improvement of code generation?*** This RQ investigates the generalizability of YanHui's prompting paradigm and conducts an ablation study of each prompting component, which aims to understand how MOB domain knowledge affects YanHui's test code generation.

Table 1: LLMs evaluated in YanHui

Model	Context Size	Generation Speed	Cost
gpt-4-0125	128k	16.988 token/s [2]	$0.13/it
starcoder2-7b	16k	14.098 token/s	N/A
codellama-python-7b	16k	18.116 token/s	N/A

[2] GPT-4 can only be accessed via paid online API. The generation speed is measured by the average duration of network requests.

4.1 Experiment Design

Experiment Platform. The experiments are executed on a workstation with the following technical specifications.

- **OS:** Ubuntu 22.04.4 LTS x86_64
- **CPU:** AMD Ryzen Threadripper PRO 5965WX
- **GPU:** NVIDIA RTX A6000 × 2
- **Memory:** 256GB

Large Language Models. To investigate the generalization of YanHui's prompting paradigm, we have involved three popular LLMs for bug reasoning and code generation.

- GPT-4 [32]: an advanced iteration of LLM developed by OpenAI, which has demonstrated its exceptional performance in generating contextually appropriate texts in various domains such as programming and math [33].
- CodeLlama [39]: an emerging model in the field of automated code generation by Meta, which is a fine-tuned version of Llama2.
- StarCoder [28]: a sophisticated code generation model that incorporates enhanced attention mechanisms to improve its understanding and manipulation of complex coding structures.

The specifications of LLMs used in our evaluation are listed in Table 1.

4.2 RQ1: Effectiveness of YanHui's Prompts

Baseline approach. The baseline experiments are adapted from the few-shot prompting approach from the state-of-the-art FuzzGPT [10]. Since the artifacts of FuzzGPT are not available at the time of our evaluation, we reproduce the few-shot approach by ourselves following its paper, which is referred to as FuzzGPT-FS hereafter.

We randomly select the samples from our labeled code snippets to construct a prompt. The information of the code snippets given to the LLM includes only API and issue titles. For each generation, four examples (average number of examples in the clusters of YanHui) are provided to the LLM. The format of prompts follows the chain-of-thought prompts demonstrated in Figure 2.

Table 2: Generation Quality with Different Approaches

	FuzzGPT-FS				YanHui			
Model	**Outputs**	**Valid**	**Relevant**	**MOB**	**Outputs**	**Valid**	**Relevant**	**MOB**
gpt-4-0125	632	192 (30.4%)	78 (12.3%)	6	595	246 (41.3%, 10.9%↑)	214 (36.0%, **23.7%**↑)	14
starcoder2-7b	708	156 (22.0%)	73 (10.3%)	2	595	208 (35.0%, **13.0%**↑)	188 (31.6%, 21.3%↑)	9
codellama-py-7b	653	134 (20.5%)	52 (8.1%)	0	595	183 (30.8%, 10.3%↑)	179 (30.1%, 22.0%↑)	6

Prompts

```
# API: torch.jit.trace
# Title: String handling is not supported by JIT
(code...)

# API: torch.jit.script
# Title: Unknown type name 'a.Type'
(code...)

# API: <infill>
```

· Different API
· Varied errors

Generation

```
PyTorch Dataloader with shuffle=True

# the following fails with TypeError: `type' object
# is not iterable on dataloader.__iter__()

class MyDataset(torch.utils.data.dataset):
...
dataset = MyDataset(456789)
dataloader = torch.utils.data.DataLoader(dataset,
                                         batch_size=32,
                                         shuffle=True)
```

· Wrong code
· Irrelevant API

Figure 7: Invalid Code Generated by FuzzGPT-FS

Evaluation metrics. We define the following metrics to evaluate the performance of YanHui.

