Towards Understanding the Challenges of Bug Localization in Deep Learning Systems

**Sigma Jahan · Mehil B. Shah ·
Mohammad Masudur Rahman**

Abstract Software bugs cost the global economy billions of dollars annually and claim ~50% of the programming time from software developers. Locating these bugs is crucial for their resolution but challenging. It is even more challenging in deep-learning systems due to their black-box nature. Bugs in these systems are also hidden not only in the code but also in the models and training data, which might make traditional debugging methods less effective. In this article, we conduct a large-scale empirical study to better understand the challenges of localizing bugs in deep-learning systems. First, we determine the bug localization performance of five existing techniques using 2,365 bugs from deep-learning systems and 2,913 from traditional software. We found these techniques significantly underperform in localizing deep-learning system bugs. Second, we evaluate how different bug types in deep learning systems impact bug localization. We found that the effectiveness of localization techniques varies with bug type due to their unique challenges. For example, tensor bugs were more accessible to locate due to their structural nature, while all techniques struggled with GPU bugs due to their external dependencies. Third, we investigate the impact of bugs' extrinsic nature on localization in deep-learning systems. We found that deep learning bugs are often extrinsic and thus connected to artifacts other than source code (e.g., GPU, training data), contributing to the poor performance of existing localization methods.

Sigma Jahan
Dalhousie University, Canada
E-mail: sigma.jahan@dal.ca

Mehil B. Shah
Dalhousie University, Canada
E-mail: shahmehil@dal.ca

Mohammad Masudur Rahman
Dalhousie University, Canada
E-mail: masud.rahman@dal.ca

1 Introduction

Software bugs are human-made errors in the code that prevent it from working correctly [1]. They are often prevalent in modern software systems and could range from hundreds to thousands in a single system [2]. Due to the bugs in software systems, the global economy loses billions of dollars every year [3, 4]. Developers also spend about 50% of their programming time dealing with software bugs and failures [3]. To correct any bug, the developers first need to identify the location of a bug within a software system, which is known as *bug localization* [5]. According to a recent survey, 49.20% of 327 software practitioners from several major technology companies (e.g., Google, Meta, Amazon, and Microsoft) consider the localization of bugs as one of the most challenging tasks during software development and maintenance [6].

While localizing bugs in traditional software systems (a.k.a, non-deep learning systems) remains a challenge, it could even be more challenging in deep learning systems. Unlike bugs in non-deep learning systems, deep learning-related bugs could be hidden in the source code, training data, trained models, or even deployment scripts [7, 8, 9]. Besides, the use of various deep learning frameworks (e.g., PyTorch, Caffe, and TensorFlow) could make these bugs even more complex [10].

Given the prevalence and costs of software bugs, any automated support to localize the bugs can greatly benefit software practitioners. Over the years, many approaches have been designed to localize bugs in traditional software systems using information retrieval [11, 12, 13, 14], dynamic program analysis [15, 16], and deep learning [17, 18, 19]. However, due to the significant differences between traditional and deep learning bugs, these existing solutions might not be adequate for localizing bugs in deep learning systems.

To date, there exist only a few techniques for detecting bugs in deep learning systems. Most of them concentrate on specific types of bugs (e.g., model bugs, training bugs) without considering the broader spectrum of deep learning systems. Wardat et al. [18] propose a dynamic approach to localize different types of model bugs in the Deep Neural Network (DNN). They identify the faulty layers containing numerical bugs by customizing the Keras' callback function and analyzing the dynamic behaviors of a model. However, their solution focuses on only model bugs from deep learning systems, strongly coupled with the Keras library, and achieves a low accuracy, which presents significant challenges for widespread adoption by the industry. In another study, Wardat et al. [20] propose a heuristic-based approach to diagnose two main categories of bugs – model bugs and training bugs. They also recommend actionable fixes of the bugs based on the diagnosis. Since their approach depends on a set of hard-coded rules, it might be limited in terms of scalability and context-awareness. In a recent work, Cao et al. [21] introduces a technique that

leverages the dynamic properties of a model and an ensemble of three machine learning classifiers (e.g., KNN, Decision Tree, Random Forest) to localize five types of training bugs (e.g., loss, gradient) from deep learning systems. Their technique might also not be able to address a broader array of bug types from deep learning systems, highlighting the issues of scalability. On the other hand, Kim et al. [22] use basic Information Retrieval (IR) algorithms, such as rVSM and BM25, to localize bugs in deep-learning systems. They report poor performance but do not perform any comprehensive analysis to understand the poor performance of IR-based techniques.

Interestingly, at least 30 techniques adopt IR algorithms to locate bugs in traditional software systems due to their computational efficiency and lightweight nature [11, 12, 14, 13, 23, 22]. They were also reported to perform comparably to the complex models (e.g., LDA) [24]. Unlike deep learning-based techniques, IR-based techniques rely on the textual similarity between bug reports and source code as a proxy of suspiciousness, which is simple and explainable. However, IR-based techniques suffer from vocabulary mismatch issues [25] and can only capture linear relationships between two items. On the contrary, deep learning-based techniques can capture the non-linear relationships between two items [26, 27, 28]. Thus, they have the potential to capture more nuances in the relevance between enriched information from the source code and bug reports. However, they also suffer from poor outlier handling, class imbalance problems, and a lack of monitoring [29]. Thus, the potential of existing solutions for localizing bugs in deep learning applications is neither well understood nor well investigated to date. Our work in this article fills in this important gap in the literature.

In this article, we conduct a large-scale empirical study to better understand the challenges of locating bugs in deep learning systems. First, we collect a total of 2,365 bugs from deep-learning systems and 2,913 bugs from traditional software systems (a.k.a non-deep-learning systems), and *empirically* show how existing techniques (e.g., BugLocator [11], BLUiR [12], BLIA [23], DNNLOC [17]), bjXnet [30]) perform in locating bugs from deep learning systems. Our work utilizes these traditional techniques as baselines, adapting their core principles to the specific nuances of deep learning bugs. Second, we categorize our collected bugs based on an existing bug taxonomy [31] and found that certain bugs from deep learning systems (e.g., GPU bugs) are more difficult than others to locate due to their multifaceted heterogeneous dependency issues. Finally, we found that deep learning bugs are connected to artifacts other than source code (e.g., GPU, training data, external dependencies) and are prone to be extrinsic in nature, which might explain the poor performance of existing techniques for these bugs. We thus answer three important research questions in our study as follows.

(a) **RQ$_1$: How effective are the existing approaches in localizing bugs from deep learning systems?**
    We evaluated the performance of five existing approaches (BugLocator [11], BLUiR [12], BLIA [23], DNNLOC [17], and bjXnet [30]) using two datasets

– Denchmark [32] and BuGL [33]. First, we found that their performance measures are poorer (e.g., 31.59% less MAP for BugLocator, 33.25% for BLUiR, 34.14% for BLIA, 31.43% for DNNLOC, 27.87% for bjXnet) in localizing bugs from deep learning systems than that of non-deep learning systems. Our statistical tests (t-test [34], Cohen's D [35]) also report that their performance is significantly lower. Second, we found that localizing bugs from the deep learning frameworks is more challenging than libraries or tools due to the frameworks' inherent complexity. Although our findings reinforce the existing understanding and belief about the challenges of the bugs in deep learning systems [36, 22, 18], we also substantiate them with solid empirical evidence and demonstrate the performance gap of existing solutions in localizing the two categories of bugs.

(b) **RQ$_2$: How do different types of bugs in deep learning systems impact bug localization?**
We use an existing taxonomy [31] of bugs to classify the bugs in deep learning systems and evaluate the performance of five existing techniques for each type of bug. First, we found that 64.80% of the bugs from deep learning systems are related to deep learning (e.g., model, training), whereas the remaining ones are not. Second, we found that bjXnet and DNNLOC demonstrated better results in locating model and tensor bugs, possibly due to their ability to capture comprehensive contextual information specific to these bugs using deep neural networks. We also found that BLUiR performs comparably to DNNLOC for training bugs, which might be attributed to its structured information retrieval. However, all five baseline techniques experienced significant difficulty localizing GPU bugs. Thus, our analysis offers valuable insights regarding the nature of different types of bugs in deep learning systems and highlights the specific strengths and weaknesses of existing techniques, which could be useful to advance debugging support for deep learning systems.

(c) **RQ$_3$: What are the implications of extrinsic bugs in deep learning systems for bug localization?**
Bugs triggered by external entities (e.g., third-party libraries, GPU) are called *extrinsic bugs* [37]. Given the frequent use of deep learning libraries and their external dependencies, the bugs in deep learning systems could be extrinsic [10]. Since the existing techniques mostly focus on intrinsic bugs (i.e., triggered by bug-introducing change), we investigate how they deal with extrinsic bugs from deep learning systems. First, we found deep learning systems have 40.00% extrinsic bugs, which is almost four times higher than that of non-deep learning systems. Second, we found that the localization performance of existing techniques degrades significantly for extrinsic bugs (e.g., 15.20% less MAP for DNNLOC) (Table 30). We also found that deep neural network-based solutions (e.g., DNNLOC) are not particularly helpful for locating extrinsic bugs because they are designed to detect code patterns, not issues from external sources or environments. Finally, we found a significant correlation between the bugs in deep learning

systems and the extrinsic factors using appropriate statistical analysis (Chi-Square test), which delivers valuable insight for designing effective solutions to find bugs in deep learning systems.

2 Background

In this section, we introduce the necessary terminologies and concepts to follow the remainder of the article. We introduce extrinsic bugs, intrinsic bugs, the taxonomy of deep-learning bugs, and the categories of projects in deep-learning systems.

Table 1: Example of deep learning-related and non deep-learning-related extrinsic bugs [38, 39]

Deep Learning-related Extrinsic Bug [38]
Title:
wmt19 model cannot run on GPU except #0.
Description:
I am running the tutorial. I successfully loaded the model from
the hub and tried to run it on the second GPU (id=1).
, Which raised an exception that data and models are stored
on different GPUs. With GPU (id=0) works fine.
Code sample:
import torch
en2de = torch.hub.load('pytorch/fairseq',
'transformer.wmt19.en-de'
checkpoint_file='model1.pt:model2.pt:model3.pt:model4.pt'
tokenizer='moses',bpe='fastbpe').to(torch.device('cuda:1'))
result = en2de.translate(['hello'])
Environment:
fairseq Version==0.9.0, PyTorch Version ==1.4.0,
OS: Ubuntu 18.04,vPython version: 3.6,
CUDA/cuDNN version: 10.2,
GPU models and configuration: RTX 2080 x 2
Non Deep Learning-related Extrinsic Bug [39]
Title:
GitHub CI on Windows is broken.
Description:
Normally, we should skip distributed tests on Windows with
SKIP_DISTRIB_TESTS=1
CI_PYTHON_VERSION="3.7"
sh tests/run_cpu_tests.sh,
but a distributed test was executed:
tests/ignite/contrib/engines/test_common.py
::test_distrib_cpu ERROR [2%]
Related to beta support of distributed on Windows in Pytorch 1.7

2.1 Extrinsic bug

A bug caused by factors external to a software system, such as changes to the operating environment, requirements, or third-party libraries, is known as *extrinsic bug.* Rodriguez-Perez et al. suggests three heuristics based on bug reports to identify extrinsic bugs as follows [37].

(a) Environment: An extrinsic bug is caused by a modification to the environment in which the software system operates. The environment could be an operating system, a physical machine, or even a cloud infrastructure.

(b) Requirement: An extrinsic bug is triggered by a change outside the project's version control system. During software development, if a user requirement changes after implementation, the development team might implement the new requirement without discarding the old feature. The old, unexpected feature will then be considered an extrinsic bug.

(c) Third-party library: The bug found in the project's third-party library is considered an extrinsic bug. For example, if a software project uses a third-party library for processing images for a mobile application, and the app crashes when processing certain image formats due to a bug in that third-party library, that bug will then be considered an extrinsic bug.

2.2 Intrinsic bug

The external factors do not cause an intrinsic bug; rather, it is caused by a bug-introducing change in the version control system [37]. For example, if a messaging application fails to deliver messages due to a logical error in a recent code change, that would be an intrinsic bug.

2.3 Taxonomy of bugs in deep learning systems

Software bugs in deep learning systems can be divided into two categories – DL bug and NDL bug [31].

Deep Learning (DL) bug refers to a software error that is connected to the deep learning module embedded in the software system, causing inaccurate or unexpected output. According to the existing literature [31], DL bugs can be divided into five main categories: Model, Training, Tensor & Input, API, and GPU.

 – **Model bug** is connected to the structure and properties of a deep learning model (Table 23 [40]). An example of a model bug is an incorrect model initialization caused by an input image size mismatch, resulting in inaccurate output in a computer vision application.
 – **Training bug** occurs during the training phase of a deep learning application (Table 24 [41]). For instance, during the training of a deep learning model for object detection, if the loss function is incorrectly defined, the

model will learn to detect objects with very poor accuracy, leading to in-correct output from the system.
- **Tensor & Input bug** (a.k.a tensor bug) occurs due to wrong tensor input or tensor calculation issues (Table 25 [42]). For instance, if the tensor input shape is declared incorrectly, it will lead to output errors.
- **API bug** occurs due to incorrect use of an API in the deep learning software system (Table 26 [43]). For example, an API bug might occur if a developer mistakenly calls the wrong API function from the deep learning framework (e.g., Tensorflow), causing inaccurate results in the output.
- **GPU bug** is connected to the Graphics Processing Unit (GPU) used in the system (Table 27 [44]). For example, if the model's memory requirements exceed the available GPU memory or the GPU is not compatible with the DL framework, then they could lead to errors during model training.

Non-Deep Learning (NDL) bug refers to a software error that is not related to the deep learning module but still leads to unexpected behaviors in deep learning applications. An example of NDL bugs could be a logical error in the source code that leads to a deadlock, making the program being stuck in an infinite loop.

As shown in Table 1, Bug 1860 [38] is a deep learning-related extrinsic bug triggered by the change in the environment. When the WMT19 model runs on multiple GPUs, the execution fails since the same GPU cannot store both the model and data. It is clearly related to the deep learning module. On the other hand, this bug is not related to the Fairseq library (a.k.a., deep learning application); rather, it is related to external factors (e.g., GPU), which indicates its extrinsic nature.

In Table 1, Bug 1426 [39] is another extrinsic bug connected to the Windows OS environment. The bug triggers when the tests from the CI pipeline are distributed over multiple Windows machines. It is clearly not related to deep learning (a.k.a., Non-DL bug), but the triggering factors are outside of the version control system, which indicates an extrinsic nature.

2.4 Categories of projects in deep learning systems

Deep learning-based projects can be categorized into Frameworks, Tools, Libraries, Applications, Engines, Platforms, and Compilers [32]. The core elements—Frameworks, Tools, and Libraries—comprise the majority of relevant bug reports, accounting for 96.6% of the dataset [32].

A **deep learning framework** is a software platform that provides the environment for designing, training, and deploying deep learning models [46]. Examples include TensorFlow[1], PyTorch[2], and Apache MXNet[3]. These frameworks come with pre-defined modules and functions and offer a structured way

---

[1] https://www.tensorflow.org/
[2] https://pytorch.org/
[3] https://mxnet.apache.org/versions/1.9.1/

Table 2: Example of a bug report from deep learning framework [45]

Framework Bug (Bug ID: 10224)
Title
Language model example cannot be run
Description
The language model example cannot be run without manually creating a data folder. There are also inconsistencies between the documentation and the code. Optional argument – data DATA location of the data corpus does not appear in the code train.py ... Detailed BR: https://github.com/apache/mxnet/issues/10224

to implement deep learning architectures using high-level programming inter-
faces [47]. The example bug in Table 2 [45] is characterized by the inability
to run a language model without manually creating a data folder. It repre-
sents a framework bug because it directly impacts the core functionalities of
the framework, specifically the design and training of models. Frameworks are
expected to provide seamless, user-friendly environments for developing deep
learning models, and issues that hinder the ease of use, such as documenta-
tion inaccuracies and additional manual setup steps, indicate problems at the
framework level.

Table 3: Example of a bug from deep learning library [40]

Library Bug (Bug ID: 313)
Title
A bug in GPT2Tokenizer
Description
GPT2Tokenizer fails to recover a sentence \"BART is a seq2seq model.\" with encoded ids of it. The output sentence is \"BART is a seqseq model.\". It should be related to numbers' processing. ... URL: https://github.com/tanyuqian/texar-pytorch /blob/master/examples/bart/gpt2_tokenizer_bug.py Detailed BR:https://github.com/asyml/texar-pytorch/issues/313

A **deep learning library** is a collection of functions that facilitate specific
tasks within deep learning. Libraries like Keras[4] and cuDNN[5] can either be
integrated into frameworks or can operate independently [48]. From Table
3 [40], the bug in the 'GPT2Tokenizer' within the `texar-pytorch` project is
classified as a library bug due to its specific component focus and the nature of
functionality. It uses the TokenizerBase class from the texar library, designed

---

[4] https://keras.io/

[5] https://developer.nvidia.com/blog/tag/cudnn/

to work with the PyTorch framework. The tokenizer's failure to accurately process text data leads to the bug, which indicates a library issue.

Table 4: Example of a bug from deep learning tool [49]

Tool Bug (Bug ID: 5596)
Title
Tensorboard can not load all Hyperparameters keys
Description
Encountered an issue in TensorBoard where it couldn't load all hyperparameter keys when I used writer.add-hparams with different hparam-dict parameters in multiple experiments. This problem made it difficult to track and compare different hyperparameter settings across these experiments, affecting the overall functionality of TensorBoard. I provided code snippets and responses to showcase the inconsistency in the display of hyperparameters in TensorBoard's interface. This issue is significant as it hinders the effective use of TensorBoard for experiment tracking and analysis. ... Detailed BR: https://github.com/tensorflow/tensorboard/issues/5596

Finally, **deep learning tools** refer to utilities that assist with the tasks related to deep learning, such as visualization or model optimization. An example would be TensorBoard[6], which is often used for TensorFlow visualization [50]. Each of the frameworks, libraries, and tools plays unique but complementary roles in the context of deep learning. From Table 4 [49], the bug in TensorBoard qualifies as a tool bug since TensorBoard is a visualization toolkit in the TensorFlow ecosystem. The bug refers to the inability of TensorBoard to load all hyperparameter keys when `writer.add-hparams` is used with varying `hparam-dict` parameters, directly impacting its core functionality as a tool. This feature is essential for monitoring and contrasting various experimental settings of TensorBoard.

