Investigating and Detecting Silent Bugs in PyTorch Programs

Shuo Hong*, Hailong Sun*, Xiang Gao*†, Shin Hwei Tan‡
*Beihang University, ‡Concordia University
*{shuohong, sunhl, xiang_gao}@buaa.edu.cn, ‡shinhwei.tan@concordia.ca

***Abstract*—Deep Learning (DL) has been widely applied in various fields. Unlike traditional software, DL programs possess the "black box" characteristic that can make it challenging for developers to debug when anomalous behaviors arise. In particular, silent bugs, a type of bugs in DL programs, can lead to erroneous behaviors without causing system crashes or suspensions, and they do not display error messages to users. This makes silent bugs more difficult for developers to discover, locate, and fix. In this paper, we present the first detailed study of silent bugs in PyTorch programs. We collect 14,523 posts from the official PyTorch forum and use a LLM-based semi-automated approach to filter the silent bugs. By analyzing the symptoms, root causes, and patterns of silent bugs, we have derived several important findings and implications: (1) most silent bugs cause abnormal outputs, which requires the design of more flexible test oracles to detect them, (2) the wide range of symptoms and root causes do not necessarily have one-to-one correspondences, which makes detecting and debugging silent bugs more challenging, (3) silent bugs exhibit common bug patterns, such as redundant, missing, or misplaced operations. Building upon these findings, we design and implement an extensible rule-based tool PYSIASSIST to help developer debug and resolve silent bugs. Evaluation results show that PYSIASSIST achieves 92.4% precision and 85.3% recall, outperforming existing techniques.**

***Index Terms*—Silent bugs, PyTorch Programs, Bug taxonomy.**

I. INTRODUCTION

Deep learning (DL) techniques have received widespread adoption in recent years. The systems built with machine/deep learning algorithms are often referred to as "software 2.0", as they rely on trained models to make decisions rather than human-written code. Unlike traditional software, where developers encode program logic in human-readable source code, trained DL models define decision logic using a set of unreadable neural networks and weights. The "black-box" nature of DL models makes it extremely challenging to debug when they exhibit unexpected behaviors.

Silent bugs, which is also referred as functional or numerical errors, are defined as bugs that do not result in system crashes, hangs, or displays error messages to users, but can lead to incorrect behavior [1]. Silent bugs in DL programs may cause a range of problems, such as erroneous output, low accuracy, training that does not converge, extra execution time, or excessive memory usage. These problems could affect the normal execution of applications, and cause catastrophic results if DL models are deployed in safety-critical domains (e.g., autonomous vehicles). Silent bugs are more problematic than more traditional bugs [1] so it is important to deal with silent bugs in DL programs properly and timely. However, due to the "black box" and stochastic nature of DL models, silent bugs are particularly difficult to debug and fix.

Several studies investigated silent bugs in deep learning frameworks (e.g. Keras[2] and TensorFlow[3]), revealing that these bugs can cause problems such as performance degradation, incorrect calculations, and incorrectly displayed messages [1]. However, existing studies only focus on silent bugs in the DL frameworks and did not investigate silent bugs in user programs. On the other hand, recent research has investigated the characteristics of bugs in DL user programs [4, 5, 6]. However, these studies focused on the characteristics (e.g., symptoms, root causes, and taxonomy) of all kinds of bugs where many of them having obvious symptoms (e.g., compilation errors, crashes, or assertion failures). They did not specifically target silent bugs, and hence only captured partial characteristics of silent bugs in DL programs. As it will be shown in our evaluation, the state-of-the-art general-purpose bug detector CodeGuru [7] can only discover 1.5% silent bugs. Moreover, prior studies focused on specific DL bugs (e.g., performance bugs [8, 9, 10], API usage bugs [11], program failure [12]). A comprehensive study of the characteristics of silent bugs in DL programs will help developers and researchers to better understand, localize and fix these bugs.

Studying silent bugs in DL programs is important due to two reasons. (1) Silent bugs affect all developers regardless of their expertise levels. For example, our study found that even developers with more than a year of experience may still raise questions regarding "simple" silent bugs [13, 14, 15, 16]; (2) Compared to other DL bugs, it is difficult to differentiate between a misjudgment and the underlying root cause. For example when `zero_grad()` is missing, gradients will accumulate continuously, leading to incorrect final training results. If the developers were to optimize the dataset, the training results might improve, which could mislead them into thinking that the root cause is the unoptimized dataset.

To bridge the gap in understanding silent bugs in DL user programs, we present a comprehensive study of silent bugs in PyTorch [17] programs. Several popular DL frameworks (PyTorch [17], Keras [2], and TensorFlow [3]) are widely used to write DL applications. Our study focuses on PyTorch because: 1) PyTorch is the most widely adopted DL framework

†Xiang Gao is the corresponding author.

according to the statistics of AssemblyAI in 2023 [18], 2) compared to TensorFlow, PyTorch exhibits better compatibility across different versions [19], which reduces the occurrence of silent bugs caused by version discrepancies, and 3) compared to other DL frameworks (e.g., Keras), PyTorch has its independent official forum [20] with an active user community, which provides a large number of high-quality posts, discussions, and solutions for fixing bugs. In addition, there are many causes for the occurrence of silent bugs in PyTorch programs, such as errors in dataset annotation and issues with hardware facilities.

To understand the silent bugs in PyTorch programs, we collected 14,523 posts from the PyTorch official forum. Since manually analyzing and filtering all posts are too time-consuming, we use Large Language Model (LLM) to assist in the filtering process. LLM (i.e., BingChat in this paper), achieves a 74.4% accuracy in identifying irrelevant posts, which helps save time and effort with reasonable accuracy. Finally, we collect 365 silent bugs for further investigation.

By analyzing the symptoms, root causes and bug patterns of the silent bugs, we have derived several key findings: (1) around 80% of silent bugs lead to abnormal outputs, (2) improper operations during training and execution are the main reasons causing silent bugs, and (3) silent bugs exhibit common patterns (e.g., redundant, missing or misplaced operations). Developers of PyTorch programs may benefit from the taxonomy derived from our study. It can assist them in understanding the nature of silent bugs in their code and provide guidance for debugging and fixing such issues. Our study also provides useful development suggestions for developers, especially novice developers, to prevent silent bugs.

Due to the silent nature, it is hard for developers, especially novice developer, to discover, debug and resolve silent bugs. To help developer in resolve **Py**Torch **Si**lent bugs, we design and implement a static programming assistant tool called PYSIASSIST. PYSIASSIST is **not** designed to detect silent bugs in the wild, instead, it is an assistant for developers, especially novice developers, to discover silent bugs in their code in the early development stage. Our evaluations on 213 testing snippets show that the PYSIASSIST can correctly detect silent bugs with 92.4% precision and 85.3% recall. Furthermore, we also extend PYSIASSIST to support the detection of more bug patterns, demonstrating its extensibility. In contrast, CodeGuru [7, 21], a general PyTorch bug detector, only achieves 1.5% recall, whereas ChatGPT only achieves 76.1% precision and 51.9% recall (relying only on ChatGPT results in relatively low accuracy).

The contributions of this work are summarized as follows:

- We present the first comprehensive study of silent bugs in PyTorch programs. Our study reveals a set of key findings and implications on the symptoms, root causes, and bug patterns of silent bugs.
- We design and implement PYSIASSIST, a static assistant that help developer handle silent bugs in PyTorch programs. Evaluation results show that PYSIASSIST achieves 92.4% precision and 85.3% recall.