- ***Valid outputs***. The generated code is valid Python code with no syntax errors. We filter out invalid outputs by actually running generated code and excluding those cases that produce exceptions of `NameError`, `SyntaxError` and `ModuleNotFoundError`.
- ***Relevant outputs***. The valid code that is testing the model optimizers inside the deep learning frameworks. The generated code is decomposed with the steps introduced in Section 3.2. The relevant outputs are selected by checking whether the model and input are created, and whether the API of optimization is applied to the model.
- ***Real MOBs***. During the testing, we keep the test cases that produce runtime errors, as they represent particular failures during model optimization. We manually review the errors to find real bugs that conform to the root causes of MOBs listed in Section 3.3.

Results of effectiveness. To ensure the fairness of the comparison, we conduct the same iterations of generation (595) for the two approaches. In the prompts of FuzzGPT-FS, we include four examples for one iteration, which is the average number of examples of YanHui's prompts. Table 2 presents the quality of the test cases generated with different approaches. The valid outputs and real MOBs are highlighted in the table. There is an improvement of up to 13.0% on the valid outputs. The ratio of relevant outputs has a more significant improvement of up to 23.7%. YanHui can also find more real MOBs.

Prompts

```
# Here is a bug of PyTorch
# Model:       nn.Module
# API:         torch.jit.script
# Data:        "abcd"
# Triggered:   RuntimeError
# From:        torch.jit.frontend
# Reason:      desugar of str is not handled in JIT
(code …)

# And a similar bug of PyTorch
# Model:       nn.Module
# API:         torch.jit.script
# Data:        **{ "hello": None }
# Triggered:   RuntimeError
# From:        torch.jit.frontend
# Reason:      expanding dictionary is not handled
(code …)

......

# Write a similar test case of PyTorch that contains
# the unhandled part of models.
```

Similar API and code structure

Generation

```
# JIT does not support the **kwargs syntax
class MyModel(nn.Module):
    def __init__(self, a=1):
        ...

    def forward(self, x, **kwargs):
        return x + 1

m = MyModel()
scripted = torch.jit.script(m)
scripted(torch.tensor([2]))
```

Valid model and API use

Innovated mutation

Figure 8: Effective Code Generated by YanHui

Case study. We investigate the reason for the quality improvement of code generation by explaining two examples from the two approaches.

Figure 7 shows an invalid testing code generated by the baseline approach. After examining the context, we notice that the examples in the prompts are not centralized enough. The types and modules of APIs are in low similarity (tinted red), which provide inconsistent API usages to LLMs. As a result, the generation may be negatively affected by the examples and produce invalid code - subclassing `torch.utils.data.dataset` does not construct a valid deep learning model. Besides, the prompts include examples of varied errors (tinted blue). The similarity of error locations between "sanity check" and "unknown type name" is also low, which caused the generated code not relevant to testing MOBs - instead, it is about the dataset API `DataLoader`.

Table 3: Real MOBs Found by LLM Generated Code

Approach	API	Desugar	Illogical
FuzzGPT-FS	8	0	0
YanHui	12	**3**	**2**

In contrast, the code generated with centralized examples selected using the concentration strategy of YanHui can test optimizers effectively, which is shown in Figure 8. The examples are focusing on the same API (`torch.jit.script`) and location (`torch.jit.frontend`), similar models (from `nn.Module`) and problems (unsupported expressions of Python code). As a result, the generation result is an expected code snippet that tests the specific API by exploring more syntax features. After code generation, it detects a problem with the compatibility of `kwargs` syntax feature in PyTorch, which is a known issue (#29637). This is an effective test because Issue #29637 is not included in our dataset due to the lack of full reproducible code. YanHui shows its ability to find real-world MOBs by learning from similar examples and exploring new APIs.

Answer to RQ1. Compared with the state-of-art approach FuzzGPT-FS, YanHui can generate higher-quality test cases in terms of both validity (+11.4% on average) and relevance (+22.3% on average). The higher efficiency of code generation reduces the disambiguation steps for finding real MOBs, and improves the efficiency of testing.