3 Study Methodology

Fig. 1 shows the schematic diagram of our conducted study. First, we collect bug reports from two benchmark datasets for two different software systems: deep learning systems [32] and traditional software systems [33]. Then, we contrast the performance of five existing techniques [11, 12, 23, 17] in locating bugs between deep learning systems [32] and traditional software systems [33]. Second, we perform an in-depth analysis to understand the challenges of localizing different deep-learning bugs. Finally, we investigate the influence of extrinsic factors on deep learning bugs and their impact on bug localization. This section discusses the major steps of our study design, which are as follows:

---

[6] https://www.tensorflow.org/tensorboard
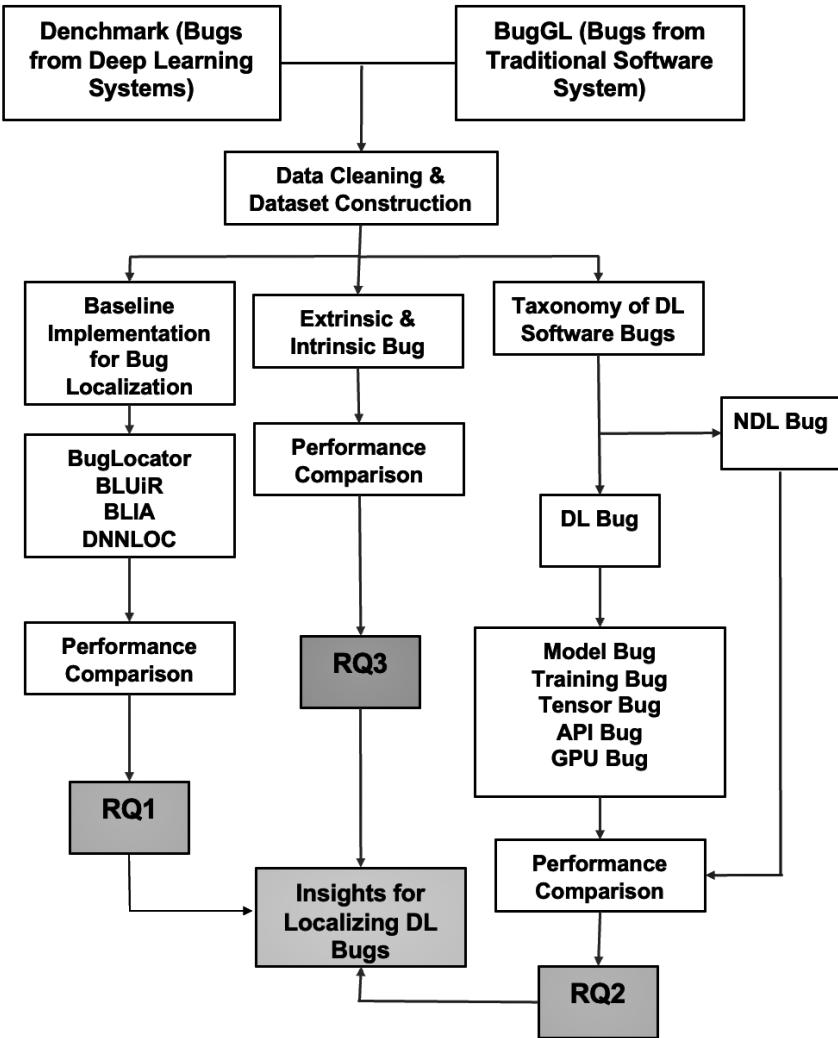
Fig. 1: Schematic diagram of our empirical study

3.1 Construction of dataset

Dataset Collection. In our study, we use two benchmark datasets: BuGL and Denchmark. BuGL contains traditional software bugs, while Denchmark focuses on bugs in deep learning systems. BuGL [33] provides a wide range of traditional software bugs across multiple programming languages (e.g., C++, Java, Python) and includes a total of 2,913 bug reports from 12 Python

projects. In contrast, Denchmark [32], published in the MSR 2021 Dataset Track, focuses on bugs in deep learning systems written in ten different programming languages, including JavaScript, Python, Java, Go, C++, Ruby, TypeScript, PHP, C#, and C. It contains 2,365 bug reports from 136 deep learning-based projects written in Python. While the Denchmark dataset may not cover all possible systems, it is representative of the current state of deep learning system debugging, encompassing a wide range of 136 DL projects (e.g., computer vision, natural language processing, image processing, and visualization).

To ensure a fair comparison between deep learning and traditional software bugs, we only included Python-based projects and bugs from both datasets. Additionally, multiple studies [33, 22] have used Denchmark and BuGL for bug localization experiments, making them suitable benchmarks for our experiments as well.

Table 5: Original [32] and Experimental Distribution of Projects and Bugs

Original Distribution		
Category	**Projects**	**Bugs**
Framework	25	836
Platform	8	150
Engine	4	47
Compiler	2	33
Tool	31	510
Library	44	666
Application	12	124
Total	**136**	**2365**
Experimental Dataset (DL Systems)		
Category	**Projects**	**Bugs**
Framework	17	746
Tool	28	455
Library	41	594
Total	**86**	**1795**

We adopted a set of filtration steps to construct our experimental dataset, which consists of 86 projects from DL systems (shown in Table 5).

- *Original Denchmark Dataset:* The original Denchmark dataset contains 136 projects and 2,365 bug reports. These projects are divided into several categories by the original authors [32]: Frameworks, Tools, Libraries, Applications, Engines, Platforms, and Compilers.
- *Filter 1:* We focused on the core elements of deep learning – Frameworks, Tools, and Libraries, as they represent the majority of relevant bug reports in the dataset (96.6%). In the original dataset, there were 25 projects from Framework, 31 projects from Tool, and 44 projects, which gave us a total of 100 projects. We excluded projects with fewer than ten reported bugs to ensure a robust dataset for meaningful comparisons of bug localization techniques. After applying this filter, we had 18 framework projects, 29 tool projects, and 42 library projects, which led to a total of 87 projects.

– *Filter 2:* We noted that some projects had multiple categories (e.g., a tool that also functions as a library). After carefully reviewing the official documentation and analyzing the role of these projects, we removed overlapping projects to ensure that each project was uniquely classified as either a framework, tool, or library.

After applying these filtering steps, the final experimental dataset consists of 86 projects and 1,795 bug reports - Frameworks: 17 projects, 746 bugs, Tools: 28 projects, 455 bugs, Libraries: 41 projects, 594 bugs.

Data cleaning and pre-processing. After collecting the data from two benchmark datasets, we cleaned and preprocessed them using a set of steps.

Corpus creation. We first download the latest version of the code repositories, ensuring that we have the most up-to-date code for analysis. Next, to accurately link bug reports to their corresponding buggy code, we used the heuristic of Kim et al. [22], focusing on commit messages and bug reports. This involved analyzing commit messages for keywords indicative of bug fixes (e.g., 'fix', 'bug', 'error', 'resolve') and connecting them to corresponding bug reports. To identify the correct bug-fix commit, we cross-referenced these commit messages with bug IDs from the reports, ensuring a precise match. To extract the buggy part from the bug-fix commit, we employed PyDriller, suggested by Kim et al. [22]. We also capture each project's most recently released version as of the bug report's date and collect the appropriate version of the buggy code, especially in cases where the bug reports did not have clear buggy version information.

Query construction. In IR-based bug localization, bug reports are treated as queries that can be executed with a search engine to detect the relevant source documents from the corpus. We construct a repository of bug reports by parsing the original datasets (Denchmark & BuGL) and extracting important information such as bug IDs, descriptions, and timestamps. We construct the queries by extracting tokens from the title and description of bug reports, removing stop words, stemming each word, and splitting the tokens.

Meta data extraction. We also capture the historical context of bugs by extracting their commit history information from the repositories, including commit messages, authors, timestamps, and code changes history. This information is extracted to replicate the existing technique BLIA [23], which provides valuable insights into the evolution of the codebase and the bugs over time.

Ground truth construction. Both benchmark datasets provide the ground truth that contains the correct locations of bugs in the code against the bug reports. To evaluate the performance of the bug localization approaches, we collect ground truth files from both of the original datasets.

To ensure a fair performance comparison of the bug localization techniques between DLSW and NDLSW, we have selected an equal amount of data from both datasets using probability sampling (1795 bug reports from each dataset), which have 95% confidence interval and 5% error margin [51]. We use the principle of randomization for selecting the subsets [52] to avoid any bias. We also

Table 6: Study dataset for bug localization [32, 33]

Data	#P	#BR	#SF		#BF		#Versions	
			Mean	Max	Mean	Max	Mean	Max
Dench.	136	1795	441.60	3,559	2.60	227	6.7	80
BuGL	12	1795	420.50	3,306	2.35	198	6.2	65

P: Projects, **BR**: Bug Reports, **SF**: Source Files, **BF**: Buggy Files

manually analyzed the projects to avoid any overlap between the two datasets. We spent $\approx$5 hours on the manual analysis.

3.2 Replicating of existing techniques for experiments

To answer our first research question, we needed to replicate existing techniques that localize bugs in deep learning systems. We thus select suitable representatives from the existing literature on bug localization. In particular, we choose baseline methods from the frequently used methodologies – Information Retrieval (IR) and Deep Learning (DL).

Over the past decade, Information Retrieval (IR)-based bug localization techniques have gained significant attention, and some of these techniques have been proven effective in localizing bugs from traditional software systems [11, 12, 14, 53, 23]. Our extensive literature review shows that more than 30 peer-reviewed research papers have utilized IR for bug localization and highlighted its benefits [22, 54]. These techniques rely on textual similarities between bug reports and source code to identify relevant files or methods. Our motivation was to explore whether the success of these well-established IR-based methods could extend to DL systems.

While spectrum-based bug localization techniques are widely studied and capable of method-level or line-level localization [55, 56], they might face significant challenges when applied to DL systems. Spectrum-based techniques analyze program traces from passing and failing tests, but DL systems often present complexities such as non-deterministic behavior, silent bugs (bugs that do not cause crashes or generate stack traces) [36, 57], and external dependencies (e.g., GPUs), which make obtaining such traces difficult or computationally prohibitive. In contrast, IR-based techniques do not require execution traces and instead rely on textual similarity between bug reports and code, which can be particularly effective in DL projects where bugs often stem from code, configuration, or data script issues [58]. Although IR techniques localize bugs at a coarser file level, this granularity serves as a practical initial step for large-scale DL projects, narrowing down the search space in extensive codebases and complex systems with multiple components [59]. Given these advantages and the computational efficiency of IR-based approaches, we focused our study on evaluating their applicability in DL systems.

We selected three IR-based techniques as our baselines for bug localization: BugLocator [11], BLUiR [12], and BLIA [23]. These techniques represent critical advancements in the field. BugLocator is a seminal work that first intro-

duced the application of Information Retrieval (IR) through the Vector Space Model (VSM) for bug localization. BugLocator paved the way for subsequent research in IR-based localization by using both source code and historical bug reports. On the other hand, BLUiR is built upon BugLocator by introducing structured information retrieval, which integrates structural elements from both source code and bug reports, marking a significant improvement over earlier methods reliant solely on textual similarity. Thirdly, BLIA extended BLUiR's approach by incorporating additional meta-components such as stack traces, code change histories, and version control information. This combination has shown superior performance over earlier techniques like Locus [60], AmaLgam [14], and BRTracer [53], making BLIA a comprehensive and effective baseline for our study. These three techniques not only represent key advancements in IR-based bug localization but are also computationally efficient, making them feasible for large-scale DL projects where obtaining execution traces is often impractical.

In addition to IR-based techniques, we selected DNNLOC [17], a hybrid approach that combines traditional IR methods with Deep Neural Networks (DNNs). Unlike pure IR-based methods, DNNLOC leverages the ability of deep learning models to capture non-linear and complex relationships between bug reports and source code. This hybrid approach brings the computational efficiency of IR-based methods while enhancing localization through deep learning's ability to uncover hidden patterns in large datasets. Moreover, we included bjXnet [30], a state-of-the-art model that applies an attention mechanism with a code property graph to localize buggy files. This ensures that our study covers a wide range of methodologies, from traditional IR-based approaches to more advanced deep learning methods with attention mechanisms.

We selected these five techniques to ensure a comprehensive evaluation of bug localization methodologies. They represent a spectrum of approaches, ranging from early IR-based methods to cutting-edge deep learning models, each contributing novel ideas to the field. We chose these baselines based on their relevance to our study goals, computational feasibility, representativeness of the existing literature, and the interpretability of their results. Even DNNLOC, despite incorporating a deep learning component, maintains computational efficiency due to its hybrid nature. The explainability of IR-based approaches offers another significant advantage, as developers can easily interpret the results based on textual similarities—a crucial factor when debugging complex DL systems. The details of each baseline are as follows.

BugLocator [11] uses rVSM (revised Vector Space Model) that takes the document length into consideration to optimize the classic VSM model for bug localization to detect relevant source code documents against a bug report. It also calculates the SimiScore, which is a measure of similarity between a newly reported bug and previously fixed bugs based on their bug reports. SimiScore is combined with rVSM to calculate the final relevancy score. The relevant source code files are then ranked based on their combined scores, and

the top-K documents are marked as buggy. Many subsequent IR-based techniques [14, 13, 23] adopted this method due to its simplicity and explainability. Hence, we chose this method as our first baseline.

BLUiR [12] uses AST parsing to extract four items: class, method, variable, and comment – from each source code document. It also captures two fields from each of the bug reports (summary & description). Then, a total of eight separate similarities are calculated between these two sets using the BM25 algorithm [12]. Then, these document scores are summed as the suspiciousness score to rank buggy files against a given bug report. BLUiR technique is the first one that leverages structured elements from both source code and bug reports to localize bugs using IR. We thus choose this as another baseline technique for our study.

BLIA [23] integrates several items such as textual similarity between bug reports and source documents [11], code structures [12], version control history [14], stack trace analysis [53], and code change analysis in the IR-based bug localization. While bug reports and source code are useful, code change history can also assist in bug localization by providing the changes likely to induce a bug. BLIA has outperformed several previous techniques: BugLocator [11], BLUiR [12], Amalgam [14], BRTracer [53], which makes it suitable as the third baseline technique for our study.

IR-based localization can be adapted to different granularity levels (e.g., method, file). We chose file-level granularity since each of the selected baselines frequently used this granularity.

DNNLOC [17] uses a hybrid method that incorporates both rVSM [11] from IR and Deep Neural Networks (DNNs) from DL. DNNs establish associations between specific terms in bug reports and the corresponding code tokens and terms in the source files. While DNNs alone do not achieve high accuracy due to dimensionality reduction in the projection process as they lose information, the integration with rVSM enhances their capability to correlate bug reports with relevant buggy files. Buggy source documents may not share textual similarities with bug reports, where IR-based techniques struggle. Thus, to address the challenges of IR-based techniques (e.g., lexical mismatch problem [25]) and to include non-linearity in relevance estimation, we chose DNNLOC as the final baseline for our study.

bjXnet [30] employs a multimodal representation learning framework that integrates the Code Property Graph (CPG) [61] and attention mechanism [62] to improve bug localization. CPG provides a rich, structured representation of the source code by incorporating different aspects such as abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs). This comprehensive representation captures both the syntactic and semantic properties of the code, facilitating a deeper understanding that is crucial for identifying buggy parts. The attention mechanism in bjXnet allows the model to focus

on the most critical components within the CPG, effectively highlighting important patterns and dependencies that are likely associated with bugs. By weighting specific nodes or edges more heavily, bjXnet can localize bugs more accurately than methods relying solely on textual similarity. bjXnet has outperformed several earlier techniques - such as BugLocator [11], DeepLocator [19], DeepLoc [63], and DreamLoc [64]. Given its cutting-edge methodology and empirical success, we selected bjXnet for our empirical study.

Since the original authors' replication packages were unavailable, we used the publicly available versions to replicate BugLocator [11], BLUiR [12], and DNNLOC [17, 65] that have been validated in peer-reviewed papers [54]. We also carefully adapted the BLIA [23] and bjXnet [30] from their original replication packages to our datasets. Our replication package is provided for further experimental details [66].

We validated our replication of the baseline techniques by conducting experiments on a common subject system – SWT. All baseline techniques [11, 12, 23, 17, 30] utilized the SWT dataset to assess performance in their original work, which makes it a viable choice for our validation.

Model	MAP (Original)	MAP (Replication)	Difference (%)
BugLocator	0.45	0.44	2.22
BLUiR	0.58	0.57	1.72
BLIA	0.65	0.63	3.08
DNNLoc	0.37	0.36	2.70
bjXnet	0.55	0.53	3.64

Table 7: Comparison of Original and Replicated Techniques in Terms of MAP Performance on the SWT Dataset

From Table 7, we notice that the MAP differences range from 1.72% to 3.64% and fall within the margin of error, confirming the reliability of our replications. We also conducted a paired t-test, which resulted in a p-value of *0.12*, showing the differences are statistically insignificant and validating the accuracy of our replication process.

Our study focuses on localizing bugs in deep learning systems, which are predominantly Python-based. To maintain consistency and eliminate confounding factors in our comparative analysis, we also examine traditional software systems written in Python. Although the baseline techniques [11, 12, 23, 17, 30] were initially developed for Java projects, recent empirical evidence suggests that their core principles are largely language-agnostic [67]. However, none of these techniques rely on language-specific features for bug localization. Our primary objective is to evaluate the effectiveness of existing bug localization techniques in deep learning systems. The results we obtained from traditional Python projects (NDLSW) are comparable to those reported for Java projects when these techniques were first introduced (Table 8). This comparability suggests that the programming language may not significantly influence the validity or applicability of our findings. By focusing on Python,

we address the predominant language in deep learning while ensuring our results remain relevant to a broader context of software engineering.

3.3 Performance Evaluation

We use three performance metrics for our study — Top-K accuracy (Top@K), Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR). These metrics have been frequently used by the relevant literature [11, 12, 14, 23, 17, 53].

3.3.1 Top@K

Top-K accuracy (Top@K) measures the percentage of bug reports for each of which at least one of the buggy files was present in the top-k retrieved files. We have used K= 1, 5, 10 for this study.

3.3.2 Mean Average Precision

Precision@K measures the precision of each buggy source document's occurrence within a ranked list. Average Precision@K (AP) computes the average precision for all buggy documents within the ranked list against a search query (a.k.a. bug report). Mean Average Precision (MAP) is the average AP@K value across all queries in a system.

$$\text{AP} = \frac{1}{D} \sum_{k=1}^{D} P_k \times \text{buggy}(k) \tag{1}$$

$$\text{MAP} = \frac{1}{|Q|} \sum_{q \in Q} \text{AP}(q) \tag{2}$$

Here, AP represents the Average Precision, and D refers to the number of total results for a query. k represents the position in the ranked list, P_k denotes the precision calculated at the k-th position and buggy(k) determines whether the k-th result in the ranked list is buggy or not.

3.3.3 Mean Reciprocal Rank

Mean reciprocal rank (MRR) calculates the average of the reciprocal ranks for a set of queries.

$$\text{MRR(Q)} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{firstRank}(q)} \tag{3}$$

where $\text{MRR}(Q)$ represents the Mean Reciprocal Rank for a set of queries Q, $|Q|$ represents the total number of queries in the set Q, $q \in Q$ represents each query in the set Q, firstRank(q) represents the rank of the first correctly retrieved buggy document for the query q.
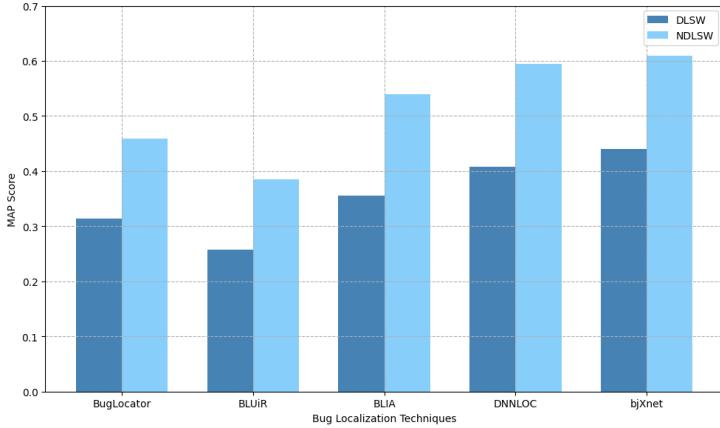
Fig. 2: Performance comparison of existing approaches (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) between Deep Learning Systems (DLSW) and Non-Deep Learning Systems) (NDLSW)

4 Study Finding

4.1 *Answering RQ_1: How effective are the existing approaches in localizing bugs from deep learning systems?*

4.1.1 *Performance comparison between DLSW and NDLSW*

Table 8 compares the performance of our baseline techniques in bug localization between deep learning systems and non-deep learning systems (a.k.a traditional software systems). We used three different evaluation metrics – Top@k, MRR, and MAP, for our comparative analysis.

Our results reveal notable performance differences between the two types of systems across all five methods. Specifically, for BugLocator, the differences in MAP and MRR are *31.59%* and *29.46%*, respectively. BLUiR exhibits differences of *33.25%* in MAP and *30.24%* in MRR. BLIA shows even larger difference, with MAP and MRR differences of *34.14%* and *30.77%*, respectively. DNNLOC, while improving overall performance, still presents significant gaps with differences of *31.43%* in MAP and *26.25%* in MRR. Notably, bjXnet, despite being a state-of-the-art technique that leverages advanced features like Code Property Graphs (CPG) and attention mechanisms, also demonstrates a substantial performance gap. The differences in MAP and MRR for bjXnet are *27.87%* and *24.24%*, respectively, which is consistent with the performance gaps observed in other techniques included in our study.