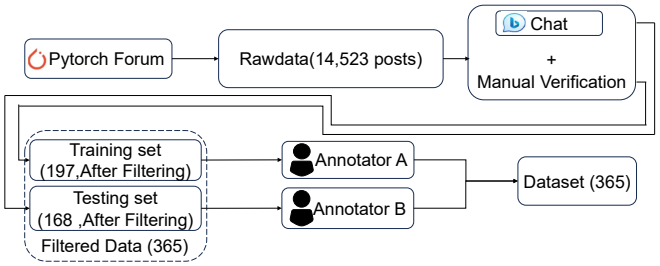

Fig. 1. Dataset construction workflow

- We create a dataset of silent bugs, including 365 posts of silent bugs. The dataset and implemented tool are open-source and available to benefit the research community[22].

II. DATASET CONSTRUCTION

To understand and recognize silent bugs in PyTorch programs, we first collect a set of posts and discussions from PyTorch forum [20]. The goal is to identify the silent bugs frequently encountered by developers and their corresponding fix strategies. The construction of the entire dataset falls into three processes: *data collection*, *data filtering*, and *data annotation*. An overview of the entire construction process is presented in Figure 1.

A. Data Collection

Information regarding silent bugs was gathered from the official PyTorch forum, a well-known Q&A website, where developers from around the world can discuss problems related to PyTorch. Due to the large number of posts in the official PyTorch forum, and new posts are continually added every day, it is impractical to collect all the posts from the forum. Hence, we select a set of representative posts based on their popularity and keywords.

Data collection based on popularity: PyTorch forum provides a ranking of the posts based on popularity, measured by the number of "viewed" counts, representing the most common problems encountered by developers. As we need to understand how a silent bug is fixed, we only select the posts with tag "solved". The solved post usually includes discussions about bug symptoms, bug patterns, root causes, and their analysis, which can help us understand a bug. Based on the ranking of the forum, we obtained 13,241 most popular posts (the largest number of posts allowed by the forum).

Data collection based on keywords: Although posts based on popularity have a certain level of representativeness, there is still a possibility that some posts related to silent bugs are not included in these 13,241 posts. We want to collect as many of these excluded silent bug posts as possible to improve the representativeness of the dataset. We used the method of data collection based on keywords to collect as many posts related to silent bugs as possible. To achieve this goal, we first choose posts that have been marked as "solved" in their tags and then search for the keyword "silent bugs". However, this approach yields only one post. This is mainly because developers pay more attention to the bug itself instead of the bug properties (i.e., "silent"). According to the definition, silent bugs do not

result in error messages, system crashes, or hangs. Hence, they could cause a wide range of problems (e.g., erroneous output, low accuracy, and high loss). Referring to the method used by Zhang et al [6] for constructing keywords, we obtained five keywords: 'performance', 'accuracy', 'learning rate', 'loss', and 'slower'. With those keywords, we obtained 1,922 posts. After merging and de-duplicating the posts collected by popularity and keywords, we finally have 14,523 posts.

Beyond understanding the silent bugs, we also aim to build a tool PYSIASSIST to help developer resolve silent bugs. To evaluate the performance of the tool, we **randomly** shuffle and equally divide the 14,253 posts into a training set and a testing set. We equally divide the dataset because we hope both training and testing sets adequately represent the overall distribution of the silent bugs. The training set will be used to build the general tool, while the testing set is used to validate its effectiveness.

B. Data Filtering

To ensure that the constructed dataset meets our objectives, we filter the dataset based on the following criteria:

- As our research targets silent bugs, the initial questions and code in the selected posts must **not** contain error message.
- The question in selected posts should encompass either source code or clear bug description, which helps us identify and analyze silent bugs accurately and efficiently.
- The answers in the selected posts should unambiguously provide the precise location and the cause of the silent bug.

Though manual filtering could be accurate, it may take unaffordable time given the large number of posts. Recently, LLMs like ChatGPT have been proven to have good text understanding capabilities, thus, ChatGPT is used to filter out obviously irrelevant posts. BingChat is built on top of the GPT4 model, and we rely on BingChat mainly because it can retrieve post content online. Then, we manually review the remaining ones to precisely select the posts we need. The data filtering process consists of the following three steps:

Step 1: Feasibility checking: As prior study showed promising results in using LLMs like ChatGPT for text comprehension and web content filtering [23], we present the first feasibility study of using LLMs for filtering forum posts in the context of conducting our empirical study. Specifically, we first randomly select a certain number of posts, manually annotate these posts, provide the URLs of these posts to BingChat, and then check the classification accuracy of BingChat. We choose 200 posts from the original data, annotate them as silent bugs or not, and develop and refine several candidate prompts. To avoid subjective factors, we adopted a method similar to Yang et al. [24]. Specifically, we manually created a few prompts and then let the LLM optimize them to produce a set of candidate prompts. Subsequently, we derive the following prompt that achieves the best result: *url + I want you to act as a URL checker for a question-and-answer post. I have given you the URL of the post and you should check if the initial question in that post has no error message and if it has a clear problem description or includes source code. Does this post meet the criteria? Please provide a reason using the format shown below: 'Yes (or No), the reason is that the post . . . ', and do not wrap!*. We manually analyze the results for the 200 posts and calculate the percentage of correctly classified posts. With the above prompt, BingChat achieves 74.4% accuracy.

Step 2: Preliminary filtering through BingChat: After the feasibility checking, we use BingChat with the best prompt to classify these 14,523 posts twice. This is to avoid the randomness of the output from BingChat as a large model itself and obtain results as accurately as possible. If the filtering results are 'yes' (meaning it is a silent bug) for both times, the corresponding post will be retained.

However, some of these posts are still unusable due to errors made by BingChat in its judgment, since defining a clear and concise problem description is a challenging task, even for humans. It is unrealistic to expect BingChat to classify every post correctly, and this motivates us to manually review the remaining posts.

Step 3: Manual data filtering: After BingChat filtering, around 14% of the posts remain relevant to silent bugs. Then, two authors manually analyzed all the remaining training and testing posts independently by reading the description and filtering them based on their joint agreement. To minimize the impact of subjective factors on the dataset quality, we conservatively removed ambiguous posts in the filtering process (i.e., if the description is unclear or two author can't reach a consensus whether it is a silent bug, we simply removed the posts). Finally, we collected 365 posts, including 197 posts from training set and 168 posts from testing set.

It is important to note that, during the data collection stage, we gathered as many popular posts as possible, and in the data filtering stage, we made efforts to minimize subjective factors. Therefore, we believe that these 365 posts can to some extent represent the general situation.

C. Data Annotation

To understand silent bugs, we manually annotated the dataset. The training and testing sets were annotated by the first two authors of this paper. Both authors have more than two years of experience of using PyTorch. Among them, Annotator A, who is responsible for annotating the training set, is also the developer of PYSIASSIST. The dataset is annotated from the following dimensions:

- *Symptoms.* refer to the abnormal phenomena mentioned in the posts, including too low accuracy (acc), loss not decreasing, or turning into NaN, etc.
- *Root cause.* refers to the causes of these abnormalities mentioned in the posts, such as the erroneous selection of loss functions, etc. It also involves the stages where the error occurs, including data preprocessing, data input, model training, evaluation, or GPU(s) configuration.
- *Buggy code patterns*. refers to the patterns of buggy code, for instance, API parameter errors, API usage errors, missing APIs, etc. PYSIASSIST will be built based on these abstracted buggy code patterns.