4.3 RQ2: Usefulness for Finding MOBs

Patterns of MOB-triggering code. To analyze the characteristics of the useful test cases generated by YanHui, we review the generated code and categorize the patterns of the code that can trigger MOBs. We find that YanHui can generate code in three ways that are effective in finding MOBs.

- **API** that has not been handled properly by the optimizer. For example, the layer `DynamicQuantizedLinear` has unsupported operation issues for quantization (PyTorch #110515).
- **Desugar**ing specific language features of Python during optimization may cause failures. For example, the `in` operator for `str` will be expanded to a special abstract syntax tree which the optimizers fail to handle (PyTorch #46682).
- **Illogical** code that is not logically meaningful but syntactically correct, for example, a custom layer that simply raises an exception instead of calculating data (PyTorch #12118).

MOBs found with LLMs. We compare the number of MOBs found with FuzzGPT-FS and YanHui in Table 3. Both approaches can detect MOBs by exploring new APIs, and YanHui detects more with the guidance of error-prone parts of optimizers. YanHui can further detect MOBs by generating test cases of desugaring and illogical code, which FuzzGPT-FS do not involve.

Examples of found bugs. We explain the usefulness of YanHui in finding MOBs with two illustrative examples.

Listing 1 shows an example of the missing supporting types [16] in the PyTorch JIT module. In this test case, PyTorch is trying to parse and process a dataclass, which is introduced in Python 3.7. The script method of PyTorch does not process the `@dataclass` decorator properly and throws an error when trying to access the code implementation of dataclass.

By checking the input prompts, none of the examples includes the usage of dataclass, while they are all related to the API called `torch.jit.script`. Therefore, the generated code is mutated to test new Python features without breaking the structure of the code snippet. Such kind of test cases are categorized as *Desugar* in Table 3. This bug is confirmed and discussed in PyTorch #66017.

```
@torch.jit.script
@dataclass(frozen=True)
class Info:
    # ...
    pass
class Model:
    def __init__(self, info: Tuple[Info]):
        self.infos = [i for i in info]  # error here
    def forward(self, x):
        return self.infos
```

Listing 1: An OSError found by fuzzing the dataclass feature of Python

Listing 2 lists an example of missing supporting operations [16] bug when scripting a model. This test case reveals a problem that PyTorch cannot compile a function that tries to access the property `fc1` in `self`. PyTorch library fails to get the full definition of the model, and the initialization of `fc1` is not captured. Therefore, the model cannot be compiled into optimized code.

This example is a typical test of illogical code, because scripting the forward method of the model is not common and should not take effect. Such cases are not easy to find with traditional code generation approaches. They are categorized as *Illogical* in Table 3.

```
class Model(tnn.Module):
    def __init__(self, h: int = 16) -> None:
        super().__init__()
        self.fc1 = tnn.Linear(h, h)

    @torch.jit.script
    def forward(self, x: torch.Tensor):
        return torch.tanh(self.fc1(x))
m = Model(42)
y = m(torch.randn(3, 5))
```

Listing 2: A RuntimeError found by accessing properties in self

Answer to RQ2. YanHui is useful for finding MOBs by fuzzing the API and models. Based on the examples from historical issues, LLMs can generate innovative code for testing the model optimizers in depth. The knowledge can help the generation process to focus on the relevant part and mutate with untested API or syntax and write uncommon code. Apart from API bugs, YanHui shows the ability to find MOBs that occur during desugaring Python language features and handling illogical code.

4.4 RQ3: Generalization and Ablation Study

To figure out how YanHui takes advantage of domain knowledge, we first investigate whether the concentration and diffusion prompting paradigm can generalize to different LLMs, then choose one model to conduct ablation studies.