We calculated these performance differences using the *PerformanceDiff* metric from Wattanakriengkrai et al. [68]. Fig. 2 visualizes the MAP measures, and the differences are clearly visible. Overall, the results show that all five

Table 8: Performance of existing approaches (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) in bug localization

Method	Top@1	Top@5	Top@10	MRR	MAP
DLSW					
BugLocator	0.344	0.547	0.615	0.371	0.314
BLUiR	0.201	0.472	0.585	0.316	0.257
BLIA	0.411	0.609	0.719	0.423	0.355
DNNLOC	0.468	0.682	0.786	0.455	0.408
bjXnet	0.500	0.700	0.800	0.500	0.440
NDLSW					
BugLocator	0.419	0.671	0.794	0.526	0.459
BLUiR	0.311	0.575	0.686	0.453	0.385
BLIA	0.512	0.716	0.820	0.611	0.539
DNNLOC	0.617	0.786	0.855	0.617	0.595
bjXnet	0.650	0.800	0.880	0.660	0.610

DLSW= Deep Learning Systems, **NDLSW**=Non-Deep Learning Systems

Table 9: Statistical tests for the performance gap of existing approaches between deep learning systems (DLSW) and non-deep learning systems (NDLSW)

Method	Metric	Sig. (p-val)	Effect Size
BugLocator	RR	0.00008721**	Medium (0.3834)
	AP	0.00004211**	Medium (0.4023)
BLUiR	RR	0.00395*	Medium (0.2709)
	AP	0.00339*	Medium (0.3154)
BLIA	RR	0.00000734***	Large (0.5012)
	AP	0.00000819***	Large (0.4896)
DNNLOC	RR	0.00000345***	Large (0.5237)
	AP	0.00000389***	Large (0.5110)
bjXnet	RR	0.00000250***	Large (0.5400)
	AP	0.00000275***	Large (0.5280)
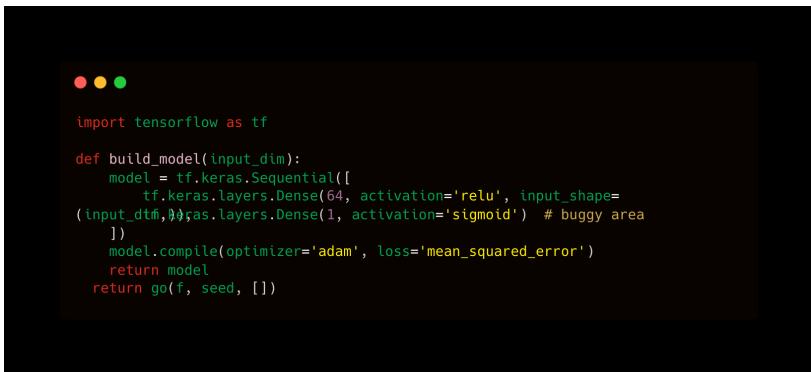
**Sig.**: Significance, **p-val**: p-value, **RR**: Reciprocal Rank, **AP**: Average Precision, ***: Large, **: Medium, *: Small

approaches perform lower when localizing bugs in deep learning systems, and the trend is consistent across all three metrics.

We also perform statistical tests to determine the significance of the performance gap between the two types of systems – deep learning systems and non-deep learning systems (Table 9). We took the Reciprocal Rank (RR) and Average Precision (AP) results of all the samples for each of the five approaches. Then, we performed *Shapiro-Wilk normality test* [69], which reported normal distribution for those metrics. Then, we used appropriate significance and effect size tests to compare the result values from the two types of systems. For the normal distribution, we used t-test as the parametric test [34]. In all significance tests, the p-values were less than the threshold (0.05) for each of the five approaches. Thus, the null hypothesis can be rejected for all comparisons. In other words, the performances of all baseline techniques significantly differ between the two types of systems.

While the significance of a result indicates how probable it is that it is due to chance, the effect size indicates the extent of the difference [70]. Hence, we performed the Cohen's D effect size test [35], and our analysis found a *medium to large* effect size for all cases (Table 9). Thus, our results from effect size tests reinforce the above finding from significance tests. Even the state-of-the-art technique bjXnet shows the gap is statistically significant with a large effect size. This indicates that the disparity is not only statistically meaningful but also practically significant, underscoring the challenges of localizing bugs in DL systems.

To better understand the challenges of bug localization in deep learning systems, we performed a deeper analysis and selected bjXnet, a state-of-the-art technique. Although it has outperformed other baselines, its performance remains inadequate for deep learning systems. Our analysis identified several key factors contributing to bjXnet's reduced effectiveness in deep learning systems. By examining a random subset of bugs (100 bugs) from the Denchmark dataset, we found that DL systems heavily rely on high-level abstractions provided by frameworks like TensorFlow and PyTorch.


```python
import tensorflow as tf

def build_model(input_dim):
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, activation='relu', input_shape=
(input_dtm,)),as.layers.Dense(1, activation='sigmoid')  # buggy area
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
  return go(f, seed, [])
```

Fig. 3: A code snippet from our DLSW dataset showing error in activation function

These abstractions include predefined components such as neural network layers (e.g., `Dense`, `Conv2D`), activation functions (e.g., `relu`, `sigmoid`), and optimizers (e.g., `Adam`, `SGD`). They encapsulate complex mathematical operations—like matrix multiplications and gradient computations—into single library calls. This encapsulation means that essential computational logic is hidden within these library functions rather than being explicitly defined in the source code, which might lead to bjXnet's low performance because the technique relies on analyzing source code to trace errors. In 87% of the cases we examined, core functionalities such as model training, gradient calculations, and loss computations were implemented using these high-level library

calls. For example, Fig. 3 shows a bug that involves the incorrect selection of activation functions in a dense layer—using `'sigmoid'` in regression tasks.

Moreover, DL systems build computation graphs where each layer and operation represents a node. In frameworks like TensorFlow, the computation graph is typically static, defined before runtime. However, in newer versions (e.g., TensorFlow[7] 2+ with eager execution) and other frameworks like PyTorch, computation graphs can evolve dynamically. This dynamic behavior contrasts with traditional systems, making it difficult for static analysis tools like bjXnet to trace errors. These tools struggle to handle the dynamic flow of data and operations, especially when key logic is abstracted in high-level library functions. Other IR-based baselines like BLUiR also have this limitation where only static code analysis is inadequate for DL systems.

In short, even though our findings above match natural intuition, we performed extensive experiments using five different baselines, which resulted in strong empirical evidence. Thus, not only do our findings reinforce the existing understanding and belief about the challenges of the bugs in deep learning systems, but also they substantiate them with solid empirical evidence and demonstrate the performance gap of existing solutions in localizing the two categories of bugs.

Potential Confounding Factors: To ensure fairness in comparison and highlight potential confounding factors, such as code scale, code complexity, and fault representativeness, which could influence the effectiveness of bug localization techniques, we conducted additional analyses as follows.

1. Code Scale: We analyzed the Lines of Code (LOC) from both datasets to investigate whether code scale biases our results. We found that the average LOC per file is higher for NDLSW (325.32) compared to DLSW (280), and the median LOC per file is also higher for NDLSW (244) than for DLSW (210). Such differences in LOC might pose a confounding factor in our experiment. To mitigate this, we repeated a part of our experiment using source documents with similar LOC.

We defined a consistent LOC threshold using the median values from both datasets (NDLSW and DLSW). The median was chosen as a robust central measure to handle the skewed distribution, a standard practice in similar studies [71]. We averaged the medians (244 and 210 LOC, respectively) to get a common median of 227 LOC, then applied a 25% margin (upper bound: 284 LOC, lower bound: 170 LOC) to define our LOC range. This thresholding method follows established guidelines for managing skewed data distributions [71]. We selected 439 files from NDLSW and 618 files from DLSW within the defined LOC threshold. Then, we randomly sampled 350 files from each dataset (95% confidence interval, 5% margin of error) to measure bug localization performance while controlling for code scale.

---

[7] https://www.tensorflow.org/api

Table 10: Performance of bug localization approaches on the subsample of 350 files

Method	Top@1	Top@5	Top@10	MRR	MAP
DLSW Subset					
BugLocator	0.325	0.523	0.600	0.355	0.300
BLUiR	0.180	0.450	0.550	0.290	0.240
BLIA	0.390	0.590	0.695	0.410	0.340
DNNLOC	0.450	0.670	0.770	0.440	0.390
bjXnet	0.470	0.690	0.780	0.460	0.410
NDLSW Subset					
BugLocator	0.400	0.650	0.770	0.500	0.450
BLUiR	0.285	0.550	0.670	0.420	0.350
BLIA	0.470	0.700	0.800	0.570	0.510
DNNLOC	0.580	0.750	0.830	0.580	0.560
bjXnet	0.600	0.770	0.850	0.600	0.580

Table 11: Statistical tests for the performance gap of existing approaches between DLSW subset and NDLSW subset

Method	Metric	Sig. (p-val)	Effect Size
BugLocator	RR	0.00009123**	Medium (0.3652)
BugLocator	AP	0.00005467**	Medium (0.3824)
BLUiR	RR	0.00412*	Small (0.2551)
BLUiR	AP	0.00374*	Small (0.2992)
BLIA	RR	0.00000912***	Large (0.4850)
BLIA	AP	0.00000899***	Large (0.4723)
DNNLOC	RR	0.00000381***	Large (0.5101)
DNNLOC	AP	0.00000419***	Large (0.4956)
bjXnet	RR	0.00000280***	Large (0.5250)
bjXnet	AP	0.00000290***	Large (0.5120)

Sig.: Significance, **p-val**: p-value, **RR**: Reciprocal Rank, **AP**: Average Precision, *******: Large, ******: Medium, *****: Small

We evaluated the performance of our bug localization techniques on these selected subsets. The results are presented in Table 10. The performance trends in the deep learning subset are similar to those observed in the full deep learning dataset. DNNLOC and bjXnet perform best across all metrics, followed by BLIA, BugLocator, and BLUiR. Although the overall performance is slightly lower due to the smaller sample size, our overall findings remain consistent. In the non-deep learning subset, all methods achieve higher scores compared to the deep learning subset, with DNNLOC and bjXnet performing significantly better. We also conducted statistical tests (refer to Table 11) to determine if the performance differences between DLSW and NDLSW are significant when the code scale confounding factor is mitigated. The p-values indicate that all methods' performance differences between DLSW and NDLSW are statistically significant, with medium to large effect sizes.

To summarize, we controlled for differences in LOC between the two datasets. After re-running our experiments using comparable file sizes, the performance trends remained consistent, and the results showed that the differences be-

Table 12: Code Complexity Metrics for deep learning systems (DLSW) and non-deep learning systems (NDLSW)

Dataset	Median LOC	AVG LOC	AVG CC	AVG Halstead	AVG MI
BuGL (NDLSW)	244	325.32	8.72	1456.23	75.30
Denchmark (DLSW)	210	280	9.5	1482.30	68.44

LOC: Lines of Code, **CC**: Cyclomatic Complexity, **MI**: Maintainability Index

tween DLSW and NDLSW are still statistically significant. This indicates that our findings are not influenced by code scale variations.

2. Code Complexity: We also considered code complexity as a potential confounding factor. To compare the code complexity between DLSW (Denchmark) and NDLSW (BuGL), we used established software metrics: Cyclomatic Complexity (CC) [72], Halstead Complexity [73], and the Maintainability Index (MI) [74].

Cyclomatic Complexity measures the number of linearly independent paths through a program's source code, providing insight into control flow complexity. Halstead Complexity assesses cognitive complexity based on the number of operators and operands, reflecting the effort required to understand the code. The Maintainability Index evaluates code maintainability by considering factors like LOC, CC, and Halstead metrics.

Our findings, summarized in Table 12, reveal that the average Cyclomatic Complexity per file is slightly higher for DLSW (9.5) compared to NDLSW (8.72). Similarly, the average Halstead Complexity is comparable between the two datasets (DLSW: 1482.30, NDLSW: 1456.23), and the Maintainability Index is slightly higher for NDLSW (75.30) than DLSW (68.44). We conducted statistical tests (e.g., independent two-sample t-tests) based on the sample size and the distribution of the data for each of the complexity metrics.

The t-test results are as follows: Cyclomatic Complexity: $t(698) = 1.45$, $p = 0.15$; Halstead Complexity: $t(698) = 0.78$, $p = 0.44$; Maintainability Index: $t(698) = -1.32$, $p = 0.19$. These p-values are all above 0.05, indicating that the differences in code complexity metrics between DLSW and NDLSW are not statistically significant.
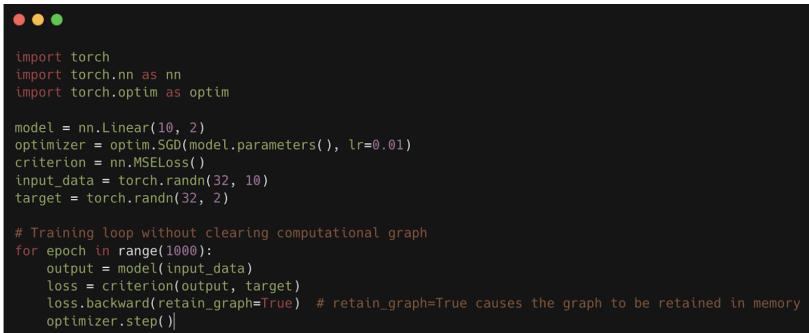
Our analysis shows that both BuGL and Denchmark exhibit similar levels of code complexity. While their structures differ—BuGL being more modular and Denchmark relying on computational tasks—the statistical analysis of Cyclomatic Complexity, Halstead Complexity, and Maintainability Index indicates that both datasets are comparable in terms of code complexity.

3. Fault Representation: We also considered fault representativeness in our extended analysis. To address this concern, we focused on API bugs, which are common in both DLSW and NDLSW. API bugs, such as incorrect interactions with external libraries or frameworks, are prevalent in both deep learning systems (e.g., TensorFlow, PyTorch) and traditional software systems (e.g., Matplotlib, Requests). We identified 195 API bugs in DLSW and 102 in

Table 13: API Bug Localization Performance (Subset of 100 Bugs) for deep learning systems (DLSW) and non-deep learning systems (NDLSW)

Method	MRR	MAP
DLSW Subset		
BugLocator	0.365	0.430
BLUiR	0.410	0.270
BLIA	0.430	0.400
DNNLOC	0.510	0.480
bjXnet	0.530	0.500
NDLSW Subset		
BugLocator	0.470	0.520
BLUiR	0.460	0.350
BLIA	0.520	0.490
DNNLOC	0.590	0.580
bjXnet	0.610	0.600

MRR: Mean Reciprocal Rank, **MAP**: Mean Average Precision

```
import torch
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(10, 2)
optimizer = optim.SGD(model.parameters(), lr=0.01)
criterion = nn.MSELoss()
input_data = torch.randn(32, 10)
target = torch.randn(32, 2)

# Training loop without clearing computational graph
for epoch in range(1000):
    output = model(input_data)
    loss = criterion(output, target)
    loss.backward(retain_graph=True)  # retain_graph=True causes the graph to be retained in memory
    optimizer.step()
```

Fig. 4: A code snippet from the DLSW dataset showing a PyTorch API misuse leading to a memory leak

NDLSW through keyword searches and manual analysis by two authors. We selected 100 API bugs from each dataset and evaluated the bug localization performance of our techniques. Despite mitigating the confounding factor by focusing on API bugs, bug localization remains more challenging in DL systems. From Fig. 4, in PyTorch, missing 'optimizer.zero_grad()' causes a memory leak, leading to out-of-memory errors after prolonged training, making the bug harder to locate using the baselines.

Our comprehensive analysis addresses potential confounding factors such as code scale, code complexity, and fault representation. Even after controlling for these factors, our findings show that bug localization remains significantly more challenging in deep learning systems compared to traditional software systems. This suggests that, despite the presence of similar bug types (e.g., API misuse), bugs in deep learning systems often manifest in more complex and indirect ways. As a result, traditional methods may be insufficient for effectively localizing bugs in deep learning systems, highlighting the need for

new approaches tailored to their unique characteristics.

4.1.2 *Comparison among the categories of deep learning systems*

The original Denchmark dataset comprises 136 projects and 2,365 bug reports, categorized into Frameworks, Tools, Libraries, Applications, Engines, Platforms, and Compilers [32]. Our focus centered on the core components of deep learning systems—Frameworks, Tools, and Libraries—as these are essential and represent the majority of relevant bug reports. We aimed to examine whether there are differences in bug localization performance across these system categories, and we classified the dataset categories as follows.

Filter 1: From the original dataset, we selected 25 Frameworks, 31 Tools, and 44 Libraries, totaling 100 projects. To ensure each project provided sufficient data for robust analysis, we excluded projects with fewer than ten reported bugs. This criterion was crucial for enabling meaningful comparisons of bug localization techniques across projects with substantial debugging histories. After applying this filter, we retained 18 Frameworks, 29 Tools, and 42 Libraries, resulting in 89 projects for further analysis.

Filter 2: We identified overlaps where projects belonged to multiple categories (e.g., a tool functioning as a library). To resolve this, we meticulously reviewed official documentation and the functional roles of these projects, assigning each to a unique category. This refinement led to a final dataset of 86 projects and 1,795 bug reports, distributed as follows: 17 Frameworks (746 bugs), 28 Tools (455 bugs), and 41 Libraries (594 bugs).

To better understand the challenges in localizing deep learning bugs, we analyzed the performance of our baseline techniques across different categories of bugs in DL systems. We employed stratified random sampling to select a balanced number of bug reports from each category for performance comparison, ensuring fair representation and reducing selection bias. We repeated the process three times with different samples to ensure robustness. The results were averaged across these evaluations, providing reliable insights into the efficacy of the bug localization methods.

According to our experimental results in Table 14, existing techniques perform higher in localizing bugs from deep learning libraries than that of frameworks and tools. This could be attributed to the modular design of libraries, aimed at handling specific tasks independently [48]. In the example bug report (Table 15 [75]), the autograd module of the Torch.Library[8] plays a crucial role in calculating gradient across all tensor operations, whereas autograd.gradcheck[9] is a utility function for verifying the accuracy of these computed gradients. Here, a specific error concerning gradient computation in gradcheck was reported during sparse matrix multiplication. Such problems can be localized more efficiently when occurring within a clearly defined module like autograd.gradcheck.

---

[8] `https://pytorch.org/docs/stable/library.html`
[9] `https://pytorch.org/docs/stable/generated/torch.autograd.gradcheck.html`

Table 14: Comparative analysis of bug localization techniques across various deep learning project classes

Project Class	Method	MRR	MAP
Framework	BugLocator	0.404	0.334
	BLUIR	0.205	0.155
	BLIA	0.466	0.392
	DNNLOC	0.493	0.412
	bjXnet	0.560	0.510
Library	BugLocator	0.559	0.453
	BLUIR	0.632	0.539
	BLIA	0.583	0.486
	DNNLOC	0.656	0.597
	bjXnet	0.685	0.640
Tool	BugLocator	0.499	0.407
	BLUIR	0.469	0.387
	BLIA	0.546	0.449
	DNNLOC	0.592	0.514
	bjXnet	0.605	0.550
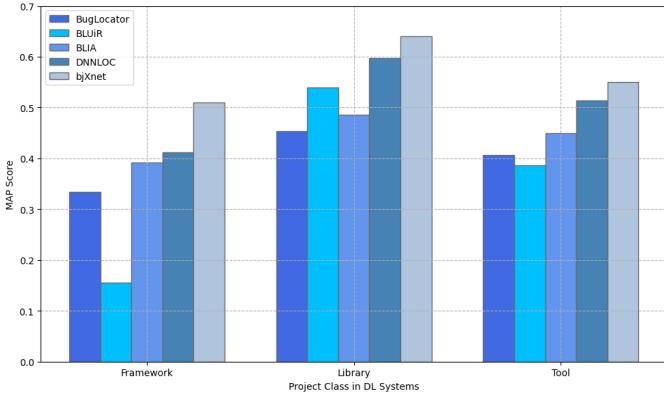
Fig. 5: Performance comparison of existing approaches (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) for various deep learning project classes

Both DNNLOC and bjXnet retrieved the buggy file at Top@1 by leveraging this modularity. DNNLOC's success in accurately locating bugs can be linked to its ability to capture complex patterns using deep learning, which works effectively with the modular design of library bugs. On the other hand, bjXnet uses attention mechanisms to focus on critical functions or variables, like autograd.gradcheck, which might lead to more precise localization and demonstrate the strength of attention-based techniques in isolating and targeting key components in modular architectures.