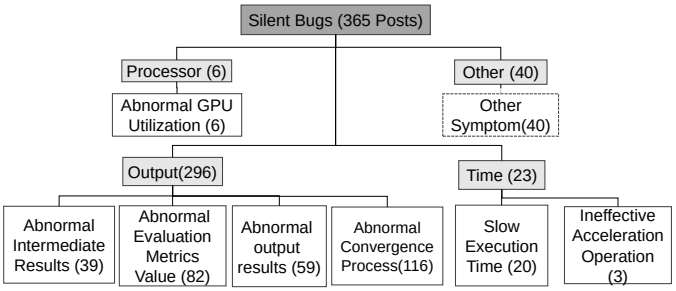

Fig. 2. Taxonomy of silent bugs symptoms

All the dimensions are used to annotate both training and testing set. The annotators meticulously review all aspects of the posts, which include the title, the question description, comments, answers, and any reference links mentioned during the discussion, to label silent bugs.

As the training and testing sets are annotated by two authors independently, we evaluate the reliability of our annotation process to prevent any potential bias. Among these dimensions, *Symptoms* are typically clearly described by the questioners, making it unlikely to introduce bias. Similarly, given the root cause, bug patterns can also be easily annotated. Hence, the most critical dimension in the entire dataset is the *Root cause*. To validate the reliability of root cause annotation, two annotators switched roles and re-annotated the root cause. Compared to the original annotation results, the new annotation results showed no significant difference, indicating that the annotation has a certain degree of reliability. Note that this process was carried out after the construction of the PYSIASSIST and the extraction of testing code snippets from the testing set posts to avoid any bias in the processes of building and evaluating PYSIASSIST.

III. EMPIRICAL RESULTS

We start by analyzing the symptoms and root causes of silent bugs within the dataset and then summarize patterns of silent bugs. It should be emphasized that these results are the aggregated analysis of the training and testing set.

A. Symptoms Analysis

Figure 2 shows the taxonomy of silent bug symptoms, which is composed of three primary categories: *Output*, *Processor*, and *Time*, which are further divided into seven subcategories. 325 instances (representing 89.0% of all cases) of the total 365 silent-bug posts belong to these three categories. The remaining 40 posts (11.0%) belong to the *Others* category.

1) Output. This category contains silent bug posts with issues in output results, accounting for 295 (80.9%) of the silent-bug posts. The *Abnormal Convergence Process* subcategory contains 116 posts (31.8%) that exhibit abnormal convergence in evaluation metrics such as accuracy (ACC) and loss. Such abnormal behaviors include sudden decreases in ACC, unchanging loss, continuously increasing loss, and etc. Subcategory *Abnormal Evaluation Metrics Value* includes 82 posts (22.5%), exhibiting abnormal values accuracy (ACC) and loss. Compared to the *Abnormal Convergence*, *Abnormal Evaluation Metrics Value* emphasizes the presence of extreme values, e.g. excessively large or small values. For example, ACC remains to be 0, or the value of loss is set as `NaN`. Moreover, 59 posts (16.2%) exhibit *Abnormal Output Results*, such as consistently predicting only a specific category during validation, and 38 posts (10.4%) exhibit *Abnormal Intermediate Results*. For example, during training, a bug may cause the gradient of a certain layer to remain unchanged.

2) Time. Silent bugs in this category, accounting for 23 posts (6.4%), cause abnormal execution time. *Slow Execution Time* (20 posts) refers to the increase of execution time including training time, inference time, forward propagation time, backward propagation time, etc. Three posts (accounting for 0.9%) belong to *Ineffective Acceleration Operation*. For example, when setting `Num_workers` > 0 to utilize multiple GPUs, but the execution time is unexpectedly increased.

3) Processor. This category only contains 6 posts (1.7%), with the subclass *Abnormal GPU Utilization* caused by low GPU utilization, GPU not used, inability to use multiple GPUs, etc. Among them, *low GPU utilization* refers to the situation where the GPU is underutilized during training or inference processes, e.g., the GPU utilization is less than 10%. *GPU not used* means that GPU utilization is 0%, indicating that it is not being utilized at all. *Inability to use multiple GPUs* indicates that only one or a few GPUs are used although there are multiple GPUs available.

Summary. *More than 80% of silent bugs in DL programs result in abnormal outputs.* Around 6.4% of silent bugs cause program slowdown, while 1.7% of them lead to abnormal GPU utilization. The diverse symptoms exhibited by silent bugs make it challenging for developers to determine whether these anomalies arise from code bugs, poor data quality, errors in the DL framework, or other factors. Hence, *understanding the root causes of the silent bugs is important for debugging*. Moreover, different from traditional programs where the developers can use assertions to check for the program's outputs, it is hard to write unit tests to validate the outputs of functions in DL programs. As most silent bugs lead to abnormal outputs, *developers or future researchers may consider designing test oracles that check for our identified types of abnormal outputs using a differential testing approach* (e.g., comparing the values of loss before and after calling certain APIs).

B. Root Cause Analysis

Figure 3 shows the taxonomy of root causes for silent bugs, including four high-level categories and 16 subcategories.

1) Training and Execution. This category encompasses silent bugs related to the model training and execution processes, representing the root causes of the highest number (196, 53.7%) of silent bugs. In this category, disrupting *dynamic* computational graph during training results in 47 (12.8%) silent bugs. Different from TensorFlow's static computational graph mechanism, PyTorch utilizes a dynamic computational graph mechanism. In each step, the dynamic mechanism first computes the values of preceding nodes. Based on these values, it then builds the computational graph for subsequent steps. Compared to the static mechanism, the

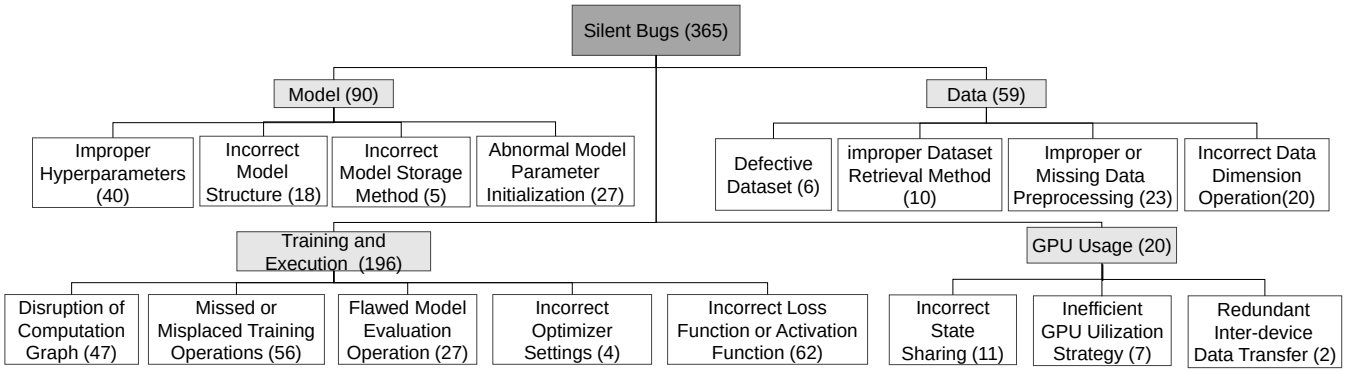

Fig. 3. Root causes of silent bugs in PyTorch

```
## from: "loss-backward-does-not-update/129991"
    logits.requires_grad = True
    labels.requires_grad = True
-   loss = Variable(loss, requires_grad = True)
    ...
    optimizer.zero_grad()
    loss.backward()
```

Fig. 4. Disruption of computational graph

```
## from: "simplest-lstm-possible-and-it-does-not-work
     /29507"
for j in range(1000):
+   optimizer.zero_grad()
    hidden = (...,torch.zeros((1,1,10),dtype=torch.double))
    ...
    loss.backward()
    optimizer.step()
```

Fig. 5. Missing or misplaced training operations

dynamic mechanism is more flexible, but it relies more heavily on the results of previous steps and requires more frequent graph constructions. Consequently, developers is more likely to disrupt the computational graph due to errors during training and execution. In this subcategory, the two most representative erroneous operations are using APIs such as `detach()` to detach tensors directly and destroy the computation result, and calling APIs (e.g., `Variable()`) to create tensors without adding the newly created tensors to the computation graph. Figure 4 shows an example where a new variable `loss` (the same name as an existing variable) is created using `Variable()` at line 4. However, the new `loss` is not added to the computational graph, leading to the destruction of the computational graph, resulting in the loss remaining unchanged. Fixing this bug involves deleting the `Variable()` at line 4, and continuing to use the original `loss`.