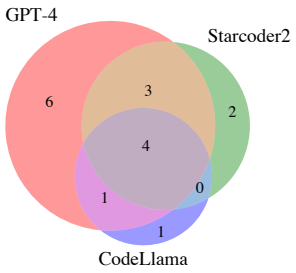

Figure 9: MOBs found by YanHui with different LLMs

Table 4: YanHui with or without Code Structure

Components	Outputs	Valid	Relevant	MOB
With code structure	595	208 (35.0%)	188 (31.6%)	9
Without code structure	595	144 (24.2%)	**67 (11.3%)**	5

Table 5: YanHui with or without Error Description

Components	Outputs	Valid	Relevant	MOB
With error description	595	208 (35.0%)	188 (31.6%)	9
Without error description	595	201 (33.8%)	**102 (17.1%)**	**4**

Generalization. Figure 9 shows the number of detected MOBs when using different LLMs for YanHui. It shows that 8 out of 17 MOBs overlap among different LLMs, which means the high-quality prompts derived from concentration and diffusion can be generalized to different models.

Next, we use the locally reproducible Starcoder2 model for the ablation studies, which remove each component of prompts, i.e., code structure, error description and mutation rules.

Influence of code structure. Table 4 presents the quality of generated code and detected MOBs with or without the component of code structure.

The comparison shows that the code structure mainly influences the ratio of valid and relevant code generated by LLMs. The detected MOBs are also affected due to the lack of effective code for testing. Among the metrics, the code structure shows a significant impact on the relevant code, which can be explained by the format of the code. By mentioning each step of the model optimization program, LLMs are more likely to generate the expected code after chain-of-thought.

Influence of error description. Table 5 presents the metrics of generated code when involving or not involving error description.

The metric differences imply that the error description component can also affect the relevant code and detected MOBs in a similar way of the code structure. The characteristics of error may highlight the code that has problems. Therefore, such information can enhance the generation quality.

Influence of mutation rules. Table 6 presents the metrics of generated code by YanHui with or without mutation rules.

Table 6: YanHui with or without Mutation Rules

Components	Outputs	Valid	Relevant	MOB
With mutation rules	595	208 (35.0%)	188 (31.6%)	9
Without mutation rules	634	201 (31.7%)	149 (23.5%)	**2**

The results are highlighted by the significant drop in detected MOBs, which indicates that the directions are effective in exploring the error-prone part of the framework.

Answer to RQ3. YanHui can be generalized to work with different LLMs. All of the three components of YanHui contribute to the improvement of code quality and detected MOBs. The code structure is especially important to relevant code, and mutation rules are effective in generating bug-triggering cases. YanHui achieves the best performance with all components included.

5 Discussions

5.1 Quality of Text Summarization

The step of text summarization in Section 3.3 may contain errors due to human work or LLM generation. To control the quality of summarization, we involve another author to validate the results. The processes of labeling MOBs [16] are followed - two authors summarize the issues independently, and then discuss until all the inconsistencies of results are resolved. Among the summarization by LLMs, 92% of the generation satisfies the format of error characterization. Although the description may not be perfect, it can still provide effective steps of error reasoning for the chain-of-thought prompting as shown in Section 4.4.

5.2 Limitations

Our study focuses on the test generation of model optimization bugs, which is a new and specific scope of deep learning software testing. As an early attempt at this problem, our study still has limitations, which we plan to resolve in our future work. The collected MOB examples are still small, because we focus on the quality of the issues, and only choose the closed issues which have been fixed by pull requests. Although the chosen issues are effective in mining information, a number of other issues can be considered to improve the variety of the example MOBs significantly. We plan to keep improving the model for labeling to achieve automatic MOB summarization from open issues. With more examples for the prompts, LLMs can generate code to test deeper in the library and cover more functionalities of model optimization.

5.3 Threats to Validity

The potential threats to the validity of our study are the stability of the LLM outputs and the testing environment of the generated code. To mitigate the unstable LLM outputs, we use a fixed setup of parameters and random seed. However, the outputs of GPT-4 cannot be controlled because it is a black-box online service. Our testing environment contains only Nvidia GPU, therefore, some code snippets requiring other platforms will cause a failure on the testing machine, which is a false positive. In our experiments, we

exclude such false positives by recognizing the clear name of the specific platform, such as MKL-DNN for Intel, and MPS for Apple.