In contrast, we notice from Fig.5 that the performance of bug localization techniques dropped for framework bugs, especially when using the BLUiR method. Frameworks consist of a broad architecture that guides the flow of control, involving multiple layers and components that interact in complex

Table 15: Example of a bug from deep learning library for bug localization [75]

Library Bug (Bug ID: 87085)
Title
gradcheck failure with sparse matrix multiplication
Description
Sparse @ dense matrix multiplication fails to pass gradcheck. A manual inspection of the gradient seems to indicate that this is a bug with gradcheck rather than the matrix multiplication itself. Steps to reproduce: ... Detailed BR: https://github.com/pytorch/pytorch/issues/87085

ways [46, 10]. The example bug in Table 15 [76] deals with CTC Loss in a Keras model with LSTM on a TensorFlow Processing Unit (TPU), highlighting a framework-level issue. For this example bug, DNNLOC retrieved the correct buggy file at the top 38 position, while BugLocator retrieved it at the top 65 position, indicating limitations in relying only on textual similarities between framework code and bug reports. BLUiR, emerging as the least effective, ranked the correct file at the top 93 positions, highlighting its insufficiency in localizing framework-level bugs with only structural code analysis. bjXnet retrieved the correct buggy file at the top 28 position. All baseline techniques struggle with framework bugs due to the complex, multi-layered architecture of frameworks, which lacks the clear modular boundaries found in libraries. Even DNNLOC and bjXnet face significant challenges because the complex dependencies and interactions across multiple components in frameworks make it harder to trace and isolate the exact location of faults.

Table 16: Example of a bug from deep learning framework for bug localization [76]

Framework Bug (Bug ID: 61297)
Title
CTC Loss errors on TPU
Description
The Keras Model with LSTM+ CTC loss runs normally on GPU, but VM TPU prints grappler errors. The errors usually don't stop the execution, but the loss is nan. It sometimes also crashes with core dump, but it's not consistent. I created a public Kaggle notebook with the code producing the issue: https://www.kaggle.com/code/shaironen/ctc-example/notebook The grappler errors are: .... In addition, I read it's recommended to use model.compile(jit-compile=True) while on GPU to pre-diagnose TPU issues. It gives similar errors and terminates. (with jit-compile=False, it runs normally on GPU only). According to tf.nn.ctc-loss documentation, it should work on tpu. Detailed BR: https://github.com/tensorflow/tensorflow/issues/61297

Lastly, the performance in bug localization slightly improved for tools, but it was not as good as with libraries. This improvement could be attributed to the fact that tools (e.g., TensorBoard) are more independent and have fewer dependencies than the entire framework (e.g., TensorFlow). While tools like TensorBoard focus on specialized tasks such as visualization, they often integrate with complex frameworks like TensorFlow. The example bug in Table 17 refers to an incorrect rendering of floating point metrics in the TensorBoard HParams dashboard. It can be classified as a tool bug, as TensorBoard is a visualization utility within the TensorFlow ecosystem. In this instance, bjXnet retrieves the correct buggy file at the top 25 position. DNNLOC identified the correct buggy file at the top 27 position, BugLocator located the correct file at the 40th position, and BLUiR identified the correct file at the top 52 position. These results suggest that while tools are more independent, their connection to complex frameworks still creates challenges. bjXnet and DNNLOC perform slightly better by understanding the structure more deeply, but all techniques find it difficult to handle the interaction between tools and frameworks.

Table 17: Example of a bug from deep learning tool for bug localization [77]

Tool Bug (Bug ID: 5948)
Title
TB.dev HParams dashboard shows floating point metrics incorrectly
Description
TensorBoard renders incorrectly locally when uploading an experiment to tensorboard.dev some of the floating point values are mangled, and zero values are shown as +/-Infinity .... Detailed BR: https://github.com/tensorflow/tensorboard/issues/5948

In short, our findings indicate that the modular design of libraries [50] facilitates better bug localization, whereas the complexities inherent in frameworks and tools hurt the bug localization performance of the existing techniques.

4.1.3 Preliminary study on spectrum-based bug localization

Specturm-based bug localization techniques have been widely used for decades in traditional software systems [55]. However, they might face significant challenges when applied to deep learning systems. Spectrum-based bug localization techniques analyze the execution traces of passing and failing tests to measure the frequency with which each program element is executed in both cases [55, 78]. They rank elements by suspiciousness to prioritize fault-prone locations. For example, Tarantula [55] calculates the suspiciousness score based on the ratio of failed to total executions of an element, highlighting those more frequently associated with failures. However, capturing execution traces against passing and failed tests from deep learning systems could be highly challenging. Besides, not all bugs in deep learning systems cause crashes or generate a stack trace [36]. Instead, some lead to incorrect behavior, known as

silent bugs [57]. To better understand the challenges of spectrum-based bug localization techniques, we conducted a case study to assess their applicability in deep learning systems.

Fig. 6: Model architecture of the selected deep learning model

Bug selection: We randomly selected a training bug from a classification model for our case study (see Fig.6). Here, the final layer of the model is defined as `activation=None` (buggy) instead of `activation='softmax'` (correct). The bug causes the final layer to output unnormalized logits. Although the cross-entropy loss is still computed, these raw outputs hinder the expected decrease in loss over multiple epochs, thereby hampering model training.

Ground truth for bug localization: The final layer's activation function is incorrect, which makes the layer buggy (a.k.a., ground truth). We aim to determine whether spectrum-based localization techniques can correctly locate this buggy layer from the model.

Choice of spectrum-based technique: To perform the preliminary study, we chose a seminal work on spectrum-based bug localization – Tarantula [55]. Tarantula uses the execution profiles of test cases to compute suspiciousness scores for program components (e.g., statements). Several contemporary studies [79] have extended Tarantula's spectra-based approach by incorporating machine learning and deep learning techniques. However, we wanted to directly assess Tarantula's effectiveness and limitations in locating faults in deep learning systems without introducing any confounding factors from hybrid methods. Hence, we selected Tarantula for our case study.

Test cases: The training of deep learning models are inherently stochastic due to factors such as random weight initialization and data shuffling [80]. Hence, designing test cases for deep learning models is challenging [81]. However, to simulate the traditional approach for spectrum-based bug localization, we needed to create both failing and passing test cases. Since the final layer of our target model is defined without a softmax activation, its output remains unnormalized. This unnormalized output is processed by the model's loss function (e.g., categorical cross-entropy), but the loss is calculated using raw logits, which is undesirable. However, in some training runs, favorable
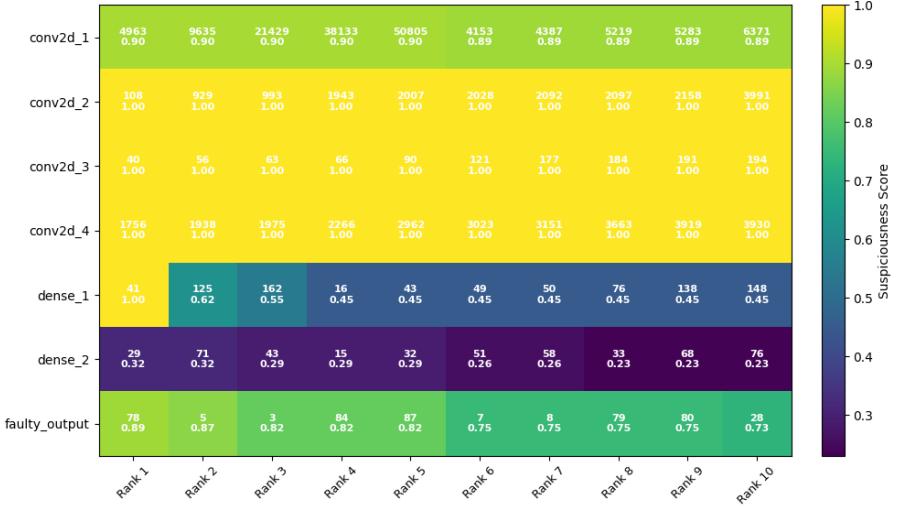
Fig. 7: Top 10 Neuron-Level Suspiousness Scores by Layers

Table 18: Pass/Fail Results from 20 Training Runs

Run	Result	Run	Result	Run	Result
1	Failed	8	Passed	15	Passed
2	Failed	9	Passed	16	Failed
3	Passed	10	Passed	17	Failed
4	Passed	11	Passed	18	Passed
5	Failed	12	Failed	19	Failed
6	Failed	13	Passed	20	Failed
7	Failed	14	Failed		

random initialization and data ordering may cause the loss to decrease, even when the bug (i.e., incorrect activation function) is present. In other cases, the loss might only decrease. Building on prior research [81, 82, 80], we conducted 20 separate training runs, each with 10 epochs and different random seeds, to capture this variability. We define our passing and failing test cases as follows:
- Failing test cases: Training runs where the loss stagnates across N epochs (here, $N = 3$, with a total of 10 epochs).
- Passing test cases: Training runs where the loss decreases despite the model containing the fault.

Coverage Metric: Our study captures test coverage at two levels: *layer level* and *neuron level*. First, we record whether each layer in the model is executed during each training run. This binary coverage metric aligns with traditional spectrum-based techniques, where suspiciousness scores are calculated based on how components are covered by the passing and failing test cases. Similarly, we calculate the suspiciousness score for a given layer l as follows:
  - Failed(l): Number of failing runs that execute layer l.
  - Passed(l): Number of passing runs that execute layer l.

Tarantula integrates these statistics into its standard formula to compute a suspiciousness score, ideally flagging layers with a higher ratio of failing-

to-passing executions as more suspicious. Second, we record the activation of individual neurons within each layer to assess the contribution of neurons to the overall model behavior. Here, a neuron is considered covered if it is activated (i.e., its output is greater than zero) during a training run. We calculate a Tarantula score for each neuron based on its activation frequency in failing versus passing test cases. Our approach is conceptually similar to the granular coverage criteria introduced by Ma et al. [81].

Table 19: Suspiciousness Scores Per Layer

Layer Name	Suspiciousness Score
conv2d	0.5000
conv2d_1	0.5000
max_pooling2d	0.5000
conv2d_2	0.5000
conv2d_3	0.5000
max_pooling2d_1	0.5000
flatten	0.5000
dense	0.5000
dense_1	0.5000
dense_2 (faulty)	0.5000

Results and Discussion: Table 18 shows that out of 20 training runs, 9 runs failed (i.e., loss did not decrease), while 11 runs passed (i.e., loss decreased). Our findings (see Table 19) show that Tarantula assigns the same suspiciousness score to all layers in the model. Since every layer is activated during the forward and backward passes, there is no variability in their passing/failing test coverage. We also incorporated neuron-level coverage for more granular analysis and visualized the results in the color-coded heat map (see Fig.7). In this plot, each row represents a layer, and each column shows the top ten neurons ranked by descending suspiciousness (from 0.0 to 1.0). We chose the top ten neurons for a concise visualization because our conv2d layers contain 64–128 filters, while the dense layers have up to 256 neurons each (see Fig.6), making it impractical to display every neuron in a single figure.

From Fig.7, we see that specific layers, such as *conv2d_2*, *conv2d_3*, and *conv2d_4*, have suspicious neurons, suggesting their strong association with failing runs. In contrast, *conv2d_1* and *faulty_output* (i.e., the last layer where the bug exists*)* exhibit high but varying levels of suspiciousness. It should be noted that the final_output layer contains the bug. Although neuron-level coverage reveals these differences, suggesting that not all neurons in a layer behave the same, the overall patterns indicate uniformly high or near-universal activations. Since traditional spectrum-based bug localization techniques rely on such scores to pinpoint the buggy location, the reported trends make bug localization difficult.

Our preliminary study thus confirms the limitation of spectrum-based bug localization techniques, such as Tarantula, in the context of deep learning systems. The results emphasize that while neuron-level coverage offers more

detailed insights than layer-level coverage alone, both coverage metrics struggle to isolate the bug in the model.

4.1.4 *Preliminary study on dynamic techniques for bug localization*

Deep learning systems differ from traditional software systems due to their reliance on high-dimensional data, hardware accelerators, and complex model architectures [36]. Bugs in deep learning systems can arise from diverse sources, including model configurations, training procedures, and runtime dependencies, making their localization particularly challenging. While static analysis techniques have been widely applied to software bug localization, their effectiveness in deep learning systems is often limited, as many issues only manifest during execution [18]. To address this gap, dynamic fault localization techniques leverage runtime analysis to detect faults by monitoring execution traces and identifying abnormal behaviors [18]. Thus, in this study, we systematically evaluate three state-of-the-art dynamic techniques – DeepLocalize [18], DeepDiagnosis [20], and DeepFD [21] – to assess their effectiveness in localizing bugs in deep learning systems. While these techniques leverage runtime analysis to detect and locate bugs, their practical applicability is constrained by several factors, as follows.

Computational and Practical Challenges: Dynamic-based approaches like DeepLocalize [18], DeepDiagnosis [20], and DeepFD [21] rely on runtime data collection, which is computationally expensive and often impractical for large-scale deep learning programs. Moreover, many bug reports in the Denchmark [32] dataset are linked to large, industry-scale deep learning programs that often miss critical information (e.g., dataset details required to run the model) or include numerous external dependencies [37]. These factors make dynamic analysis both time-consuming and, in many cases, impractical.

Nature of the Techniques: Dynamic approaches differ fundamentally from the static, query-based techniques that we employ. These tools (e.g., DeepLocalize [18], DeepDiagnosis [20], DeepFD [21]) first detect and diagnose faults during program execution, identifying issues at a fine-grained level (such as bugs in specific layers or components like loss function). In contrast, traditional static methods use textual and structural information in bug reports to locate files, methods, or lines where bugs exist [30, 17, 11, 12, 64]. Our study focuses on file-level bug localization, which aligns more with developers' practical steps when addressing a bug, first locating the relevant file in large codebases and then examining its details.

Scope of Bugs Addressed: Another important consideration is that DeepLocalize [18], DeepDiagnosis [20], and DeepFD [21] are specifically designed to address training faults. For example, DeepLocalize [18] targets training-phase numerical faults by analyzing dynamic traces. DeepFD [21] uses machine learning classifiers (e.g., KNN, Decision Tree, and Random Forest) to predict and localize specific training faults (e.g., issues with loss functions, epochs, optimizers, activation functions, or learning rates). DeepDiagnosis [20] focuses on CNN architectures by mapping runtime error symptoms to root causes. Since

our dataset, which reflects real-world scenarios, contains a diverse mix of deep learning bugs beyond just training faults, applying these techniques across the entire dataset is neither practical nor appropriate.

Framework and Architecture Limitation: All these techniques are specifically designed to detect and address faults within the TensorFlow/Keras framework only [18, 20, 21]. Moreover, neither DeepLocalize [18] nor Deep-Diagnosis [20] supports RNN architectures.

Evaluation of Dynamic Approaches: To ensure a fair and comprehensive evaluation, we followed a structured approach:

1. First, we selected 350 bug reports from our dataset (i.e., Denchmark) using stratified sampling, ensuring that they exclusively use TensorFlow/Keras framework. This subset was chosen to achieve a 95% confidence level with a 5% margin of error, ensuring it is statistically representative of our deep learning system (DLSW) dataset.
2. Second, to better understand the performance of dynamic approaches on the training faults they are designed to handle, we manually identified training bugs within our stratified sample. Among the 350 bug reports, we identified 123 unique DL models. This is because multiple bug reports often correspond to the same DL model in the Denchmark dataset for DLSW. We also excluded the RNN-based DL models since DeepDiagnosis and DeepLocalize do not support them, which left us with 101 DL models.
3. Finally, two independent annotators (the authors) conducted a two-phase labeling process. In the first phase, each DL buggy model was categorized as either a training bug or another type of bug. In the second phase, the training bugs were further classified by their root causes, following the definitions provided by dynamic-based approaches [31]. Before starting the complete annotation, the annotators jointly labeled a sample of 10 buggy DL models to ensure a shared understanding of the classification criteria. The inter-annotator reliability, measured by Cohen's kappa coefficient, was 0.88, indicating a strong level of agreement. Any disagreements were resolved through a systematic consensus-building process that involved re-examining the cases using clear taxonomy definitions and additional resources such as literature.

This process resulted in 21 unique DL models containing training bugs with well-defined root causes. We then evaluated [18], DeepDiagnosis [20], and DeepFD [21] on these models.

Results and Discussion:
The performance of dynamic approaches across the 21 models reflects the challenges posed by the heterogeneous nature of DL bugs. While these dynamic approaches demonstrate strengths in localizing specific training bugs (e.g., numerical anomalies, learning rate issues), their limitations underscore the multifaceted nature of DL bugs and the inadequacy of existing state-of-the-art techniques for bug localization. Table 20 shows the performance of these

Table 20: Performance of Dynamic Approaches for Locating Training Bugs from DLSW

Technique	Accuracy	Precision	Recall	F1-Score
DeepLocalize	0.23	0.25	0.20	0.22
DeepFD	0.28	0.30	0.25	0.27
DeepDiagnosis	0.33	0.35	0.30	0.32

techniques against each of the 21 bugs. Based on our evaluation, we found several limitations of these techniques, as follows.

1. Limited Scope: All three dynamic techniques demonstrate significant limitations, even when applied to a small subset of training bugs, due to their narrow scope (see Table 21). DeepLocalize, for instance, focuses on identifying numerical anomalies like NaN or Inf values during bug tracing. While it successfully located a faulty layer caused by an incorrect activation function by detecting NaN values in the loss (Bug 17), it failed to detect bugs that did not involve numerical abnormalities (e.g., Bug 1, 2, 3). Similarly, DeepDiagnosis relies on predefined heuristics to map symptoms to root causes, which restricts its ability to identify issues like an incorrect dropout rate (Bug 21). DeepFD, which uses classifiers trained on runtime features, also struggled with bugs outside its predefined fault categories, such as regularization issues (Bug 10). Our analysis above reveals a troubling trend: even for the specific category of bugs for which they were targeted (e.g., training bugs), these state-of-the-art techniques often fail to detect issues. This suggests that they may not be sufficient for effective bug localization in real-world deep-learning projects, where the bugs tend to be more complex and varied, even within the same category.

2. Inability to Handle Extrinsic Bugs: A key finding from RQ3 is that 40% of DL bugs are extrinsic, meaning they arise from external dependencies such as GPU issues, third-party libraries, or training data. We found that these dynamic bug localization techniques, which rely on analyzing runtime information, are inherently unable to detect such bugs. For example, in our evaluation (see Table 21), the three methods failed to identify bugs caused by extrinsic factors like incorrect or mislabeled training data (Bug 4, 16). These issues originate from issues in the data pipeline rather than the model's execution, making them invisible to runtime analysis. This limitation highlights a critical gap — extrinsic bugs require approaches that go beyond code execution or runtime monitoring.