Missed or Misplaced Training Operations is another main root cause of silent bugs, encompassing 56 bugs (15.4%). This is mainly because developers may not be familiar with the results of each computational operation in the calculation process. Among them, the most common mistake is missing or misplacing one of the three operations: `optimizer.zero_grad()`, `optimizer.step()`, and `loss.backward()`. Figure 5 shows an example where developers forgot to call `optimizer.zero_grad()`, which can reset the gradients to zero to prevent accumulating values

Add prefix https://discuss.pytorch.org/t/ to the post source to get the URL of the post. The same applies to the following examples.

```
##from "training-acc-going-up-test-acc-doesnt-change/43349"
+   model.eval()
    with torch.no_grad(): # switch to evaluation mode
        images, labels = data
        ...
```

Fig. 6. Flawed model evaluation operation

from previous epochs. Missing `optimizer.zero_grad()` would cause *Abnormal Output Results*.

The lack of familiarity with the correct usages of loss and activation functions, together with their interrelationship, leads to *Incorrect Loss Function or Activation Function* utilization, causing 62 silent bugs (16.9%). In this subcategory, developers typically make two types of mistakes: 1) using inappropriate combinations of loss functions and activation functions, e.g., pairing `sigmoid()` with `NLLLoss()` (in fact, `NLLLoss` is typically used with `log_softmax`; 2) adding redundant activation function before loss function, e.g., adding `softmax` before `CrossEntropyLoss()` but an embedded activation function is already included in `CrossEntropyLoss()`.

During the model evaluation process, it is necessary to consider freezing certain parameters. Lacking a clear understanding of these operations leads to 27 silent bugs (7.5%). Figure 6 shows one representative example of a flawed model evaluation operation where the developer switches to evaluation mode using `with no_grad()`, but did not invoke the `model.eval()` API. This oversight can lead to unexpected updates of batch normalization (bn) layer. Conversely, using `model.eval()` without `with no_grad()` can result in unnecessary gradient computations during evaluation, potentially interfering with the model's performance and consuming extra memory resources. Similarly, some developers did not clarify which parameters to include in the optimizer or how to customize the optimizer to manipulate these parameters, resulting in 4 silent bugs (1.1%).

2) Model. There are 24.7% (90 posts) of bugs associated with DL models. Among them, 40 silent bugs (11.0%) are caused by *Improper Hyperparameter*. *Hyperparameter* is defined as numerical parameters that users need to specify before training, such as learning rate and momentum. To maintain clear classification, configurations relying on non-numerical values, e.g., loss function and optimizer selection, are not included in this category. In this subcategory, the most typical mistake is setting the learning rate to be too large or too small.

Improper usage of initialization methods may lead to *Ab-*

```
## from "large-performance-gap-between-pytorch-and-keras-
    for-imdb-sentiment-analysis-model/135659"
def __init__(self, vocab_size, em_dim, output_dim,...):
    self.embedding = nn.Embedding(vocab_size, ...)
    self.fc = nn.Linear(em_dim * maxlen, output_dim)
+   nn.init.uniform_(self.embedding.weight, -0.05, 0.05)
+   nn.init.xavier_uniform_(self.fc.weight, 1.0)
    ...
```

Fig. 7. Abnormal model parameter initialization

normal Model Parameter Initialization, causing 27 silent bugs (7.4%). Figure 7 shows an example where a developer tries to compare the performance between Keras and PyTorch by migrating a Keras model to PyTorch, but encounters lower accuracy. This is mainly because the default parameter in Keras is `Normal(0,0.05)`, whereas PyTorch uses a different default value `Normal(0,1)`. Fixing this bug requires to add initialization for the parameters in lines 5-6. Incorrect model structures (e.g., overlapping activation layers or missing fully connected layers) are classified in subcategory *Incorrect Model Structure*, which contains 18 silent bugs (4.9%). *Incorrect Model Storage Method* (1.4%) leads to errors during the executing or fine-tuning of pre-trained models.

3) Data. We defined *Data* as the input data of the model or loss function. Flawed data processing causes 59 (16.2%) silent bugs, including four subcategories: *Improper or Missing Data Preprocessing*, *Improper Dataset Retrieval Method*, *Defective Dataset* and *Incorrect Data Dimension Operation*.

Improper or Missing Data Preprocessing involves 23 silent bugs (6.3%). For example, a novice developer encountered the "abnormal convergence process" symptom (the loss is not decreasing) due to improper data preprocessing [25]. This silent bug can be resolved by adding proper data processing operations. Before the data preprocessing, *Improper Dataset Retrieval* and *Defective Dataset* are two other reasons that may cause silent bugs, causing 10 and 6 silent bugs, respectively. For example, a developer forgot to shuffle the dataset [26], which is classified as improper data retrieval, resulting in low accuracy (ACC). Typical examples for defective dataset involve imbalanced dataset and the failure to use *WeightedRandomSampler()*. The term *Data* in the above three subcategories primarily pertains to model's inputs. Subcategory *Incorrect Data Dimension Operation* involves both the input data of the model and loss function. This subcategory includes 20 silent bugs that resulted from feeding data with wrong dimensions into model or loss function.

4) GPU Usage. 20 silent bugs (5.4%) are resulted from improper GPU usage. When using multiple CUDA GPUs, *Incorrect State Sharing* result in inconsistent GPU state information for different tasks or threads, causing 11 silent bugs. For example, after using `.cuda()`, if `.synchronize()` is not used, it will lead to *Slow Execution Time*. Similarly, when using a specific GPU or multiple GPUs simultaneously, developers' unfamiliarity with GPU management can result in *Inefficient GPU Utilization Strategy*, which resulted in 7 silent bugs. For example, on a GPU supporting cudnn, not setting `torch.backends.cudnn.benchmark=True` can cause *Slow Execution Time* [27]. Moreover, *Redundant Inter-device Data Transfer* between CPU and GPU causes 2 silent bugs.