6 Related Work

Testing deep learning libraries. Researchers have extensively studied the bugs and testing in deep learning libraries, especially the training and inference parts. To understand the characteristics of bugs, a great deal of literature [4, 5, 7, 16, 22, 40] has analyzed the symptoms and root causes of various deep learning framework bugs, including common program bugs, performance bugs, deployment bugs, compiler bugs, and the emerging optimization bugs.

To test with deep learning models, CRADLE [35] introduces an effective test oracle for deep learning libraries, which adopts differential testing [29] for deep learning models. The framework bugs are exposed when the model outputs have any inconsistency. Based on the test oracle, Audee [17] focuses on fuzzing Deep Neural Network (DNN) models for testing. It uses existing popular DNN models as seeds, and mutates the input and weight to generate new models for testing. LEMON [49] proposes an effective approach to generating models by introducing heuristic rules to guide the mutation and increase the probability of exposing bugs. NNSmith [27] leverages SMT solvers to generate various DNN models that satisfy the constraints in the computation graph.

Apart from the inputs and models, API is also an aspect of testing. FreeFuzz [50] collects API calls from open-source code. By identifying the parameters in the function calls, it can fuzz deep learning libraries by mutating the parameters to generate new test cases. DocTer [55] aims to fuzz the API functions by mining the documentation from the deep learning libraries.

Fuzzing deep learning libraries with LLMs. Recently, with the emergence of Large Language Models, researchers are adopting this powerful utility for software testing. TitanFuzz [9] first introduces LLMs to fuzzing DL libraries. It generates a seed input with prompts mentioning API, and mutates the input by masking some parts of the code and asks LLMs to fill in the masks. FuzzGPT [10] applies few-shot prompts with examples of code and descriptions to generate testing code.

7 Conclusion

In this paper, we propose YanHui, an effective approach for finding Model Optimization Bugs (MOBs) via the test code generated by Large Language Models (LLMs). The generation is prompted with chain-of-thought and contains informative knowledge and instructions after concentration and diffusion. This paradigm produces high-quality prompts, which focus on the scope of testing MOBs. Our experiments show that the ratio of valid/relevant code can be improved by 11.4%/22.3%. With the 595 generated testing cases, 17 real bugs can be found, including five deep MOBs associated with uncommon code, which are typically difficult to detect. Our analysis on the detected bugs reveals that the mutation rules derived from domain knowledge can help LLMs generate more specialized tests to expose bugs. YanHui showcases the potential of LLMs in software testing and explores a practical way to detect specific types of bugs.

Acknowledgment

We thank our anonymous reviewers for their constructive comments and insightful suggestions on the paper. This work is partially supported by the National Natural Science Foundation of China (Grant No. 61932021) and the Australian Research Council Discovery Projects (DP230101196, DP240103068).

References

[1] Meta AI. 2023. Llama 2. https://ai.meta.com/llama/. Accessed: 2023-09-22.
[2] Mihalj Bakator and Dragica Radosav. 2018. Deep learning and medical diagnosis: A review of literature. *Multimodal Technologies and Interaction* 2, 3 (2018), 47.
[3] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep Learning with Low Precision by Half-Wave Gaussian Quantization. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Institute of Electrical and Electronics Engineers, 5406–5414. https://doi.org/10.1109/CVPR.2017.574
[4] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 357–369. https://doi.org/10.1145/3540250.3549123
[5] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (sep 2023), 31 pages. https://doi.org/10.1145/3587155
[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[7] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 674–685. https://doi.org/10.1109/ICSE43902.2021.00068
[8] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. PentestGPT: An LLM-empowered Automatic Penetration Testing Tool. arXiv:2308.06782 [cs.SE] https://arxiv.org/abs/2308.06782
[9] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067
[10] 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
[11] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. https://doi.org/10.1145/3540250.3549085
[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423
[13] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using

metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 118–128. https://doi.org/10.1145/3213846.3213858

[14] The Linux Foundation. 2022. PyTorch. https://pytorch.org. Accessed: 2022-08-14.