3. Challenges with Complex System Interactions: A major insight from our evaluation of training bugs (21 buggy models) is that both static and dynamic methods struggle to address the intricate, often indirect interactions inherent in deep learning systems. For example, state-of-the-art static techniques like bjXnet [30] focus on code and textual bug reports, often overlooking issues such as unchanged model weights that only surface during execution (e.g., Bug 9). Meanwhile, dynamic tools like DeepLocalize, DeepDiagnosis, and DeepFD detect runtime anomalies (e.g., NaN values in Loss) but often fail to link these symptoms to their root causes, such as improper dropout rates causing

Table 21: Performance Comparison of Dynamic Approaches for Training Bug Localization

No.	Symptom	Root Cause (Bug)	DL	DF	DD
1	Loss Not Decreasing	Incorrect Loss Function	✗	✓	✓
2	Loss Not Decreasing	Improper Activation Function	✗	✗	✓
3	Loss Increasing After Epochs	Incorrect Optimizer	✗	✓	✗
4	Model Overfitting	Incorrect Training/Validation Data	✗	✗	✗
5	Static Validation Loss	Incorrect Dropout Usage	✗	✗	✗
6	Vanishing Loss & Overfitting	Small Training Data	✗	✗	✗
7	Overfitting	Incorrect Regularizer & Small Data	✗	✗	✗
8	Loss Oscillating & Overfitting	Incorrect Dropout Rate	✗	✗	✗
9	Unchanged Weights & Static Loss	Incorrect Learning Rate	✗	✓	✓
10	Poor Generalization	Incorrect Normalization & No Regularizer	✗	✗	✗
11	NaN Values in Loss	Incorrect Weight Initialization	✓	✗	✓
12	Poor Performance (Low Accuracy)	Incorrect Epoch Number	✗	✓	✗
13	Train/Val Loss Divergence	Incorrect Batch Normalization	✓	✗	✗
14	Neurons Outputting Zero	Incorrect Weight Initialization	✓	✗	✓
15	Training Slowdown	Incorrect Dropout Rate	✗	✗	✗
16	Overfitting	Incorrect Regularizer/Augmentation	✗	✗	✗
17	NaN Values in Loss	Incorrect Activation Function	✓	✓	✓
18	NaN Values in Loss	Zero Sample Weights	✓	✗	✗
19	Loss Plateaus	Improper Bias Initialization	✗	✗	✗
20	Performance Degradation	Incorrect Learning Rate	✗	✓	✓
21	Poor Generalization	Incorrect Dropout Rate	✗	✗	✗
		Accuracy (%)	**23.8**	**28.6**	**33.3**

DL: DeepLocalize, DF: DeepFD, DD: DeepDiagnosis. ✓: Successfully identified, ✗: Failed to identify.

NaN values in the loss (Table 21). Though dynamic approaches identified some training faults, their overall accuracy remained below 33% (Table 20), highlighting their inability to address deeper systemic interactions.

Our findings demonstrate that deep learning software presents unique challenges that cannot be fully resolved by static or dynamic analysis alone. The limitations observed in both approaches suggest that bugs often arise from complex interactions between code, runtime behaviour, and system-level dependencies, which current tools are not equipped to handle comprehensively. These results align with our broader evaluation, emphasizing the need for more integrated techniques to localize bugs in DLSW effectively.

Summary of RQ$_1$: We evaluate the performance of five existing techniques in localizing bugs from deep learning systems and non-deep learning systems using three evaluation metrics. Our findings show that all five approaches perform significantly lower (e.g., **34.14%** less MAP for DNNLOC, **27.87%** less MAP for bjXnet) when localizing bugs from deep learning systems. We also compare their performance when localizing bugs from deep learning frameworks, libraries, and tools. We found that localizing bugs from frameworks is most challenging due to the complex interaction of their components.

4.2 Answering RQ$_2$: How do different types of bugs in deep learning systems impact bug localization?

In this research question, we investigate the characteristics and localization challenges of different types of bugs in deep learning systems through manual analysis. To classify different types of bugs in deep learning systems, we adopted the following steps:
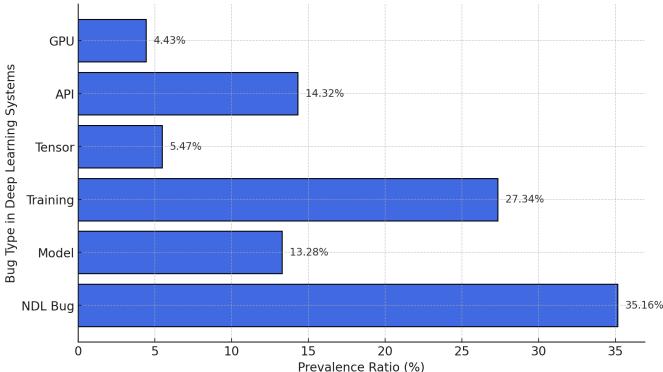

Fig. 8: Prevalence ratio of each category of bugs from deep learning systems

- We conducted our manual analysis on a sample of 385 bug reports, which have a 95% confidence level and a 5% margin of error. This allows our findings to be generalizable to the original population of deep learning system bugs.
- We classified deep learning-related bugs into five categories using the taxonomy of Humbatova et al. [31]. This taxonomy provides explicit definitions for various DL bug types, including Model, Training, Tensor, API, and GPU issues. For instance, bugs involving neural network architecture or weight initialization were classified as 'Model' bugs, while issues with data preprocessing or augmentation during model training were categorized as 'Training' bugs. On the other hand, file handling errors or user interface glitches were categorized as Non-deep learning-related (NDL) bugs.

– Two independent annotators (authors of this study) conducted a two-phase annotation. In the first phase, each bug was labeled as DL or NDL. In the second phase, DL bugs were further categorized into the five sub-types mentioned above. Prior to annotation, the annotators jointly labeled a sample of 20 bug reports to assess their understanding of the classification criteria.

– The annotators cross-checked their annotations/classifications with project documentation and bug reports to prevent overlaps or misclassifications. Project documentation provided context and clarified the intended system functionality, helping to classify bugs accurately and avoid misclassification.

– We measured inter-annotator reliability using Cohen's kappa coefficient [83], resulting in a score of 0.80. This substantial agreement indicates a high level of consistency in our classification process. We resolved disagreements through a systematic consensus-building process involving joint re-examination of disputed cases, guided by taxonomy definitions and cross-referenced with additional documentation to ensure accuracy and consistency.

– We documented this manual analysis using an Excel sheet, with a total of ≈55 hours spent by each author, which is provided in our replication package [66].

Prevalence ratio of deep learning-related bugs: We found that 64.80% of the bugs from deep learning systems are related to deep learning algorithms (a.k.a DL bugs). That is, they are related to inputs, data, or training of deep learning models, underlying API endpoints, and computational resources. In particular, we found 27.30% training bugs, 13.30% model bugs, 5.50% tensor bugs, 14.30% API bugs, and 4.40% GPU bugs (Fig. 8). Such a distribution informs us where the debugging efforts should be concentrated. Our findings also indicate that the majority of DL bugs are related to the training process. Training is a crucial step in deep learning that involves large amounts of data, complex learning algorithms, and optimization techniques, making it more susceptible to bugs and failures.

Prevalence ratio of non-deep learning-related bugs: We found that 35.20% of the bugs from deep learning systems are not related to deep learning algorithms (a.k.a NDL bugs). These bugs do not directly affect the functionality of the deep learning model, but they still lead to unexpected, erroneous behaviors in a software system. Bug 1426 in Table 1 is an NDL bug, which occurs when the tests from the CI pipeline are spread across multiple Windows machines. Although it is not directly connected to the deep learning module, it originated from the PyTorch-Ignite project, which is indeed a deep learning system.

Localization of bugs in deep learning systems: To gain a deeper understanding of the challenges in localizing deep learning bugs, we analyzed the
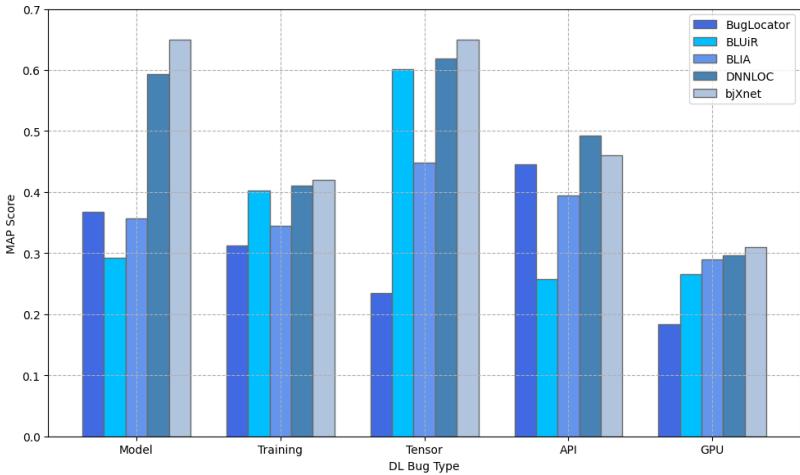
Fig. 9: Performance of existing bug localization techniques (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) for each type of bug in deep learning systems

performance of our baseline techniques across different categories of bugs in DL systems. We employed stratified random sampling [84] to select a balanced number of bug reports (100 bugs) from each category for performance comparison, ensuring fair representation and reducing selection bias. We repeated the process three times with different samples to ensure robustness. The results were averaged across these evaluations, providing reliable insights into the efficacy of the bug localization methods. In addition to the quantitative results, we conducted an in-depth analysis of each type of DL bug to understand their inherent challenges and how they impact the overall performance of the bug localization techniques. This detailed analysis, with examples, is provided below.

Model bugs: Model bugs present unique challenges in bug localization for deep learning systems. Our analysis in Table 22 reveals that bjXnet demonstrates superior performance in localizing model bugs. This can be attributed to its attention mechanisms, which enable focused examination of key subcomponents. Model bugs typically relate to a model's type, properties, and layers. We observed significant vocabulary overlap between bug reports describing model-related issues and the implementation of deep learning models. bjXnet's attention-based approach allows it to prioritize critical code sections, such as specific methods or functions directly implicated in the bug report. DNNLOC also performs well, followed closely by bjXnet, by employing deep neural networks to capture complex patterns and utilizing rVSM to assess textual relevance between bug reports and source code. In contrast, BLUiR shows the poorest performance in locating model bugs.

Table 23 [40] illustrates an example of a model bug from the `CASL.ai` project, with the corresponding code snippet shown in Fig. 12 [40] (Appendix

Table 22: Experimental result of existing bug localization techniques (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) for each category of bugs in deep learning systems

Method	Model	Training	Tensor	API	GPU
MAP					
BugLocator	0.368	0.312	0.235	0.446	0.183
BLUiR	0.293	0.403	0.601	0.258	0.266
BLIA	0.357	0.345	0.448	0.395	0.290
DNNLOC	0.593	0.411	0.619	0.493	0.296
bjXnet	0.650	0.420	0.650	0.460	0.310
MRR					
BugLocator	0.532	0.386	0.358	0.378	0.223
BLUiR	0.387	0.427	0.621	0.424	0.311
BLIA	0.472	0.419	0.553	0.447	0.327
DNNLOC	0.585	0.467	0.682	0.548	0.336
bjXnet	0.720	0.480	0.680	0.510	0.360

MAP= Mean Average Precision **MRR**=Mean Reciprocal Rank

Table 23: Example of a model bug [40]

Model Bug (Bug ID: 313)
Title
A bug in GPT2Tokenizer
Description
GPT2Tokenizer fails to recover a sentence
\"BART is a seq2seq model.\"
with encoded ids of it.
The output sentence is \"BART is a seqseq model.\".
It should be related to numbers' processing.
A script to show the bug is here:
https://github.com/tanyuqian/texar-pytorch
/blob/master/examples/bart/gpt2_tokenizer_bug.py
Detailed BR:https://github.com/asyml/texar-pytorch/issues/313

A). The bug occurs in the `bpe` method of the `GPT2Tokenizer`, causing faulty tokenization that impacts the GPT2 language model's functionality.

We noticed that IR-based techniques struggle with locating model bugs. For instance, BugLocator incorrectly identifies 'SentencePieceTokenizer.py' (Fig. 13, Appendix A) as the Top@1 result, which is likely to stem from lexical overlap between the bug report and source code. Four key terms from the bug report (Table 23) – *GPT2Tokenizer, recover, seq2seq, and model* – appear in the incorrect file, potentially leading to this incorrect localization. Notably, BugLocator retrieves the actual ground truth file at the 8th position.

On the other hand, BLIA ranks the ground truth relatively low at the 17th position, potentially because meta components, such as stack trace, version control history, or commit history, are less effective in localizing model bugs. Model bugs typically involve deep architectural issues within the neural network, which are not easily traced through surface-level meta-information,

making BLIA less suitable for these types of bugs compared to traditional software bugs.

BLUiR performs even worse, ranking the ground truth code at the 131st position. This suggests that the similarity between bug reports and source code elements may be insufficient for accurately locating model bugs. In the example from Table 23, the bug manifests when tokenizing the sentence 'BART is a seq2seq model.' BLUiR mistakenly retrieves the source code file containing the `Seq2Seq` class as the Top@1 result. This error likely occurs because several important keywords from the bug report align with similar code elements (e.g., 'seq2seq', 'encode', and 'model' matching the `seq2seq` class). Such misalignment results from BLUiR's strong emphasis on structural elements (e.g., class, method, variable) and insufficient consideration of the bug report's semantic aspects.

In contrast, both bjXnet and DNNLOC successfully retrieve the buggy file at Top@1. Unlike IR-based methods, DNNLOC's hybrid approach leveraging neural networks can capture deeper semantic links beyond lexical similarities. For instance, despite textual mismatches, it can better connect terms like 'seq2seq' and 'tokenizer' in the bug report to their corresponding code-level implementations in GPT2Tokenizer. DNNLOC's ability to leverage non-linear relationships likely contributes to its superior performance in localizing model bugs in deep learning systems. bjXnet's success can be primarily attributed to its attention mechanism, which enables the targeting of critical subcomponents within the code, such as the `bpe` method in the GPT2Tokenizer. By focusing on the most relevant methods and variables implicated in the bug report, bjXnet achieves higher precision, especially in modular deep-learning components like tokenization and layer definitions, where pinpoint accuracy is crucial for fault identification.

Training bugs: All five approaches perform poorly in localizing training bugs, as shown in Table 22. bjXnet, DNNLOC, and BLUiR perform comparatively better than the other techniques, though their overall MAP values remain low (40.30% – 42.00%). Table 24 [41] illustrates a training bug from the `fast.ai` project, where the interaction between `Gradient Accumulation` and the `MixedPrecision Callback` leads to artificially high training loss values.

For this example, DNNLOC retrieved the ground truth file at Top@4, while BLUiR ranked it at Top@7. bjXnet performs slightly better than DNNLOC in localizing training bugs and retrieving the ground truth file at Top@3. bjXnet's inability to rank the correct file at Top@1 likely stems from the distributed nature of training bugs. These bugs often span across multiple interconnected components like callbacks, loss functions, and training loops, making it harder to pinpoint a single root cause. While bjXnet's attention mechanism can highlight important sections like callbacks and loss scaling, training bugs tend to involve interactions between different parts of the code. This complexity means that the root cause may not be confined to a single location, requiring a broader contextual understanding of the training process, which limits bjXnet's ability to achieve Top@1 localization

Table 24: Example of a training bug [41]

Training Bug (Bug ID: 3048)
Title
Gradient Accumulation + Mixed Precision shows artificially high training loss
Description
OB: The bug occurs when Gradient Accumulation and the MixedPrecision Callback are both used. Gradient Accumulation runs before Mixed Precision and causes the after_backwards to not be run, meaning that the loss is not unscaled before it is logged. This means that very large losses, such as 6000000+, are to be logged. S2R: seed=random.randint(0,2**32-1) with no_random(seed): db=synth_dbunch(bs=8,n_train=1,n_valid=1,cuda=True) learn = synth_learner(data=db) learn.fit(1, lr=0.01) #start without gradient overflow max_loss_scale=2048.0 with no_random(seed): db=synth_dbunch(bs=1,n_train=8,n_valid=8,cuda=True) learn = synth_learner(data=db,cbs=[GradientAccumulation(n_acc=8)]) learn.to_fp16(max_loss_scale=max_loss_scale) learn.fit(1, lr=0.01) The training loss will be very high, 5000+ for fp16. fp32 will be reasonable EB: Similar training loss between the fp32 and fp16 versions. <2 difference in loss. Detailed BR: https://github.com/fastai/fastai/issues/3048

EB=Expected Behaviour, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

On the other hand, BLIA also performs poorly, ranking the correct file at Top@21. Meta information like stack traces and version history from BLIA might not be particularly helpful for training bugs, as they often involve complex, abstract issues (e.g., improper configurations of loss functions or callbacks), which do not leave clear traces in version control or stack traces. Moreover, BugLocator performs the worst (e.g., Top@23 position for the example bug in Table 24) in localizing the training bugs (Fig. 16) (as shown in Appendix A), which suggests that similarity analysis between bug reports and source code might not be sufficient for identifying these bugs. Training bugs, which occur during the model's training phase and might involve aspects such as an improperly defined loss function, are more conceptual. These bugs might not be easily located using these existing methods and might require techniques that can offer a deeper insight into the model's architecture and training process.

Tensor bugs: Tensors are central to deep learning and often involve intricate dimensional and mathematical issues. From Fig. 9, we notice that bjXnet, DNNLOC, and BLUiR are more effective in localizing tensor bugs than other

techniques (e.g., BugLocator, BLIA). In the example bug from Table 25 [42], the bug report described the issue with the 'nd.slice' function in MXNet, which should return an empty tensor when the 'begin' and 'end' parameters are equal. Instead, the function returns a tensor with the same shape as the data.

Table 25: Example of a tensor bug [42]

Tensor Bug (Bug ID: 13760)
Title
nd.slice does not return empty tensor when begin=end
Description
OB: For mxnet.ndarray.slice(data, begin, end),
if begin=end, it does not return an empty tensor.
Instead, it returns a tensor with the same shape as the data.
Environment info:
......
......
......
S2R: import mxnet.ndarray as nd
a = nd.normal(shape=(4, 3))
nd.slice(a, begin=0, end=0)
nd.slice(a, begin=2, end=2)
Detailed BR: https://github.com/apache/mxnet/issues/13760

BR=Bug Report, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

For this specific tensor bug, bjXnet, DNNLOC, and BLUiR all successfully retrieved the correct buggy file at Top@1 due to their targeted handling of tensor-related operations. bjXnet's attention mechanism focuses on key operations, such as the `slice` function, which mishandles tensor shapes when `begin` equals `end`. This targeted focus might allow bjXnet to effectively identify the specific code causing the bug. DNNLOC captures the complex interactions between the bug report and tensor manipulations, mapping the error in how the `nd.slice` function handles dimensions directly to the buggy code. Interestingly, by parsing the AST of the code, BLUiR identified the relevant code snippet in the test_slice() function, which shares significant keyword overlap with the bug report. BLUiR determines the relevance of low-level code elements (e.g., class names, method names) against a bug report, which helps to reduce noise in the code segments where tensors could be handled or manipulated.

On the other hand, BLIA retrieved the ground truth file at the top 12 positions. However, our analysis revealed that incorporating the stack trace information negatively impacted its bug localization performance. Tensor bugs are typically related to data manipulation and computations [31] rather than code execution flow or call stack [85] that can be found in stack traces. By excluding the stack trace from the BLIA approach, we were able to improve the ranking of the buggy file from the 12th position to the 9th position.

Lastly, BugLocator's difficulty in accurately locating tensor bugs is demonstrated by its retrieval of the ground truth file at the 47th position (Fig. 17) (as shown in Appendix A) and the incorrect file at Top@1 (Fig. 18) (as shown in Appendix A). It can be attributed to its reliance on textual similarity. We found that BugLocator's heavy reliance on textual similarity and the overlapping of trivial words (e.g., environment, system, and hardware) with the incorrect file (Fig. 18) (as shown in Appendix A) led to the incorrect ranking.

API bugs: From Fig. 9, we observe that DNNLOC performs well in localizing API bugs, whereas BLUiR is the least effective. This could be due to BLUiR's heavy reliance on structural information from the source code, which might not capture the specifics of API bugs (e.g., incorrect API calls). Another factor could be the rapid evolution of deep learning APIs, which affects versioning and compatibility [86]. Due to frequent structural changes in the code, the technique might be less effective in assessing the relevance between code and bug reports.