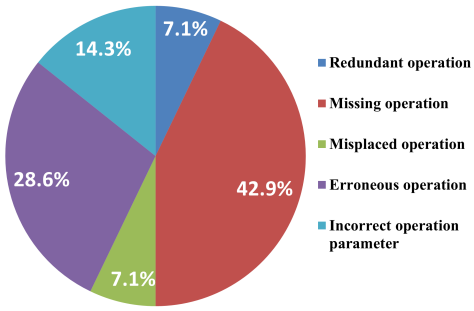

Fig. 8. The categories of bug patterns

Summary. Our study shows that more than half of the silent bugs are introduced by the category *Training and Execution*, around one-fourth of them are introduced by the category *Model*, approximately one-sixth of them are introduced by the category *Data*, and the remaining ones are caused by *GPU Usage*. The root causes of silent bugs are diverse and complex, and there is no simple one-to-one correspondence between the 16 types of root causes and the eight symptoms. Moreover, silent bugs do not generate error messages, and anomalies in a model may not be attributed to the code itself. Hence, developers may face challenges in identifying the existence and the scope of bugs in the code. In this case, our study of common root causes of silent bugs in PyTorch programs serves as a guideline for developers to help them better understand, debug, and fix these bugs.

C. Bug Patterns

With the analysis of the dataset, we discovered that there are many similar code segments that led to silent bugs. To automatically detect silent bugs, we identified 12 common patterns in the training set. Using the same approach, we identified 14 common patterns in the testing set, with 12 of them overlapping with the patterns from the training set. The number of posts corresponding to these patterns is 181, accounting for 49.6% of the total number of posts in the dataset. The remaining ones do not exhibit common features, e.g., improper learning rate does not have common patterns. We further classify these 14 patterns into five categories: *Redundant Operation*, *Missing Operation*, *Misplaced Operation*, *Erroneous Operation*, *Incorrect Operation Parameter*. In this context, *operation* refers to code actions (e.g., assignment, computation, function calls).

Redundant Operation refers to implementing the same functionality multiple times. For example, function `CrossEntropyLoss()` has already included an activation operation, hence when using `CrossEntropyLoss`, another activation function, e.g., `softmax()`, is not needed. Using `softmax()` and `CrossEntropyLoss()` together will be classified as redundant operation.

Erroneous Operation refers to situations where certain operations are incorrectly used in a specific context. For instance, in post [28], tensor dimensions should be transformed using `permute()`, but `view()` was mistakenly used.

Missing Operation refers to the situation where a specific operation is expected in the code context but is actually missing. The code shown in Figure 5 belongs to this category. In the code for model evaluation, `with no_grad()` that prevents the update of batch normalization layers is missing.

Misplaced Operation refers to the situation where the order of two or more operations is wrong. For example [29], the invocation of `zero_grad()` should be ahead of backward propagation. Misplacing `zero_grad()` after `backward()` will result in a higher loss than actual values to be predicted.

Incorrect Operation Parameter refers to the situation where the operation is correct, but some parameters are incorrect. For instance, when loading and splitting training/testing set, argument `shuffle` should be set as `True` instead of `False`.

We further measure the distribution of the bug patterns within the entire dataset (shown in Figure 8). We observe that *Missing Operation* is the most common pattern, accounting for 42.9%. *Erroneous Operation* and *Incorrect Operation parameter* come next, accounting for 28.6% and 14.3% respectively. *Misplaced Operation* and *Redundant Operation* are the least frequent with each of them accounting for only 7.1%. Overall, our study shows that *developers are more likely to introduce silent bugs due to forgetting certain operations or being unfamiliar with certain operations*. Redundant operations and misplaced positions are unlikely to cause silent bugs.

IV. Design and Implementation

Based on the results of our study, we design and implement PySIAssist, an assistant tool for helping developer resolve PyTorch Silent bugs. To prevent overfitting, we only use the training set for our design of PySIAssist.

A. Overall Design

We follow the design of CodeQL [30] and Joern [31] that detects bugs by first representing program as graphs and the quering the graphs to check against pre-defined patterns. Figure 9 shows the workflow of PySIAssist with three key modules: (1) *Configuration*, (2) *Graph Generation*, and (3) *Bug Detection*. The Configuration module allows users to specify bug patterns by providing configuration files. The Graph Generation module parses the code under test into graph representations, including Control Flow Graphs (CFG) and Data Flow Graphs (DFG). The Bug Detection module queries the constructed graphs to check against the patterns defined in the configuration files.

Configuration. The goal of the configuration module is to provide an interface for users to specify bug patterns and extend PySIAssist easily. There are several reasons to support such an extension:

1) According to empirical results, 12 patterns only cover 49.6% of the posts, while the remaining 50.4% bugs follow different patterns.

2) As the most popular DL framework, PyTorch is rapidly evolving, indicating that patterns may require frequent update.

3) Our dataset is based on posts from the PyTorch forum, which is representative but may not cover all cases.

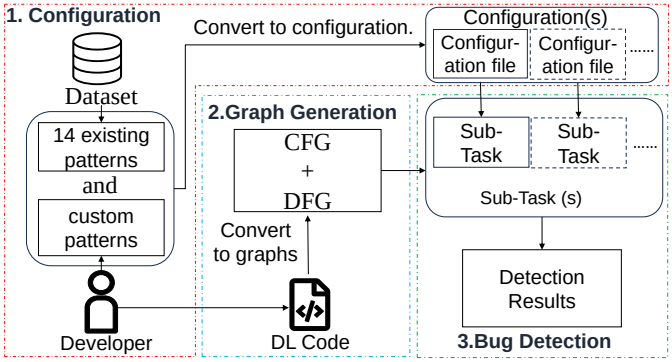

Fig. 9. An Overview of PySIAssist

With the extensible framework, supporting new patterns only requires the addition of corresponding configuration files, and there is no need to modify or rebuild the tool itself. With the configuration files provided by users, PySIAssist could automatically translate them into formal patterns. Then, PySIAssist will be able to recognize corresponding types of bugs based on the respective formal patterns.

Graph generation. The code under test is first parsed into CFG and DFG, which will facilitate the subsequent detection of PySIAssist based on these graphs. We choose to convert the code into CFG and DFG because : (1) control flow graph (CFG) represents all paths that might be traversed by a program during its execution [32], and (2) data flow graph (DFG) represents data dependencies between several operations [33]. As PyTorch is based on dynamic computational graphs, it may encounter silent bugs during the execution process of rebuilding the computational graph. Utilizing CFG and DFG can better simulate the program execution, thereby making PySIAssist more accurate in detecting silent bugs.

Bug detection. The detection module searches for paths in the CFG within the specified range based on the configuration, and then checks whether the sequence of operations along a certain path matches any bug patterns. In PySIAssist, a detection task is decomposed into a set of sub-tasks where each of them is simple and standalone. For example, detecting the bug in Figure 6 can be decomposed into two sub-tasks: (1) detecting whether *eval()* is invoked before *no_grad()* to avoid BN layer configuration errors, and (2) detecting if there is a *no_grad()* after the *eval()* to prevent an unwarranted deletion of computational resources. The combination of sub-tasks enables the detection of more complex patterns. Note that PySIAssist can be used to analyze both partial and complete PyTorch programs.

B. Implementation

The implementation details of three modules are as follows:

Configuration. PySIAssist stores configuration information in JSON files. The configuration parameters include graph to explore (CFG, DFG, or both), detection scope (upstream context, downstream context, or the entire document), detection start/end flag, and etc. Currently, PySIAssist includes configurations corresponding to the 12 patterns.

Graph Generation. We utilize Scalpel [34], a Python static analysis tool, to generate CFG and DFG. CFG supports the control dependency analysis and build control flows. DFG supports data dependency analysis in the upstream and downstream contexts. Users can choose to use DFG, CFG, or both.
Bug Detection. The detection module recognizes silent bugs by traversing the constructed graphs and check against the patterns defined in the configuration file. PYSIASSIST will inform developer when the given PyTorch code satisfies a bug pattern. Although PYSIASSIST is designed for silent bugs, but it can also be generalized to other kinds of bugs.