[15] Zhipeng Gao. 2021. When deep learning meets smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1400–1402. https://doi.org/10.1145/3324884.3418918

[16] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Institute of Electrical and Electronics Engineers, 147–158. https://doi.org/10.1109/ICSE48619.2023.00024

[17] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2021. Audee: automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 486–498. https://doi.org/10.1145/3324884.3416571

[18] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[19] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1158–1161. https://doi.org/10.1109/ASE.2019.00126

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[21] Google Inc. 2022. TensorFlow. https://www.tensorflow.org. Accessed: 2022-08-14.

[22] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *International Conference on Database Systems for Advanced Applications*. Springer, 604–620.

[23] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. 1–12.

[24] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599* (2023).

[25] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*. PMLR, 2849–2858.

[26] Geert Litjens, Clara I Sánchez, Nadya Timofeeva, Meyke Hermsen, Iris Nagtegaal, Iringo Kovacs, Christina Hulsbergen-Van De Kaa, Peter Bult, Bram Van Ginneken, and Jeroen Van Der Laak. 2016. Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis. *Scientific reports* 6, 1 (2016), 1–11.

[27] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[28] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

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

[30] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A multilanguage static analysis of python programs with native C extensions. In *International Static Analysis Symposium*. Springer, 323–345.

[31] Bence Nagy, Tibor Brunner, and Zoltán Porkoláb. 2021. Unambiguity of Python Language Elements for Static Analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 70–75.

[32] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[33] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, and Shawn Jain et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[34] 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).

[35] 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)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[37] Sebastian Ramos, Stefan Gehrig, Peter Pinggera, Uwe Franke, and Carsten Rother. 2017. Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling. In *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1025–1032.

[38] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI EA '21)*. Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages. https://doi.org/10.1145/3411763.3451760

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[40] 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 (ESEC/FSE)*. 968–980.

[41] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227.

[42] 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 (ISSTA)*. 690–702.

[43] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. 2019. Release Strategies and the Social Impacts of Language Models. arXiv:1908.09203 [cs.CL]

[44] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[46] Guanyu Wang, Yuekang Li, Yi Liu, Gelei Deng, Tianlin Li, Guosheng Xu, Yang Liu, Haoyu Wang, and Kailong Wang. 2024. MeTMaP: Metamorphic Testing for Detecting False Vector Matching Problems in LLM Augmented Generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. 12–23.

[47] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP*.

[48] Zihan Wang, Zhongkui Ma, Xinguo Feng, Ruoxi Sun, Hu Wang, Minhui Xue, and Guangdong Bai. 2024. CoreLocker: Neuron-level Usage Control. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. https://doi.org/10.1109/SP54263.2024.00233

[49] 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 (ESEC/FSE)*. 788–799.

[50] 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.

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[52] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. https://doi.org/10.1145/3491102.3517582

[53] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. 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 126, 13 pages. https://doi.org/10.1145/3597503.3639121

[54] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[55] 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.

[56] Chuan Yan, Mark Huasong Meng, Fuman Xie, and Guangdong Bai. 2024. Investigating Documented Privacy Changes in Android OS. *Proc. ACM Softw. Eng.* 1, FSE, Article 119 (jul 2024), 24 pages. https://doi.org/10.1145/3660826

[57] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 627–638. https://doi.org/10.1145/3468264.3468612

[58] YanHui-2024. 2024. *YanHui-2024/YanHui: 2024-07-05*. https://doi.org/10.5281/zenodo.12671638

[59] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AutoTrainer: An Automatic DNN Training Problem Detection and Repair System. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 359–371. https://doi.org/10.1109/ICSE43902.2021.00043

[60] Hao Zhong. 2023. Enriching Compiler Testing with Real Program from Bug Report. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. https://doi.org/10.1145/3551349.3556894

[61] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv:1710.01878 [stat.ML] https://arxiv.org/abs/1710.01878

Received 2024-04-12; accepted 2024-07-03