Table 26: Example of an API bug [43]

API Bug (Bug ID: 13862)
Title
[1.4.0] unravel_index no longer works with magic '-1' in shape parameter as in 1.3.1
Description
OB: The unravel_index op seems to no longer correctly work with 'magic' shape values, such as '-1's. The following example still works with mxnet 1.3.1, but does not on the latest master (it returns all zeros in the result without throwing an error) or 1.4.0. We have a use case for this in Sockeye. Environment info (Required): ... S2R: Input data taken from Sockeye unit tests. x = mx.nd.array([335, 620, 593, 219, 36], dtype='int32') mx.nd.unravel_index(x, shape=(-1, 200)) With mxnet==1.5.0b20190111, the result is incorrect: With mxnet==1.3.1, the result is correct: However, if the shape parameter is fully specified (shape=(5,200)), mxnet==1.5.0b20190111 returns the correct values. Detailed BR: https://github.com/apache/mxnet/issues/13862

BR=Bug Report, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

The bug from Table 26 [43] involves the `unravel-index` function in MXNet, where its behavior incorrectly varies with certain input parameters across different versions (Fig. 19) (as shown in Appendix A). DNNLOC located the correct buggy code for the example bug (Table 26) at the 2nd position, while BugLocator ranked it in the top 5th. DNNLOC might be effective for this API bug because textual similarity (through rVSM) matches API-specific terms in

bug reports with source code, while neural networks (through DNN) can capture complex, non-obvious relationships in the API's usage and functionality. Moreover, our findings indicate that relying on either element (rVSM, DNN) in isolation is less effective, resulting in a noticeable drop in performance.

On the other hand, BLUiR retrieved the buggy file at the 38th position. BLUiR incorrectly retrieved the file containing the `NDArray` class. In the case of API bugs, the class file may not always be relevant. This is because API bugs frequently arise at the interface between different layers of abstraction [87]. Such bugs could be attributed to the interaction of a higher-level function (e.g., `unravel-index` from Table 26) with lower-level components rather than an issue within the code of the function itself. Meanwhile, BLIA initially placed the buggy file at the top 7th position without the stack traces but improved to 5th when stack trace information was included. It indicates that stack trace information, highlighting execution flow and function calls, is beneficial for locating API bugs.

Interestingly, despite its advanced attention mechanisms, bjXnet ranked the correct buggy file at the 6th position, lower than DNNLOC and BugLocator. We found that bjXnet's attention mechanism might over-prioritize certain structural sections of the code, causing it to miss key interactions between components, which are critical for localizing API bugs. For example, in the unravel-index bug (Table 26), bjXnet correctly identified the key `unravel-index` function but failed to capture how the interactions between the API and different framework versions caused the bug. The issue arose from the `shape` parameter's change in behavior across MXNet versions, but bjXnet's focus on specific code segments may have led it to miss this broader context.

GPU bugs: Our investigation reveals that GPU bugs are the most difficult to localize for all five existing approaches. One possible explanation could be the complex nature of GPU bugs, as they can be triggered by a variety of factors, such as the compatibility between hardware (e.g., GPU device) and software (e.g., PyTorch). It might not even be located in the source code (a.k.a extrinsic GPU bug) [31]. From Table 29, we notice that only 17.65% of the GPU bugs can be found in the source code (a.k.a intrinsic GPU bug). Some examples of intrinsic GPU bugs that we found – the wrong reference to a GPU device, failed parallelism, incorrect state sharing between subprocesses, and faulty data transfer to a GPU device.

Table 27 [44] shows a GPU bug triggered by the codebase (a.k.a intrinsic). The bug is connected to the use of multiple GPUs during training. According to the report, the machine contains multiple GPU devices, but only one GPU is used during computation. All five bug localization techniques performed poorly in locating the actual buggy code for this GPU bug. BLUiR retrieved the buggy file at the 50th position, followed by BugLocator at the 65th, BLIA at the 47th, DNNLOC at the 43rd, and bjXnet at the 42nd position in their respective rankings.

Table 27: Example of a GPU bug [44]

GPU Bug (Bug ID: 1238)
Title
How to use multiple GPUs?
Description
I want to use a single machine with multiple GPUs for training, but it has no actual effect. OB: Only one single GPU is doing all the computations, the other three remain idle. When following @FontTian and inserting distribution_strategy=strat into the initialization of the image classifier, the same error RuntimeError: Too many failed attempts to build the model occurs. The same happens when adding tuner='random' to ak.ImageClassifier. As suggested by @haifeng-jin, I ran a basic KerasTuner example on 4 GPUs which worked just fine. Furthermore, in #440 (comment), I read that the clear_session() before every run might wipe out the GPU configuration. Removing this line from the code did not change anything with respect to the errors/problems stated above. I am specifying 4 GPUs (out of 8) to train the current model in a distributed fashion, using tf.distribute.MirroredStrategy( ) since tf.keras.utils.multi_gpu_model( ) is deprecated and removed since April 2020. S2R: def make_model(ckpt_path, max_try = 1): ...... run_search(checkpoint, max_try = 3) Detailed BR: https://github.com/keras-team/autokeras/issues/1238

BR=Bug Report, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

We observed that there is almost no keyword overlapping and no structural similarity between the bug report (Table 27) and the actual buggy code (Fig. 20) (as shown in Appendix A). This suggests that these techniques struggled due to the lack of both textual and code-wise similarity between the bug report and source code, making it challenging to identify the buggy code for the GPU bug.

We observed a minimal keyword overlap and structural similarity between the bug report (Table 27) and the actual buggy code (Fig. 20) (as shown in Appendix A), presenting a significant challenge in locating such bugs using BugLocator. Additionally, BLUiR's analysis of smaller code segment similarity also proves ineffective for GPU bugs, as it solely concentrates on source code and overlooks the hardware-software interactions and runtime specifics crucial for comprehending such bugs. Similarly, BLIA's integration of stack trace information, commit history, or version control history fails to encapsulate the unique aspects of GPU bugs as well. Moreover, despite DNNLOC's and bjXnet's capability to capture non-linear complex relationships through deep neural networks, it did not help to locate GPU bugs. One reason might be that these bugs often involve complex hardware-software dynamics that are

not typically addressed by standard source code analysis or within the code-base's non-linear mappings.

Table 28: Experimental result of existing bug localization techniques (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) for non-deep learning-related (NDL) bugs Bugs in deep learning systems

Method	MAP	MRR
BugLocator	0.362	0.417
BLUiR	0.292	0.334
BLIA	0.437	0.381
DNNLOC	0.588	0.439
bjXnet	0.495	0.450

NDL bugs: We found that 35.20% of bugs in deep learning applications are not directly related to deep learning, but they impact system behavior (e.g., failed CI build due to GPU compatibility issues). These bugs are known as Non-Deep Learning-related (NDL) bugs in deep-learning systems. From Table 28, we find that the performance of existing techniques in localizing these bugs is also poor. DNNLOC outperforms other techniques, whereas BLUiR performs the lowest in locating NDL bugs. We observed that NDL bugs are less complex than their deep-learning counterparts. However, they are more prone to be extrinsic than the traditional bugs (48.15% from Table 29); we provided the details about extrinsic bugs in RQ$_3$. Since existing baseline techniques focus on code-level artifacts only, they might fall short in detecting these bugs from deep learning systems.

Bug report quality for bugs in deep learning systems: Our analysis showed that bug reports from deep learning systems contain more code snippets (83.11%) than traditional software systems (33.24%). Unfortunately, that does not help much in bug localization, as code snippets alone might not be sufficient. Deep learning bugs often involve intricate dependencies that extend beyond specific code components (e.g., training data bugs and GPU bugs). Complex bugs (e.g., gradient instability during training) warrant a deeper understanding of the model architecture, its dynamic behavior, and training processes, which the code snippets may not always capture.

Summary of RQ$_2$: We found that **64.80%** bugs in deep learning systems (DLSW) are related to deep learning algorithms, whereas the remaining bugs are not related to deep learning. Our analysis shows that Tensor bugs and API bugs are easier to localize than model and training bugs. However, GPU bugs are the most difficult to localize for each of the five approaches. Thus, our results not only inform the distribution of DL bugs but also highlight their localization challenges through extensive experiments.
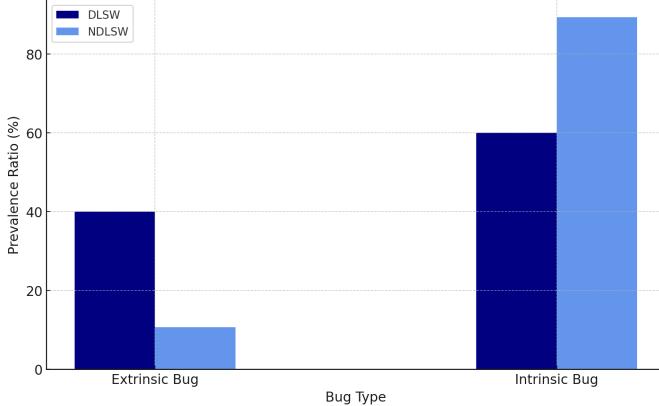
Fig. 10: Prevalence ratio of extrinsic and intrinsic bugs in deep learning systems (DLSW) and non-deep learning systems (NDLSW)

4.3 *Answering RQ$_3$: What are the implications of extrinsic bugs in deep learning systems for bug localization?*

Most of the traditional bug localization techniques rely on the similarity between bug reports and source code [11, 6, 12, 17, 14, 13, 23, 53]. However, if a bug is of an extrinsic nature (e.g., originates from the operating system), simply relying on source code may not be effective for its localization.

To investigate the impact of extrinsic bugs in deep learning systems, we performed another manual analysis following a set of steps as follows.

- We conducted our manual analysis on a sample of 385 bug reports, which have a 95% confidence level and a 5% margin of error. This allows our findings to be generalizable to the original population of deep learning system bugs.
- We manually labeled 385 bug reports from deep learning software (DLSW) and 385 from non-deep learning software (NDLSW) as either extrinsic or intrinsic. We selected equal amounts of bug reports using randomization principles to ensure a fair comparison during the evaluation of bug localization techniques. We used stratified random sampling [84] ensuring that our dataset was representative of both deep learning and traditional systems, reducing sampling bias that could affect the prevalence ratio of extrinsic and intrinsic bugs across categories.
- We strictly adhered to established heuristics [37] to annotate extrinsic and intrinsic bugs. For example, a bug was classified as extrinsic only if there was clear evidence of external factors, such as system call failures or library version mismatches, documented in the bug report or related discussions.
- We achieved a Cohen's kappa [83] score of 0.87, indicating substantial agreement and reinforcing the reliability of our manual labeling process. We resolved disagreements through a systematic consensus-building process involving joint re-examination of disputed cases, guided by taxonomy

definitions and cross-referenced with additional documentation to ensure accuracy and consistency.

– We documented the manual analysis using an Excel sheet, which can be found in our replication package [66]. Each author spent a total of ≈ 20 hours on the analysis.

Prevalence ratio of extrinsic & intrinsic bugs: We found 40.00% extrinsic bugs within a total of 385 bug reports from deep learning systems (Denchmark dataset). The notion of extrinsic bugs is relatively new, especially in the case of bugs from deep learning systems. For a better comparison, we also manually inspected 385 bugs from non-deep learning systems (BuGL dataset) and determined the prevalence ratio of extrinsic and intrinsic bugs. We found only 10.65% extrinsic bugs in non-deep learning systems. Thus, deep learning systems contain almost four times more extrinsic bugs (Fig. 10) than non-deep learning systems.

Table 29: Prevalence ratio of extrinsic and intrinsic bugs in deep learning systems

Type		Extrinsic (%)	Intrinsic (%)
NDL		48.15	51.85
DL	Model	35.29	64.71
	Training	21.90	78.10
	Tensor	38.10	61.90
	API	38.19	61.81
	GPU	82.35	17.65

Prevalence ratio of extrinsic & intrinsic bugs from deep learning systems: We randomly select 100 samples for each type of bug from deep learning systems (same subsets from RQ$_2$) and determined the prevalence ratio of extrinsic and intrinsic bugs for each type. Table 29 shows the results of our manual analysis for different bug categories in deep learning systems in terms of extrinsic and intrinsic bugs. We see that the prevalence ratios of deep learning-related extrinsic bugs range from 21.90% to 82.35%, whereas for non-deep learning-related bugs, the prevalence ratio is 48.15%. This suggests that the deep learning components of a software system might be more likely to trigger extrinsic bugs than non-deep learning components.

Localization of extrinsic & intrinsic bugs from both systems: To further analyze the impact of extrinsic bugs on bug localization, we experimented with our baseline techniques from RQ$_1$ on extrinsic and intrinsic bugs separately. We chose 100 random bugs to evaluate the performance of all five techniques in bug localization from each category of both benchmark datasets. We repeated the evaluation three times using three different random subsets and then calculated the average result for a fair comparison.
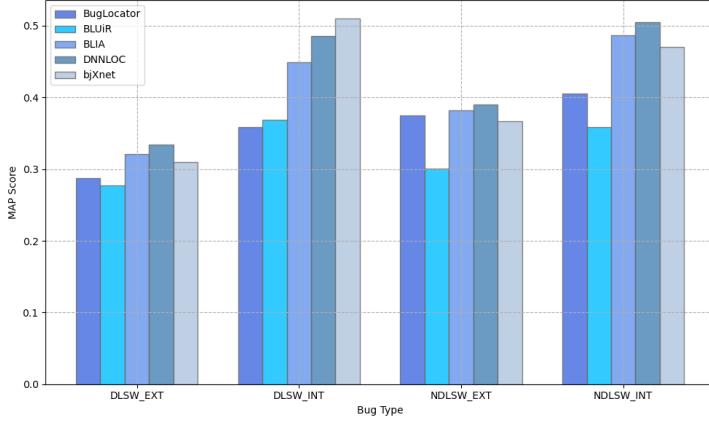
Fig. 11: Performance of existing approaches (BugLocator, BLUiR, BLIA, DNNLOC, bjXnet) for localizing extrinsic and intrinsic bugs in both systems (DLSW & NDLSW)

From Table 30, we notice that DNNLOC performs slightly better than other techniques in localizing extrinsic bugs for both systems. However, it shows a clear performance gap in localizing extrinsic and intrinsic bugs. The technique is less effective with extrinsic bugs, particularly in DLSW. It suggests a shortcoming in handling external complexities despite being able to capture non-linear relationships between bug reports and source code. We notice that BugLocator's localization performance for extrinsic bugs in DLSW is lower than NDLSW. BugLocator might not be able to locate the extrinsic bugs in deep learning systems due to its naive approach, i.e., considering code as regular texts. We also found that BLUiR shows less performance gap between extrinsic and intrinsic bugs. It extracts different structured items (e.g., methods, classes) from the source code and bug reports. Thus, even if they reside outside the current codebase and are invoked from an external library, they could be matched with relevant keywords from a bug report.

Interestingly, BLIA performs slightly better for extrinsic bugs in DLSW than NDLSW, which could be possible due to its diverse use of meta components (e.g., stack traces, version control history). Stack traces from deep learning systems (DLSW) often contain more intricate information and dependencies (e.g., complex neural network data flows, dependencies on specialized libraries, GPU-related synchronization issues) related to the bugs, unlike stack traces from non-deep learning systems, which might not provide the same level of detailed information [88, 89].

When considering bjXnet, its performance with extrinsic bugs is slightly lower than DNNLOC in DLSW but outperforms it in NDLSW. While bjXnet's attention mechanism effectively isolates critical components in the code, extrinsic bugs often involve dependencies on external libraries and system-wide interactions that are harder to capture through localized attention. This limitation could explain bjXnet's slightly reduced performance for extrinsic bugs,

Table 30: Performance of Bug Localization Techniques in DL and TRAD Categories

Method	DLSW+EXT	DLSW+INT	NDLSW+EXT	NDLSW+INT
MAP				
BugLocator	0.287	0.359	0.375	0.405
BLUiR	0.277	0.369	0.301	0.359
BLIA	0.321	0.449	0.382	0.487
DNNLOC	0.334	0.486	0.390	0.505
bjXnet	0.310	0.510	0.367	0.470
MRR				
BugLocator	0.371	0.489	0.409	0.476
BLUiR	0.296	0.445	0.309	0.364
BLIA	0.385	0.518	0.404	0.505
DNNLOC	0.398	0.544	0.420	0.521
bjXnet	0.380	0.535	0.392	0.496

DLSW+EXT = Extrinsic Bug from Deep Learning Systems
DLSW+INT = Intrinsic Bug from Deep Learning Systems
NDLSW+EXT = Extrinsic Bug from Non-Deep Learning Systems
NDLSW+INT = Intrinsic Bug from Non-Deep Learning Systems

particularly in deep learning systems where external factors, like third-party libraries or hardware dependencies, play a significant role. Despite this, bjXnet's ability to focus on key sections of the code still allows it to perform well with intrinsic bugs across both systems.

Overall, these results suggest that extrinsic bugs are hard to localize, whether related to deep learning or not. However, deep learning bugs with an extrinsic nature are the more difficult to localize. On the other hand, from Fig. 11, we note that the performance of all five approaches for intrinsic bugs in NDLSW is higher compared to the intrinsic bugs in DLSW, which supports the fact that the bugs related to deep learning algorithms (a.k.a DL bugs) from deep learning systems are more challenging to localize. Furthermore, we conducted appropriate statistical significance tests and found that the performance differences between extrinsic and intrinsic bugs in both systems are statistically significant, with a p-value of 0.014 for the difference between DLSW+EXT and DLSW+INT, and a p-value of 0.021 for NDLSW+EXT and NDLSW+INT, both indicating medium effect sizes, which reinforce the observed trends.

Correlation of extrinsic & deep learning-related bugs: To determine the potential correlation between the extrinsic bugs and the bugs in deep-learning systems, we performed a Chi-Square test to determine any significant association [90]. We conducted three iterations with different sample data to validate the Chi-Square test, averaging the results. We got a p-value of $\approx 1.79\text{e-}14$. Such a low p-value (far below the conventional threshold of 0.05) indicates that the observed association is not a product of random chance. Instead, it implies a strong dependency between external factors contributing to bugs and the specific bug patterns within deep learning systems. Our manual analysis also supports the hypothesis, showing a higher prevalence of extrinsic bugs in deep learning systems (DLSW) compared to non-deep learning systems

(NDLSW) (Fig. 10). The prevalence ratio of extrinsic bugs varies from 21.90% to 82.35% across different types of deep-learning bugs, confirming a strong association between extrinsic factors and bugs from deep-learning systems. Our experiments also suggest that extrinsic bugs might have an underlying connection with deep-learning bugs (refer to RQ$_3$: Localization of extrinsic & intrinsic bugs from both systems), which contributes to the poor performance of the existing bug localization techniques.

Summary of RQ$_3$: We found that deep learning systems (DLSW) contain almost *four times* more extrinsic bugs than non-deep learning systems (NDLSW). The performance of our baseline bug localization techniques for extrinsic bugs is lower (e.g., **31.27%** less MAP for DNNLOC) compared to that of intrinsic bugs. Our research also shows a strong connection between extrinsic bugs and bugs in deep learning systems.

4.4 Key findings

Our empirical results reveal several key challenges that hinder bug localization in deep learning systems. All five baseline techniques (BugLocator [11], BLUiR [12], BLIA [23], DNNLOC [17], bjXnet [30]) performed significantly worse with deep learning systems than traditional software systems (e.g., ~30% lower MAP on average), confirming a substantial performance gap. Through our three research questions, we identified the following factors contributing to this performance gap:

1. Dominance of Extrinsic Bugs

Challenge: Deep learning systems contain a significantly higher number of extrinsic bugs. Extrinsic bugs are issues triggered by external factors such as hardware-software compatibility, operating systems, third-party libraries, or training data [37]. We found nearly four times more extrinsic bugs in deep learning systems than in traditional ones. Since these bugs stem from outside the source code (e.g., GPU driver incompatibilities), they often escape code-centric bug localization methods. All five baseline techniques from our study struggled to locate the extrinsic bugs. GPU bugs were among the hardest to localize, with all our techniques performing the worst in this category (e.g., MAP below 0.30 across all the baselines). The complex interplay of deep learning system layers (framework ¿ CUDA library ¿ GPU hardware) [91] makes their localization challenging. Using the Chi-Square test, we also found a strong statistical correlation (e.g., a p-value of ~1.79e-14) between deep learning projects and extrinsic bugs.