V. EVALUATION

To assess the effectiveness of PYSIASSIST in helping developer recognize silent bugs, we answer the following research questions.

- **RQ1:** What is the effectiveness of PYSIASSIST in recognizing silent bugs in users' programs?
- **RQ2:** How does PYSIASSIST compare with state-of-the-art tools for detecting PyTorch bugs?

A. RQ1: Effectiveness of PYSIASSIST

Experimental Setup and Dataset. We design PYSIASSIST based on the training set, and evaluate it on the testing set. In the testing set, the author responsible for annotating the testing set discovered 14 patterns, while 12 of them overlap with the patterns from the training set. Table I presents the categories of the overlapped patterns between the training and testing dataset. As PYSIASSIST cannot detect bugs beyond the implemented patterns, we only focus on posts that corresponds to the 12 overlapped patterns from the testing set. To evaluate the precision and recall of PYSIASSIST, we extract code snippets from each post p and label them as follows:

1) Due to the code in the post being code snippets rather than executable code, we are unable to verify whether the code in the question and answer contains other minor issues. When labeling the positive and negative samples for the code, we only focus on the original question in the post.
2) If the original question in p contains a code snippet c, it is assumed that c has corresponding silent bugs that need to be detected, then c is considered a positive sample.
3) If an accepted answer from the respondent in p contains a code snippet c_a (usually patched code), c_a is regarded as a negative sample, assuming c_a does not have silent bugs.
4) If the accepted answer from the respondent in p does not have any code snippet but provides clear fix suggestions, and the person asking the question has modified the corresponding source code accordingly, the fixed code c_f is regarded as a negative sample.

Note that, even the accepted answer's code could potentially contain minor errors. To ensure the accepted answers correctly fixed the bugs, we also conduct manual reassessments of the samples. Eventually, we obtain 192 segments of testing code snippets, including 129 positive and 63 negative code snippets.

Results. Table I presents the evaluation results. PYSIASSIST achieves 92.4% precision and 85.3% recall on these 192 samples. The lowest precision of PYSIASSIST is 85.7%, however, as a static analysis tool, PYSIASSIST may produce false positives (e.g., when detecting a pattern that requires runtime information). Among these pattern categories, the lowest recall is found in the "wrong-para" category, with a recall of 66.7%. This is because many PyTorch users tend to design different parameters based on their specific circumstances. To improve accuracy in error reporting, we only check the necessary parameters during the design process, resulting in a decrease in recall. For all other pattern categories, PYSIASSIST achieves $>75\%$ recall. Nevertheless, as we design the detection logic of PYSIASSIST based on real-world scenarios where users encountered silent bugs, the overall precision and recall of PYSIASSIST is relatively high. As an assistant tool that provides development or debugging suggestions to developers, we believe high precision is more important than high recall, as producing too many false positives may introduce noises for developers and then make them to loss trust to the assistant tool. Hence, when developing PYSIASSIST, we implement restrictive bug patterns to reduce false positives. In future, it is worthwhile to design more advanced program analysis techniques to further improve the accuracy. The experimental results, detailed pattern descriptions and corresponding confusion matrix can be found in our open-source repository [22].

TABLE I
PRECISION AND RECALL OF PYSIASSIST IN RECOGNIZING SILENT BUGS IN USER PROGRAMS

Category	missing	misplace	wrong-para	errornance	redundant	Total
#pattern	5	1	1	4	1	12
Precision	86.7%	85.7%	100%	100%	94.4%	92.4%
Recall	89.7%	85.7%	66.7%	88.6%	77.3%	85.3%

B. RQ2: Comparison with existing tools

Experimental Setup. We select two approaches as our baseline: CodeGuru [7, 21] and ChatGPT. CodeGuru is a static analysis tool that can detect bugs in Python code [35]. Recently, it has added the capability to detect issues in PyTorch code [21]. To use CodeGuru, we install the CodeGuru binary from the Amazon's CodeGuru homepage, and run it based on the User Guide [36]. Specifically, we use the Python detector with the machine-learning tag that can detect PyTorch bugs [7]. We choose ChatGPT because recent studies have shown its bug detection capability [37, 38, 39]. We follow the same steps shown in Section II-B to construct prompt. Here we also adopted the method similar to Yang et al. [24] to design and optimize prompts. After three rounds of prompt improvements and optimizations, the final prompt that achieves the best results is: "Please review the following PyTorch code snippet and list any potential issues". We evaluate all tools using the same dataset as RQ1.

Results. Table II shows the comparison results. Although ChatGPT can successfully discover silent bugs in PyTorch programs, its precision and recall are relatively low. In comparison, PYSIASSIST achieves higher precision (92.4% vs 76.1%) and recall (85.3% vs 51.9%). Hence, we believe relying on LLM to solve a specific task (i.e., data filtering) instead of

TABLE II
PRECISION AND RECALL OF PYSIASSIST, CODEGURU AND CHATGPT

Tool	PYSIASSIST	ChatGPT	CodeGuru
Precision	92.4%	76.1%	100%
Recall	85.3%	51.9%	1.5%

the bug detection task, which requires more sophisticated analysis, may be a better choice. Compared to CodeGuru which is designed for detecting general PyTorch bugs instead of silent bugs, PYSIASSIST also achieves much better results. CodeGuru can only detect two silent bugs from our dataset, achieving a recall of 1.5%. Although these two detected bugs are correct (i.e., 100% precision), such a low recall is not acceptable. Overall, PYSIASSIST has improved over 15% in both recall rate and precision rate compared to ChatGPT, and achieves much higher recall than CodeGuru.

VI. IMPLICATIONS AND DISCUSSION

We discuss implications and future directions below:

Tool support for PyTorch programs. Our study reveals the common symptoms and the underlying causes of silent bugs when running DL programs. As most symptoms of silent bugs correspond to abnormal outputs (Section III-A), developers and tool designers should consider *designing more flexible test oracles that automatically detect these abnormal outputs* (e.g., in metrics such as accuracy or loss). Moreover, the taxonomy derived from our study may help developers of PyTorch programs in understanding silent bugs in their code, debugging, and fixing them. As Table 8 shows that "Missing operation" is the most common pattern for silent bugs in PyTorch programs (Section III-C), *future research in automated repair of these bugs can focus on synthesizing the missing operation* for fixing these bugs. To reduce manual efforts in detecting silent bugs in PyTorch programs, we identified 14 common bug patterns and designed PYSIASSIST for automated detection of these patterns (Section IV). As PYSIASSIST does not require executing the program (which may be challenging for testing silent bugs), it takes the first step in *designing practical customized tools for detecting and debugging silent bugs* in PyTorch programs. The current implementation of PYSIASSIST only relies on lightweight pattern-matching techniques with CFG/DFG graphs for checking the bug patterns. Hence, it is worthwhile to explore how to *enhance the detection with more sophisticated program analysis techniques* to improve its precision and recall. Nevertheless, we hope that PYSIASSIST will inspire the future design of practical tools for testing, debugging, and fixing PyTorch silent bugs.

Developing PyTorch programs. Our study provides useful development suggestions for preventing silent bugs in PyTorch programs. We observe that many of these silent bugs occur due to the unfamiliarity of developers with the APIs in PyTorch. To prevent silent bugs caused by *Incorrect Loss Function or Activation Function*, developers should carefully read the documentation of activation functions and loss functions to familiarize themselves with (1) the usage of loss functions, (2) the applicable scenarios for different loss functions and activation functions, and (3) common combinations of loss functions and activation functions. To prevent silent bugs caused by *disruption of computation graph*, developers should be aware of operations that may disrupt dynamic computational graph and avoid making incorrect modifications of the *.data* attribute.

Semi-automated data filtering via LLM. Empirical studies that mine software repositories or forums usually involve manually collecting and filtering a large amount of data. Due to the large number of posts that fulfilled the given search query, reading and manually filtering irrelevant posts are time-consuming. Our study made one of the first attempts to use LLMs to assist the filtering process in an empirical study. The results indicated that while LLM may not accurately identify all the desired posts, they can help researchers in excluding obviously irrelevant posts, thereby reducing the time and effort required for the study. In future, it is worthwhile to *explore whether LLMs can be used for semi-automatic or even fully automatic data filtering for other software engineering tasks*.

Extensible checker for detecting silent bugs. PYSIASSIST has extensibility to facilitate the addition of new patterns. Hence, we designed a simple experiment to verify its extensibility. Specifically, we extend PYSIASSIST to support two more bug patterns. These two new patterns are added by an author that did not develop the tool, simulating an user that want to extend PYSIASSIST. We just change the configuration files when we extend PYSIASSIST. Then we collect 21 code snippets to evaluate the extended PYSIASSIST. On these two new patterns, PYSIASSIST achieves a detection accuracy of 95.2% . So we believe PYSIASSIST can be easily extended to support more bug patterns. The reason we focus on the extensibility of PYSIASSIST is that bugs in deep learning programs, especially silent bugs, exhibit a wide range of symptoms and root causes that do not necessarily have one-to-one correspondences (Section III-B). This further increases the time and effort researchers spent during the analysis phase. In this case, our design aims to encourage researchers to focus their limited time and energy on studying the symptoms in posts, analyzing the causes, and defining new patterns without having to worry about supporting each newly defined pattern to validate it.

VII. THREATS TO VALIDITY

Construct validity We filter and annotate the dataset manually, which may affect the reliability of our final results. To alleviate this threat, three authors independently filter and annotate the dataset. For verifiability, we make our dataset and tool publicly available.

External vaildity. Our research investigates bugs in PyTorch programs. Compared to other DL frameworks (e.g., TensorFlow), PyTorch relies on dynamic computational graph. Hence, it is unclear whether our empirical results can be generalized to other DL frameworks. In future, we will analyze different DL libraries and use the extensibility of PYSIASSIST to empirically validate our analysis results.

Internal vaildity. We conducted our study based on PyTorch forum posts. However, there are many forums, e.g., GitHub

TABLE III
DIFFERENCES WITH RELEVANT WORK

Study	Scope	Framework	Data Source	Tool?
Tambon et al. [1]	Silent bug in DL frameworks	Tensorflow & Keras	1168 issues from GitHub.	no
Zhang et al. [6]	General DL bug	Tensorflow	27,845 stackoverflow posts & 82 GitHub commits	no
Islam et al. [5]	General DL bug	Caffe, Keras, Tensorflow, Theano & PyTorch	2716 stackoverflow posts and 500 Github commits	no
Cao et al. [8]	DL Bugs related to performance	Tensorflow & Keras	18,730 stackoverflow posts	yes(3 patterns)
Ours	Silent bug in the programs	PyTorch	13,241 official PyTorch forum posts	yes(14 patterns)

and Stack Overflow, that could be valuable sources of silent bugs. It is unclear whether our analysis results can be generalized to other forums. We plan to continue exploring these silent bugs in future work.

VIII. RELATED WORK

We discuss the closely related work in understanding and analyzing deep learning bugs and silent bugs. To clearly show the differences, Table III presents the differences between our study and the three most relevant works from four perspectives: related work, research object, corresponding DL framework, and data source.

Silent bug study. Several studies have been conducted on silent bugs [1, 40, 41, 42]. Among them, the most relevant work to ours is the work by Tambon et al. [1], which also studied silent bugs. The key difference as shown in Table III is that they focused on silent bugs within the internal code of DL framework (TensorFlow and Keras). Differently, we focus on silent bugs in user program, which are mainly caused by incorrect API usage of DL framework. Moreover, researchers have also analyzed the silent bugs in traditional programs [40, 41, 42]. Specifically, Vahabzadeh et al. [40] focus on silent bugs in test code, which may lead to problematic code passing the tests. Xu et al. [41] analyzed silent bugs introduced by compilers during the compilation process. Van Der Kouwe et al. [42] evaluated the impact of silent bugs on systems via fault injection and proposed automated technique for detecting and assessing the injected bugs. These studies also reveal that silent bugs are hard to detect and can cause adverse effects.

General DL bug study. With the continuous development of DL technology, an increasing number of researchers have begun to study DL bugs. Among them, some researchers have conducted studies on general bugs in DL programs [4, 5, 6, 43, 44, 45, 46]. For instance, Zhang et al. [6] collected 175 TensorFlow program bugs from StackOverflow and GitHub commits. They analyzed these errors from the two dimensions of symptoms and root causes, and discussed the challenges and strategies for detecting and locating these errors. Islam et al. [5] expanded the scope of Zhang et al. [6]'s study, analyzing 970 errors in DL Programs written in Caffe, Keras, TensorFlow, Theano, or Torch from four dimensions: types, root causes, impacts, and pipeline stages. Furthermore, the fix patterns of these bugs were also studied in their follow-up work [43]. Humbatova et al. [4] established a classification of errors in deep learning systems through manual analysis of 375 DL bugs and interviews with 20 developers. Jia et al. [45] further extended the research scope to the interior of DL libraries, where they analyzed the symptoms, root causes, and locations of 202 bugs within the TensorFlow framework. Different from these studies, we focus on silent bugs, which are much harder to detect and analyze.

Specific DL bug study. In addition to general bugs, many researchers have also studied specific DL bugs [47, 48, 49, 50, 51]. Chen et al. [47] studied bugs caused by deploying deep learning models to mobile devices. Zhang et al. [51] summarized common bugs when training DL models and developed a tool for their detection and repair. Vélez et al. [10], Wan et al. [11] identified performance-related patterns of API misuse in cloud AI services. Wu et al. [48] analyzed bugs related to tensor shape errors in deep learning models and developed corresponding detection and repair mechanisms. Cao et al. [8] studied the bugs related to model performance issues. Different from these studies, we are the first systematic study of silent bugs in PyTorch programs.

DL bug detection. Moreover, advancements have been made in the analysis and testing of DL programs. Lagouvardos et al. [52] developed a static analysis tool to identify bugs in tensor shape mismatches. Wardat et al. [53] proposed a new automated tool localizing bugs in DL program based on dynamic analysis. Zhang et al. [54] conducted a comprehensive review of research related to DL program testing. Although these works may be able to detect some of the silent bugs, PYSIASSIST is the first extensible tool that is specifically designed for detecting silent bugs.

IX. CONCLUSION

In this paper, we conducted the first empirical study to analyze the characteristics of silent bugs in PyTorch programs. From 14,523 posts in the PyTorch forum, we selected 365 silent bug posts and analyzed and annotated them based on symptoms, root causes, and buggy code patterns. In the end, we obtained 8 symptom classifications, 16 root cause classifications, and 14 patterns. Moreover, we developed an extensible static checker, PYSIASSIST, which supports the detection of existing silent bug patterns. The validation outcomes affirmed that our tool attains a precision of 93.6% and a recall rate of 84.8% for the existing silent bug patterns, while it demonstrates an accuracy of 95.5% for the newly introduced silent bug patterns. Although this work is done for PyTorch programs, the methodology can be transferred to the programs with other DL frameworks, which will be an important direction for our future work.