Implication: Existing bug localization approaches mainly focus on source code analysis and thus miss critical extrinsic bugs. Appropriate tools and processes capturing external factors and environmental context will be needed to address extrinsic bugs in deep learning systems. Integrating keyword search or stack trace pattern analysis [92] into bug triage might help identify externally

triggered failures. Similarly, categorizing extrinsic bugs into distinct classes – issues related to third-party libraries (e.g., version mismatches), hardware-related problems (e.g., GPU bugs), operating system incompatibilities, and dataset validity – can facilitate a more systematic localization process. For example, localizing hardware-specific extrinsic bugs may greatly benefit from targeted environmental checks, utilizing tools for GPU profiling [93] (e.g., NVIDIA Visual Profiler[10] and PyTorch Profiler[11]).

2. Limitations of Textual Representation

Challenge: Many existing techniques for bug localization (e.g., IR-based) heavily depend on textual similarity between bug reports and source code, which might not be sufficient for localizing complex, deep-learning bugs. For example, bugs related to gradient vanishing require a deeper conceptual understanding of neural network behaviour beyond what textual descriptions in bug reports typically convey (e.g., 'the model is not learning'). Similarly, the bug reports could contain highly abstract issues (e.g., incorrect model architectures, incorrect loss function), which warrant the knowledge of the model architecture, hyperparameters, loss function, training procedure, and dataset. Our study revealed that existing techniques significantly struggled with such nuanced cases. Although the bug reports for deep learning systems might contain more code snippets compared to traditional software systems, these snippets did not substantially enhance bug localization, according to our findings.

Implication: Given the abstract concepts in bug reports and the complexities in many deep learning bugs, future localization methods or tools should carefully leverage domain-specific knowledge [6]. In particular, a comprehensive understanding of neural network concepts (e.g., layers, gradients, loss functions), common framework usage patterns, and data-processing pipelines should be injected into these tools. For example, the bug report stating 'the model isn't learning' was traced back to the vanishing gradient problem caused by multiple sigmoid layers. A domain-aware technique might recognize this underlying connection and point toward these sigmoid layers. Another way to tackle the challenge is to incorporate rich contextual information into bug reports. Essential details (e.g., model configurations, hyperparameters, dataset characteristics, and runtime conditions) in the bug report might improve bug localization since they can provide critical information to map the errors with the root cause.

3. Dynamic Complexity during Training

Challenge: Bugs emerging during model training represent another major challenge. We identified training bugs as the most prevalent bug type (∼27%) in deep learning systems. They involve dynamic behaviors such as model convergence and iterative parameter updates. Our baseline techniques

---

[10] https://developer.nvidia.com/nvidia-visual-profiler
[11] https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

often struggle to locate these bugs since they emerge from sequences of training steps and are not visible within the code. For example, a misconfigured batch size might only reveal itself after several epochs of training, rendering traditional, code-centric bug localization ineffective.

Implication: To effectively localize bugs manifesting during the model training phase, future techniques should incorporate dynamic analysis along with static analysis [7]. By capturing runtime information and associating it with the relevant source code, training-phase bugs, especially silent bugs, might be localized and diagnosed.

4. Lack of a Tailored Approach

Challenge: Our study indicates that no single technique performs consistently across all bug categories in deep learning systems. Techniques based on information retrieval (e.g., BLUiR) often perform well when textual cues in a bug report map directly to the source code (e.g., API bugs). On the other hand, deep learning–based approaches (e.g., DNNLOC, bjXnet) can more effectively address conceptual bugs (e.g., model bugs). However, neither approach fully addresses a major subset of bugs from deep learning systems, ranging from hardware compatibility problems to complex model-level inconsistencies, exposing the limitations of a 'one-size-fits-all' paradigm.

Implication: Unlike traditional software systems, different bug types in deep learning systems might benefit from different localization strategies. Future work should explore more holistic and adaptive techniques. Rather than relying on a single method, researchers could identify specific bug contexts (e.g., model, GPU, or API bug) and then apply appropriately tailored localization methods.

5 Threats to Validity

We identify a few threats to the validity of our findings. In this section, we discuss these threats and the necessary steps taken to mitigate them as follows.

Threats to internal validity relate to experimental errors and human biases [94]. Traditional bug tracking systems (e.g., Bugzilla, GitHub, Jira) contain thousands of bug reports, and their quality cannot be guaranteed. This could be a source of threat as the bug reports are used as queries to locate the buggy files. Bug reports often contain poor, insufficient, missing, or even inaccurate information [95]. Hence, we used data from existing benchmarks ([32, 33], where the authors took necessary steps to avoid low-quality or invalid bug reports. Thus, such threats might be mitigated.

Another potential source of threat could be the replication of existing work. The original replication package was unavailable; hence, we used the publicly available version of BugLocator, BLUiR [54], and DNNLOC [65]. For BLIA and bjXnet, we reused the author's replication package [23, 30]. We validated our implementation of the existing methods using their original dataset

and achieved comparable results (e.g., with differences $\approx$ 2.00%–3.00% using MAP).

Threats to conclusion validity. The observations from our study and the conclusions we drew from them could be a source of threat to conclusion validity [96]. In this research, we answer three research questions using two different datasets and re-implement five existing techniques. We use appropriate statistical tests (e.g., t-test) and report the test details (e.g., p-value, Cohen's D) to conclude. Thus, such threats might also be mitigated.

Threats to construct validity relate to the use of appropriate performance metrics. We evaluate all the methodologies using MRR, MAP, and Top@K, which have been used widely by the related work [11, 12, 14, 17, 53, 19, 60]. Thus, such threats might also be mitigated.

6 Related Work

6.1 Software bug

Understanding the nature and characteristics of bugs is essential for effective debugging and testing. They can differ across different programming languages and development frameworks [97]. Over the last 50 years, hundreds of studies have been conducted to tackle bugs in traditional software systems. Recently, bugs from deep learning systems have garnered much attention due to their great interest and significance. Humbatova et al. [31] proposed a taxonomy of bugs from deep learning systems with five main categories - model, training, tensor & input, API, and GPU. Chen et al. [7] focused on the unique obstacles for deep learning-based software deployment. According to Islam et al. [36], data bugs and logic bugs are the most severe in deep-learning software systems. Another study by Islam et al. [98] showed that the bugs or repair patterns of deep learning models significantly differ from those of traditional systems. As a result, traditional software debugging approaches, such as bug localization techniques, might not be effective for deep learning systems. Therefore, empirical research like ours, which focuses on the challenges posed by deep learning bugs in the software debugging process (specifically in bug localization), is essential.

6.2 Spectrum-based bug localization

Spectrum-based bug localization techniques have been widely studied for bug localization in traditional software systems. These techniques analyze program traces from passing and failing tests to measure how frequently each program element is executed in both cases [78]. Based on these frequencies, spectrum-based techniques rank program elements by their suspiciousness, with the aim of prioritizing fault-prone locations. For example, Tarantula [55] is a widely known spectrum-based localization technique that calculates suspiciousness

by measuring the ratio of failed to total executions of a particular element, highlighting the ones more frequently associated with failures.

However, capturing execution traces for spectrum-based approaches is not as straightforward in deep learning systems. One of the main reasons for this is that not all bugs in DL systems result in crashes or stack traces [36]. Instead, many DL system bugs manifest as silent bugs [57], leading to incorrect but seemingly valid behavior, making it difficult to generate program traces that are traditionally used in spectrum-based techniques. Moreover, the non-deterministic and data-driven nature of DL models further complicates the application of spectrum-based bug localization techniques. For instance, Deep-Fault [56] attempts to apply spectrum-based techniques to Deep Neural Networks (DNNs) by focusing on neuron activation spectra to identify suspicious neurons. While this approach is promising, it fails to handle more complex bugs, such as silent bugs or those caused by external factors, including GPU-related issues.

Another major challenge of applying spectrum-based bug localization techniques to deep learning systems is the resource-intensive nature of these systems. DL models are computationally expensive to train and execute, requiring substantial hardware resources, such as GPUs and large memory capacities. The overhead of collecting and processing execution traces for spectrum-based approaches can, therefore, become prohibitive, particularly in large-scale DL projects. Given these limitations, spectrum-based bug localization techniques are not well-suited for our study, which focuses on localizing bugs in DL systems.

6.3 Information Retrieval-based bug localization

One of the crucial steps toward fixing a software bug is to detect its location within the software code. Many existing approaches [11, 12, 14, 13, 23] use Information Retrieval (IR) to locate bugs by matching keywords between a query and the source code.

Zhou et al. [11] introduce BugLocator, which leverages textual similarity between bug reports and source code using rVSM for bug localization. Saha et al. [12] propose BLUiR, which determines the textual similarity between source code and bug reports using the Okapi-BM25 algorithm [99]. BLUiR also leverages structural items from both bug reports and source code, which boosts its localization performance. Later, Wang and Lo [14] propose AmaLgam, which incorporates the textual similarity from BugLocator, structured items from BLUiR, and version control history into IR-based bug localization.

Wang et al. [100] analyzed IR-based fault localization techniques and found their effectiveness to be limited, mainly due to the frequent unavailability of high-quality bug reports. The quality of bug reports makes it challenging to localize bugs using traditional IR-based techniques. Rahman and Roy [13] propose BLIZZARD, which leverages the quality aspect of bug reports and introduces context-aware query reformulation into bug localization. Wong

et al. [53] proposed BRTracer, which improves upon BugLocator by combining source document segmentation and stack-trace analysis.

Le et al. [101] used an automated method to predict the effectiveness of IR-based bug localization by leveraging features extracted from bug reports and localization methods, with their findings focusing on the significance of metadata features (e.g., commit history, stack traces) in enhancing the performance of these techniques. Another technique, namely Locus [60], uses the software change information from commit logs and change histories to improve bug localization. Youm et al. [23] proposed BLIA, which integrates bug reports, structured information of source files, and source code change history. It localizes bugs in two granularity levels - file level and method level – and outperforms prior approaches.

All these IR-based approaches have been designed with a focus on traditional software bugs. Bugs in deep learning applications pose several unique challenges: (a) non-deterministic behavior due to factors like random initialization and stochastic optimization [102], (b) complex relationships between high-dimensional data and model behavior and the influence of data-specific issues without direct code-level manifestations [36], (c) strong external dependencies on hardware (e.g., PyTorch leverages GPU) [10]. Although IR-based bug localization techniques have shown promising results in traditional software systems, their performance might decline while localizing bugs in deep learning systems. Our experiments also show relevant evidence to support this observation. Please check Section 4 for further details on our experiments.

Recently, Kim et al. [22] used basic IR-based techniques (e.g., VSM, rVSM, BM25) for locating bugs in deep learning systems but reported poor performance without any comprehensive analysis or explanation. Thus, the potential of existing IR-based solutions for bugs in deep-learning applications is not well understood yet. Our work in this article fills in that significant gap in the literature.

To the best of our knowledge, no IR-based techniques have been developed exclusively for localizing bugs in DL systems. Existing approaches for bug detection in DL systems, such as DeepLocalize [58], DeepDiagnosis [20], and DeepFD [21], focus on specific types of DL bugs (e.g., model bugs, training bugs) and rely on dynamic analysis or heuristic-based methods. While these approaches are valuable, they are limited in terms of scope, scalability, and generalizability. They cannot be applied to non-DL bugs. For example, DeepLocalize is tightly coupled with the Keras library and exhibits low accuracy in bug detection. DeepDiagnosis depends on hard-coded rules, restricting its adaptability across different contexts. Moreover, these techniques are highly specific to DL models and cannot be adapted to traditional software bugs, making direct comparisons infeasible. On the other hand, IR-based techniques like BugLocator, BLUiR, and BLIA are more general-purpose and can be applied to both DL and non-DL systems.

By including traditional IR-based and recent deep learning-based baselines in our study, we attempt to identify the strengths and limitations of these techniques when applied to DL systems.

6.4 Deep learning-based bug localization

Unlike the above IR-based methods, deep learning can detect non-linear relationships between bug reports and source code for bug localization [26, 27, 28]. Polisetty et al. [103] evaluated deep learning-based bug localization models against traditional machine learning (ML) models, finding that while deep neural network (DNN) models generally outperform conventional ML models in performance, they require substantial resources such as GPUs and memory. Lam et al. [17] propose DNNLOC combining with information retrieval (e.g., rVSM [11]) and deep learning for bug localization. Xiao et al. [19] propose DeepLocator, where they use CNN and AST to extract features from bug reports and source documents, respectively. To learn unified features from natural language and source code during bug localization, Huo et al. [104] propose NP-CNN, which integrates both lexical and program structure information. Liang et al. [105] propose CAST, combining a tree-based CNN (TB-CNN) with customized AST to locate buggy files. However, these deep learning-based techniques are developed and evaluated using the source code from traditional software systems (e.g., JDT, SWT, Tomcat, AspectJ). These software systems do not represent deep learning applications, and thus, the designed techniques above might not be sufficient to tackle all the challenges of deep learning-related bugs.

Wardat et al. [18] propose an approach to locate Deep Neural Network (DNN) bugs through dynamic and statistical analysis. However, their method's sole focus on model and training bugs, low accuracy, and over-reliance on the Keras library pose challenges for practical adoption. Deep learning-based approaches also lack explainability and heavily rely on source code, which may not be sufficient for the bugs with external dependencies (a.k.a extrinsic bugs) in deep learning applications.

Additionally, Cao et al. [59] characterize the performance problems in deep learning (DL) systems and identify key stages where these problems are introduced and exposed. They highlight the unique challenges of detecting and localizing bugs in DL systems, particularly due to dependencies on GPUs and external libraries like TensorFlow and Keras. However, their focus is limited to performance problems related to hardware dependencies, overlooking broader system components such as DNN model architecture and training dynamics. For example, they do not address how issues like vanishing gradients during model training or incorrect hyperparameter configurations (e.g., learning rate or batch size) can also lead to performance degradation in deep learning systems. Therefore, empirical research like ours, which focuses on the challenges posed by deep learning bugs in the software debugging process (specifically in bug localization), is essential.

To address the above gap, in this empirical study, we replicated five existing techniques [11, 12, 23, 17, 30] to locate bugs in deep learning systems. Unlike Kim et al. [22], our study extends beyond bug localization from deep learning systems. Our study evaluates existing bug localization techniques, categorizes deep-learning bugs, analyzes their prevalence and challenges, and assesses each

technique's effectiveness for different bug types. We also conduct extensive manual analysis and explain they are difficult to localize (e.g., extrinsic factors, multifaceted dependencies), which makes our work *novel*.

7 Conclusion

Identifying the location of a bug within a software system (a.k.a. bug localization) is crucial to correct any bug. In recent years, bug localization techniques have received considerable attention in the context of traditional software systems. However, they might not be sufficient for deep learning systems as deep learning bugs pose a greater challenge due to their multifaceted dependencies. However, the potential of existing approaches for localizing bugs in deep learning systems is not well understood to date. In this work, we replicated five existing bug localization approaches and found that they show poor performance in localizing bugs from deep-learning systems. Secondly, through an in-depth analysis, we found that localizing certain categories of bugs (e.g., training bugs & GPU bugs) is more difficult than other bugs in deep learning systems. Finally, we investigate and find that deep learning bugs are more likely to be extrinsic, i.e., connected to non-code artifacts (e.g., training data). Our research thus offers empirical evidence and actionable insights for deep learning software bugs, advancing automated software debugging research. Future work can focus on developing a new framework for automated software debugging based on the insights from this empirical study.

Statement and Declarations

Conflicts of Interests
The authors declare they have no conflict of interest.

Ethical Approval
Not applicable.

Informed Consent
Not applicable.

Author Contributions
Conceptualization: Sigma Jahan; Methodology: Sigma Jahan; Formal analysis and investigation: Sigma Jahan; Writing—original draft preparation: Sigma Jahan; Writing—review & editing: Sigma Jahan; Resources: Sigma Jahan, Mohammad Masudur Rahman; Supervision: Mohammad Masudur Rahman;

Data annotation: Sigma Jahan, Mehil B Shah; Replication of baseline technique: Sigma Jahan, Mehil B Shah.

Data Availability Statement

The dataset and supplementary materials are publicly available at `https://doi.org/10.5281/zenodo.15091574`.

References

1. A. Arcuri. On the automation of fixing software bugs. In *ICSE*, pages 1003–1006, 2008.
2. R. M. Karampatsis and C. Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *MSR*, pages 573–577, 2020.
3. J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. ICSE*, pages 361–370, 2006.
4. Consortium for Information & Software Quality (CISQ). The cost of poor quality software in the us: A 2022 report. `https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/`, 2022. Accessed: 2024-Jan-24.
5. J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.
6. W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. How practitioners perceive automated bug report management techniques. *IEEE TSE*, 46 (8):836–862, 2020.
7. Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu. A comprehensive study on challenges in deploying deep learning-based software. In *ESEC/FSE*, pages 750–762, 2020.
8. S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *ICSE-SEIP*, pages 291–300, 2019.
9. D. Gonzalez, T. Zimmermann, and N. Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *MSR*, pages 431–442, 2020.
10. L. Nganyewou Tidjon, B. Rombaut, F. Khomh, and A. E. Hassan. An empirical study of library usage and dependency in deep learning frameworks. *arXiv e-prints*, pages arXiv–2211, 2022.
11. J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.
12. R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.
13. M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *ESEC/FSE*, pages 621–632, 2018.
14. S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, pages 53–63, 2014.
15. L. Moreno, J. Treadway, J, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *ICSME*, pages 151–160, 2014.
16. A. Perez, R. Abreu, and A. Riboira. A dynamic code coverage approach to maximize fault localization efficiency. *Journal of Systems and Software*, 90:18–28, 2014.

17. A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *ICPC*, pages 218–229, 2017.
18. M. Wardat, W. Le, and H. Rajan. Deeplocalize: Fault localization for deep neural networks. In *ICSE*, pages 251–262, 2021.
19. Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin. Improving bug localization with an enhanced convolutional neural network. In *APSEC*, pages 338–347, 2017.
20. M. Wardat, B. D. Cruz, W. Le, and H. Rajan. Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *ICSE*, pages 561–572, 2022.
21. J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S. C. Cheung. Deepfd: Automated fault diagnosis and localization for deep learning programs. In *ICSE*, pages 573–585, 2022.
22. M. Kim, Y. Kim, and E. Lee. An empirical study of ir-based bug localization for deep learning-based software. In *ICST*, pages 128–139, 2022.
23. K. C. Youm, J. Ahn, and E. Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.
24. M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *ESEC/FSE*, pages 621–632, 2018.
25. I. Chawla and S. K. Singh. Performance evaluation of vsm and lsi models to determine bug reports similarity. In *Proc. IC3*, pages 375–380, 2013.
26. O. Obulesu, M. Mahendra, and M. ThrilokReddy. Machine learning techniques and tools: A survey. In *ICIRCA*, pages 605–611, 2018.
27. J. S. Almeida. Predictive non-linear modeling of complex data by artificial neural networks. *Current opinion in biotechnology*, 13(1):72–76, 2002.
28. Z. Bitvai and T. Cohn. Non-linear text regression with a deep convolutional neural network. In *ACL*, pages 180–185, 2015.
29. L. Deng and Y. (Eds.) Liu. *Deep learning in natural language processing*. Springer, 2018.
30. J. Han, C. Huang, S. Sun, Z. Liu, and J. Liu. bjxnet: an improved bug localization model based on code property graph and attention mechanism. *Automated Software Engineering*, 30(1):12, 2023.
31. N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *ICSE*, pages 1110–1121, 2020.
32. M. Kim, Y. Kim, and E. Lee. Denchmark: A bug benchmark of deep learning-related software. In *MSR*, pages 540–544, 2021.
33. S. Muvva, A. E. Rao, and S. Chimalakonda. Bugl–a cross-language dataset for bug localization. *arXiv preprint arXiv:2004.08846*, 2020.
34. T. K. Kim. T test as a parametric statistic. *Korean Journal of Anesthesiology*, 68(6):540, 2015.
35. M. E. Rice and G. T. Harris. Comparing effect sizes in follow-up studies: Roc area, cohen's d, and r. *Law and human behavior*, 29:615–620, 2005.

36. M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *ESEC/FSE*, pages 510–520, 2019.

37. G. Rodriguez-Perez, M. Nagappan, and G. Robles. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE TSE*, 2020.

38. Facebook AI Research. Issue 1860. `https://github.com/facebookresearch/fairseq/issues/1860`, 2024. Accessed: 2024-01-30.