X. ACKNOWLEDGEMENTS

This work was supported by National Natural Science Foundation of China under Grant Nos (62141209, 62202026) , and partly by Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent Processing.

References

[1] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, "Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow," *arXiv preprint arXiv:2112.13314*, 2021.

[2] Keras-Team, "Keras-team/keras: Deep learning for humans." [Online]. Available: https://github.com/keras-team/keras

[3] Tensorflow, "Tensorflow/tensorflow: An open source machine learning framework for everyone." [Online]. Available: https://github.com/tensorflow/tensorflow

[4] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[5] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[6] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[7] AWS, "Codeguru machine learning detectors," https://docs.aws.amazon.com/codeguru/detector-library/python/tags/machine-learning/, 2023, accessed: 2023-05.

[8] J. Cao, B. Chen, C. Sun, L. Hu, S. Wu, and X. Peng, "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*, 2022, pp. 357–369.

[9] Y. Zhao, L. Xiao, X. Wang, L. Sun, B. Chen, Y. Liu, and A. B. Bondi, "How are performance issues caused and resolved?-an empirical study from a design perspective," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 181–192.

[10] T. C. Vélez, R. Khatchadourian, M. Bagherzadeh, and A. Raja, "Challenges in migrating imperative deep learning programs to graph execution: an empirical study," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 469–481.

[11] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are machine learning cloud apis used correctly?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 125–137.

[12] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1159–1170.

[13] Dozed, "Large performance gap between pytorch and keras for imdb sentiment analysis model," Nov 2021. [Online]. Available: https://discuss.pytorch.org/t/large-performance-gap-between-pytorch-and-keras-for-imdb-sentiment-analysis-model/135659

[14] C. Charlemagne, "Accuracy of signal prediction model stuck at 51epochs," Aug 2021. [Online]. Available: https://discuss.pytorch.org/t/accuracy-of-signal-prediction-model-stuck-at-51-after-few-epochs/130082

[15] FaultyBagnose, "Resnet101 sensitive to previous evaluations in train mode?" Aug 2020. [Online]. Available: https://discuss.pytorch.org/t/resnet101-sensitive-to-previous-evaluations-in-train-mode/93015

[16] J. Schwabedal, "Unusual scaling in per-example gradient computation," Feb 2020. [Online]. Available: https://discuss.pytorch.org/t/unusual-scaling-in-per-example-gradient-computation/71145

[17] Pytorch, "Pytorch/pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration." [Online]. Available: https://github.com/pytorch/pytorch

[18] AssemblyAI, "Pytorch vs tensorflow in 2023," https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023, 2023, accessed: 2023-05.

[19] J. Wang, G. Xiao, S. Zhang, H. Lei, Y. Liu, and Y. Sui, "Compatibility issues in deep learning systems: Problems and opportunities," 2023.

[20] PyTorch, "Pytorch forums," https://discuss.pytorch.org, 2023, accessed: 2023-05.

[21] B. Liblit, L. Luo, A. Molina, R. Mukherjee, Z. Patterson, G. Piskachev, M. Schäf, O. Tripp, and W. Visser, "Shifting left for early detection of machine-learning bugs," in *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*. Springer-Verlag, 2023, p. 584–597.

[22] G. R. Link, "Hong-shuo/pysiassist." [Online]. Available: https://github.com/Hong-Shuo/PYSIASSIST

[23] T. Vörös, S. P. Bergeron, and K. Berlin, "Web content filtering through knowledge distillation of large language models," *arXiv preprint arXiv:2305.05027*, 2023.

[24] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," *arXiv preprint arXiv:2309.03409*, 2023.

[25] J. M. Kim, "[question] loss is not decreasing," Apr 2021. [Online]. Available: https://discuss.pytorch.org/t/question-loss-is-not-decreasing/119029

[26] Swd, "Implement a keras model using pytorch doesn't learn," Aug 2020. [Online]. Available: https://discuss.pytorch.org/t/implement-a-keras-model-using-pytorch-doesnt-learn/92181

[27] PyTorch, "Pytorch forums," https://discuss.pytorch.org/t/synchronization-slow-down-caused-by-item-which-is-not-caused-by-data-0/19394, 2023, accessed: 2023-05.

[28] ——, "Pytorch forums," https://discuss.pytorch.org/t/pytorch-model-ported-from-keras-model-not-learning/77192, 2023, accessed: 2023-09.

[29] Rgodbey, "Linear regression model–problems with loss,"

Nov 2020. [Online]. Available: https://discuss.pytorch.org/t/linear-regression-model-problems-with-loss/102879

[30] "Codeql," 2023, accessed: 2023-09. [Online]. Available: https://codeql.github.com/

[31] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[32] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19.

[33] J. Dennis, *Data Flow Graphs*. Boston, MA: Springer US, 2011, pp. 512–518.

[34] L. Li, J. Wang, and H. Quan, "Scalpel: The python static analysis framework," *arXiv preprint arXiv:2202.11840*, 2022.

[35] R. Mukherjee, O. Tripp, B. Liblit, and M. Wilson, "Static Analysis for AWS Best Practices in Python Code," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 14:1–14:28.

[36] N. Quinnin, "Codeguru visual c++ bian cheng jing cui," 2004. [Online]. Available: https://docs.aws.amazon.com/codeguru/

[37] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology Computer Engineering (IJITC) ISSN : 2455-5290*, vol. 3, no. 01, p. 17–22, Jan. 2023.

[38] M. A. Haque and S. Li, "The potential use of chatgpt for debugging and bug fixing," *EAI Endorsed Transactions on AI and Robotics*, vol. 2, May 2023. [Online]. Available: https://publications.eai.eu/index.php/airo/article/view/3276

[39] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," 2023.

[40] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 101–110.

[41] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao, "Silent bugs matter: A study of Compiler-Introduced security bugs," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023, pp. 3655–3672.

[42] E. Van Der Kouwe, C. Giuffrida, and A. S. Tanenbaum, "On the soundness of silence: Investigating silent failures using fault injection experiments," in *2014 Tenth European Dependable Computing Conference*. IEEE, 2014, pp. 118–129.

[43] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1135–1146.

[44] M. Ashraf, S. M. Ahmad, N. A. Ganai, R. A. Shah, M. Zaman, S. A. Khan, and A. A. Shah, "Prediction of cardiovascular disease through cutting-edge deep learning technologies: an empirical study based on tensorflow, pytorch and keras," in *International Conference on Innovative Computing and Communications: Proceedings of ICICC 2020, Volume 1*. Springer, 2021, pp. 239–255.

[45] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *Database Systems for Advanced Applications, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 2020, pp. 604–620.

[46] M. J. Islam, "Towards understanding the challenges faced by machine learning software developers and enabling automated solutions," *master thesis, Iowa State University*, 2020.

[47] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An empirical study on deployment faults of deep learning based mobile applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 674–685.

[48] D. Wu, B. Shen, Y. Chen, H. Jiang, and L. Qiao, "Tensfa: detecting and repairing tensor shape faults in deep learning systems," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 11–21.

[49] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Automated testing of software that uses machine learning apis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 212–224.

[50] H. Y. Jhoo, S. Kim, W. Song, K. Park, D. Lee, and K. Yi, "A static analyzer for detecting tensor shape errors in deep neural network training code," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 337–338.

[51] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.

[52] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in tensorflow programs," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[53] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.

[54] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2020.