39. PyTorch Ignite Team. Issue 1426. `https://github.com/pytorch/ignite/issues/1426`, 2024. Accessed: 2024-01-30.

40. ASYML Texar-PyTorch Team. Issue 313. `https://github.com/asyml/texar-pytorch/issues/313`, 2024. Accessed: 2024-01-30.

41. FastAI Team. Issue 3048. `https://github.com/fastai/fastai/issues/3048`, 2024. Accessed: 2024-01-30.

42. Apache MXNet Team. Issue 13760. `https://github.com/apache/mxnet/issues/13760`, 2024. Accessed: 2024-01-30.

43. Apache MXNet Team. Issue 13862. `https://github.com/apache/mxnet/issues/13862`, 2024. Accessed: 2024-01-30.

44. Keras Team. Issue 1238. `https://github.com/keras-team/autokeras/issues/1238`, 2024. Accessed: 2024-01-30.

45. Apache MXNet Team. Issue 10224. `https://github.com/apache/mxnet/issues/10224`, 2024. Accessed: 2024-01-30.

46. G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. López G., I. Heredia, P. Malík, and L. Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52:77–124, 2019.

47. S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun. Deep learning tools and libraries. In *A Guide to Convolutional Neural Networks for Computer Vision*, pages 159–167. Springer, 2018.

48. Z. Wang, K. Liu, J. Li, Y. Zhu, and Y. Zhang. Various frameworks and libraries of machine learning and deep learning: a survey. *Archives of computational methods in engineering*, pages 1–24, 2019.

49. TensorFlow Team. Issue 5596. `https://github.com/tensorflow/tensorboard/issues/5596`, 2024. Accessed: 2024-01-30.

50. B. J Erickson, P. Korfiatis, Z. Akkus, T. Kline, and K. Philbrick. Toolkits and libraries for deep learning. *Journal of Digital Imaging*, 30:400–405, 2017.

51. P. A. Hernandez, C. H. Graham, L. L. Master, and D. L. Albert. The effect of sample size and species characteristics on performance of different species distribution modeling methods. *Ecography*, 29(5):773–785, 2006.

52. A. S. Acharya, A. Prakash, P. Saxena, and A. Nigam. Sampling: Why and how of it. *Indian Journal of Medical Specialties*, 4(2):330–333, 2013.

53. C. P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *ICSME*, pages 181–190, 2014.

54. J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In
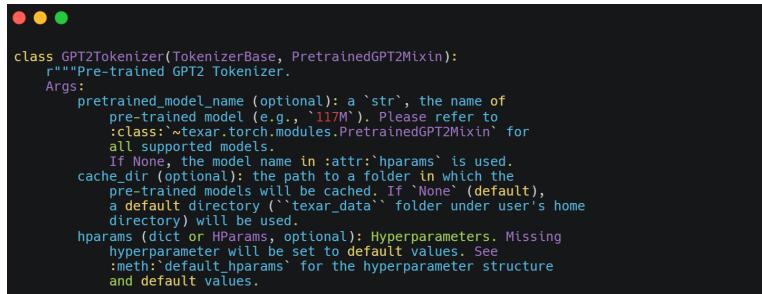
ISSTA, pages 61–72, 2018.

55. James A. Jones and Mary J. H. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2005.

56. H. F. Eniser, Simos G., and Alper S. Deepfault: Fault localization for deep neural networks. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2019.

57. F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol. Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow. *Empirical Software Engineering*, 29(1):10, 2024.

58. Mohammad W., Wei L., and Hridesh R. Deeplocalize: Fault localization for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

59. 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*, pages 357–369, 2022.

60. M. Wen, R. Wu, and S. C. Cheung. Locus: Locating bugs from software changes. In *ASE*, pages 262–273, 2016.

61. 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*, pages 590–604. IEEE, 2014.

62. V. Ashish. Attention is all you need. *Advances in neural information processing systems*, 30:I, 2017.

63. Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 105:17–29, 2019.

64. B. Qi, H. Sun, W. Yuan, H. Zhang, and X. Meng. Dreamloc: A deep relevance matching-based framework for bug localization. *IEEE Transactions on Reliability*, 71(1):235–249, 2021.

65. D. Emre and C. Alperen. `https://github.com/emredogan7/bug-localization-by-dnn-and-rvsm`, 2019.

66. Replication Package. `https://bit.ly/BL-Challenges-DLSW`, 2023.

67. Mohammad R. and Carlo A. F. An empirical study of fault localization in python programs. *Empirical Software Engineering*, 29(4):92, 2024.

68. S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE TSE*, 48(5):1480–1496, 2020.

69. P. Royston. Approximating the shapiro-wilk w-test for non-normality. *Statistics and computing*, 2(3):117–119, 1992.

70. R. Rosenthal, H. Cooper, and L. Hedges. Parametric measures of effect size. *The handbook of research synthesis*, 621(2):231–244, 1994.

71. D. C. Howell. *Statistical Methods for Psychology*. Wadsworth Publishing, 8th edition, 2012.

72. C. Ebert and M. Ernst. Cyclomatic complexity. *IEEE software*, 33(6): 27–29, 2016.

73. T. Hariprasad and R. Balamurugan. Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 758–761. IEEE, 2017.

74. N. M. A. M. Najm. Measuring maintainability index of a software depending on line of code only. *IOSR J. Comput. Eng.*, 16(2):64–69, 2014.

75. PyTorch Team. Issue 87085. `https://github.com/pytorch/pytorch/issues/87085`, 2024. Accessed: 2024-01-30.

76. TensorFlow Team. Issue 61297. `https://github.com/tensorflow/tensorflow/issues/61297`, 2024. Accessed: 2024-01-30.

77. TensorFlow Team. Issue 5948. `https://github.com/tensorflow/tensorboard/issues/5948`, 2024. Accessed: 2024-01-30.

78. S. A Akbar and A. C Kak. A large-scale comparative evaluation of ir-based tools for bug localization. In *MSR*, pages 21–31, 2020.

79. Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. Deepfault: Fault localization for deep neural networks. In *International Conference on Fundamental Approaches to Software Engineering*, Cham, 2019. Springer International Publishing.

80. D. Zhuang, X. Zhang, S. Song, and S. Hooker. Randomness in neural network training: Characterizing the impact of tooling. In *Proceedings of Machine Learning and Systems*, volume 4, pages 316–336, 2022.

81. B. Ma, F. Li, Y. Zhang, W. Xu, and Y. Chen. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (IC-CAD)*, 2018.

82. K. Tian, Y. Zhang, and S. Neogi. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

83. P. F. Watson and A Petrie. Method agreement analysis: a review of correct methodology. *Theriogenology*, 73(9):1167–1179, 2010.

84. H. Aoyama. A study of stratified random sampling. *Ann. Inst. Stat. Math*, 6(1):1–36, 1954.

85. M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner. On debugging the performance of configurable software systems: Developer needs and tailored tool support. In *ICSE*, pages 1571–1583, 2022.

86. T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim. An empirical study of common challenges in developing deep learning applications. In *IEEE ISSRE*, pages 104–115, 2019.

87. D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

88. N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh. Deep learning vs. traditional computer vision. In *CVC*, pages 128–144, 2020.

89. N. Karasov, A. Khvorov, R. Vasiliev, Y. Golubev, and T. Bryksin. Aggregation of stack trace similarities for crash report deduplication. *arXiv preprint arXiv:2205.00212*, 2022.

90. M. L. McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

91. Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012.

92. Korosh Koochekian Sabor, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj. Automatic prediction of the severity of bugs using stack traces and categorical features. *Information and Software Technology*, 123:106205, 2020.

93. Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pinar Tozun. Profiling and monitoring deep learning training tasks. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*, pages 18–25, 2023.

94. Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.

95. S. Gupta and S. K. Gupta. A systematic study of duplicate bug report detection. *International Journal of Advanced Computer Science and Applications*, 12(1), 2021.

96. M. A. García-Pérez. Statistical conclusion validity: Some common threats and simple remedies. *Frontiers in psychology*, 3:325, 2012.

97. R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *ESEC/FSE*, pages 1556–1560, 2020.

98. M. J. Islam, R. Pan, G. Nguyen, and H. Rajan. Repairing deep neural networks: Fix patterns and challenges. In *ICSE*, pages 1135–1146, 2020.

99. S. Robertson and H. Zaragoza. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

100. Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *ISSTA*, pages 1–11, 2015.

101. T. Bs Le, F. Thung, and D. Lo. Predicting effectiveness of ir-based bug localization techniques. In *IEEE ISSRE*, pages 335–345, 2014.

102. M. Sajjadi, M. Javanmardi, and T. Tasdizen. Regularization with stochastic transformations and perturbations for deep semi-supervised learning. *Advances in Neural Information Processing Systems*, 29, 2016.

103. S. Polisetty, A. Miranskyy, and A. Başar. On usefulness of the deep-learning-based bug localization models to practitioners. In *PROMISE*, pages 16–25, 2019.

104. X. Huo, M. Li, and Z. H. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, volume 16, pages 1606–1612, 2016.

105. H. Liang, L. Sun, M. Wang, and Y. Yang. Deep learning with customized abstract syntax tree for bug localization. *IEEE Access*, 7:116309–116320, 2019.

A Appendix

```
class GPT2Tokenizer(TokenizerBase, PretrainedGPT2Mixin):
    r"""Pre-trained GPT2 Tokenizer.
    Args:
        pretrained_model_name (optional): a `str`, the name of
            pre-trained model (e.g., `117M`). Please refer to
            :class:`~texar.torch.modules.PretrainedGPT2Mixin` for
            all supported models.
            If None, the model name in :attr:`hparams` is used.
        cache_dir (optional): the path to a folder in which the
            pre-trained models will be cached. If `None` (default),
            a default directory (``texar_data`` folder under user's home
            directory) will be used.
        hparams (dict or HParams, optional): Hyperparameters. Missing
            hyperparameter will be set to default values. See
            :meth:`default_hparams` for the hyperparameter structure
            and default values.
```

Fig. 12: Code snippet of the ground truth for the model bug [40] (`https://bit.ly/3XJCdqq`)

```
class SentencePieceTokenizerTest(unittest.TestCase):

    def setUp(self):
        self.tmp_dir = tempfile.TemporaryDirectory()
        self.SAMPLE_VOCAB = maybe_download(
            'https://github.com/google/sentencepiece/blob/master/'
            'python/test/test_model.model?raw=true', self.tmp_dir.name)

        self.tokenizer = SentencePieceTokenizer.load(self.SAMPLE_VOCAB)

        self.tokenizer.save(self.tmp_dir.name)

    def tearDown(self):
        self.tmp_dir.cleanup()

    def test_load(self):
        tokenizer = SentencePieceTokenizer.load(self.tmp_dir.name)
```

Fig. 13: Code snippet of the incorrect source file for the model bug retrieved by BugLocator (`https://bit.ly/43klnj9`)

```python
class Seq2Seq(nn.Module):

    def __init__(self, train_data):
        super().__init__()

        self.source_vocab_size = train_data.source_vocab.size
        self.target_vocab_size = train_data.target_vocab.size

        self.bos_token_id = train_data.target_vocab.bos_token_id
        self.eos_token_id = train_data.target_vocab.eos_token_id

        self.source_embedder = tx.modules.WordEmbedder(
            vocab_size=self.source_vocab_size,
            hparams=config_model.embedder)

        self.target_embedder = tx.modules.WordEmbedder(
            vocab_size=self.target_vocab_size,
            hparams=config_model.embedder)

        self.encoder = tx.modules.BidirectionalRNNEncoder(
            input_size=self.source_embedder.dim,
            hparams=config_model.encoder)

        self.decoder = tx.modules.AttentionRNNDecoder(
            token_embedder=self.target_embedder,
            encoder_output_size=(self.encoder.cell_fw.hidden_size
+
self.encoder    input_size=self.target_embedder.dim,
            vocab_size=self.target_vocab_size,
            hparams=config_model.decoder)
```
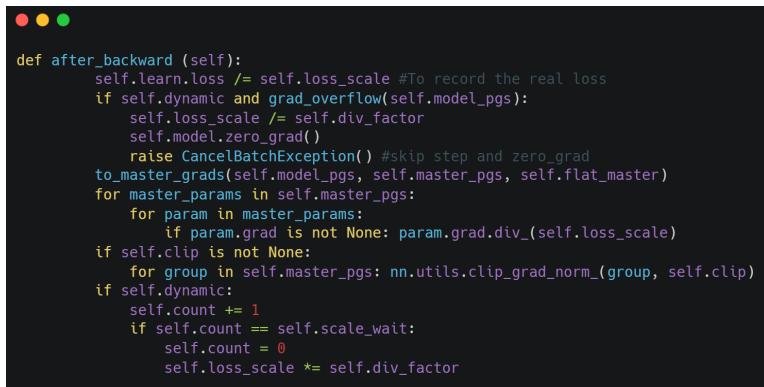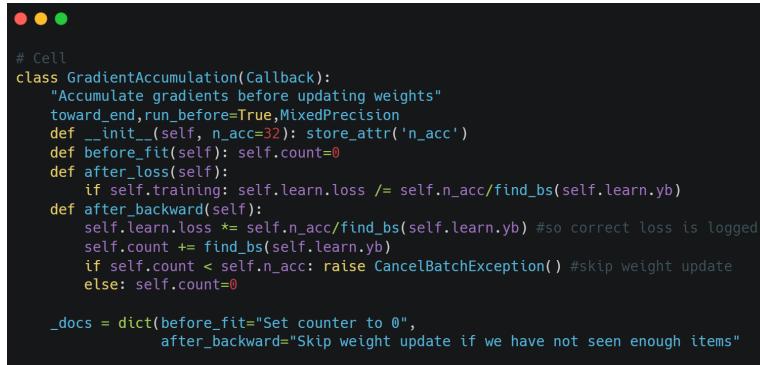
Fig. 14: Code snippet of the incorrect source file for the model bug retrieved by BLUiR (https://bit.ly/3PQo88U)

```python
def after_backward (self):
        self.learn.loss /= self.loss_scale #To record the real loss
        if self.dynamic and grad_overflow(self.model_pgs):
            self.loss_scale /= self.div_factor
            self.model.zero_grad()
            raise CancelBatchException() #skip step and zero_grad
        to_master_grads(self.model_pgs, self.master_pgs, self.flat_master)
        for master_params in self.master_pgs:
            for param in master_params:
                if param.grad is not None: param.grad.div_(self.loss_scale)
        if self.clip is not None:
            for group in self.master_pgs: nn.utils.clip_grad_norm_(group, self.clip)
        if self.dynamic:
            self.count += 1
            if self.count == self.scale_wait:
                self.count = 0
                self.loss_scale *= self.div_factor
```

Fig. 15: Code snippet of the ground truth for training bug [41] (https://bit.ly/3NKfRAH)
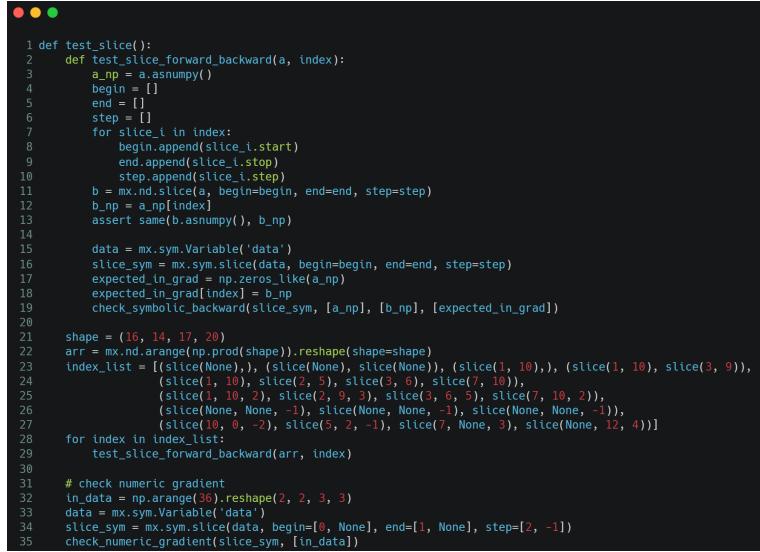
```
# Cell
class GradientAccumulation(Callback):
    "Accumulate gradients before updating weights"
    toward_end,run_before=True,MixedPrecision
    def __init__(self, n_acc=32): store_attr('n_acc')
    def before_fit(self): self.count=0
    def after_loss(self):
        if self.training: self.learn.loss /= self.n_acc/find_bs(self.learn.yb)
    def after_backward(self):
        self.learn.loss *= self.n_acc/find_bs(self.learn.yb) #so correct loss is logged
        self.count += find_bs(self.learn.yb)
        if self.count < self.n_acc: raise CancelBatchException() #skip weight update
        else: self.count=0

    _docs = dict(before_fit="Set counter to 0",
                 after_backward="Skip weight update if we have not seen enough items"
```

Fig. 16: Code snippet of the incorrect source file for the training bug retrieved by BugLocator (https://bit.ly/3O5yEIa)
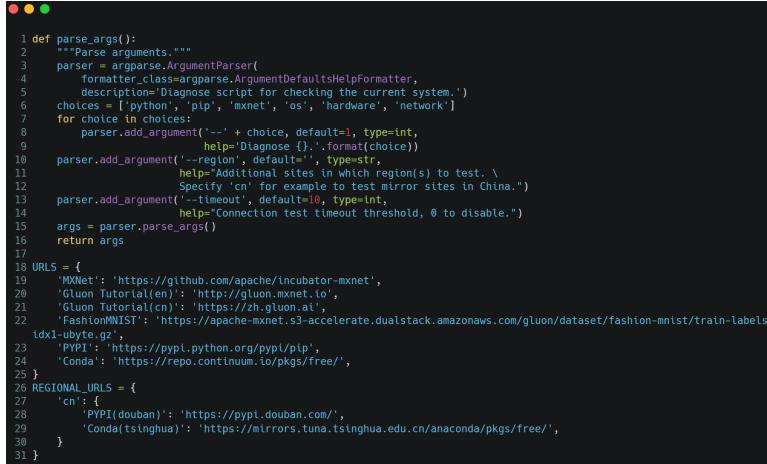
```
1 def test_slice():
2     def test_slice_forward_backward(a, index):
3         a_np = a.asnumpy()
4         begin = []
5         end = []
6         step = []
7         for slice_i in index:
8             begin.append(slice_i.start)
9             end.append(slice_i.stop)
10            step.append(slice_i.step)
11        b = mx.nd.slice(a, begin=begin, end=end, step=step)
12        b_np = a_np[index]
13        assert same(b.asnumpy(), b_np)
14
15        data = mx.sym.Variable('data')
16        slice_sym = mx.sym.slice(data, begin=begin, end=end, step=step)
17        expected_in_grad = np.zeros_like(a_np)
18        expected_in_grad[index] = b_np
19        check_symbolic_backward(slice_sym, [a_np], [b_np], [expected_in_grad])
20
21    shape = (16, 14, 17, 20)
22    arr = mx.nd.arange(np.prod(shape)).reshape(shape=shape)
23    index_list = [(slice(None),), (slice(None), slice(None)), (slice(1, 10),), (slice(1, 10), slice(3, 9)),
24                  (slice(1, 10), slice(2, 5), slice(3, 6), slice(7, 10)),
25                  (slice(1, 10, 2), slice(2, 9, 3), slice(3, 6, 5), slice(7, 10, 2)),
26                  (slice(None, None, -1), slice(None, None, -1), slice(None, None, -1)),
27                  (slice(10, 0, -2), slice(5, 2, -1), slice(7, None, 3), slice(None, 12, 4))]
28    for index in index_list:
29        test_slice_forward_backward(arr, index)
30
31    # check numeric gradient
32    in_data = np.arange(36).reshape(2, 2, 3, 3)
33    data = mx.sym.Variable('data')
34    slice_sym = mx.sym.slice(data, begin=[0, None], end=[1, None], step=[2, -1])
35    check_numeric_gradient(slice_sym, [in_data])
```

Fig. 17: Code snippet of the ground truth for the tensor bug [42] (https://bit.ly/3XLe82y)
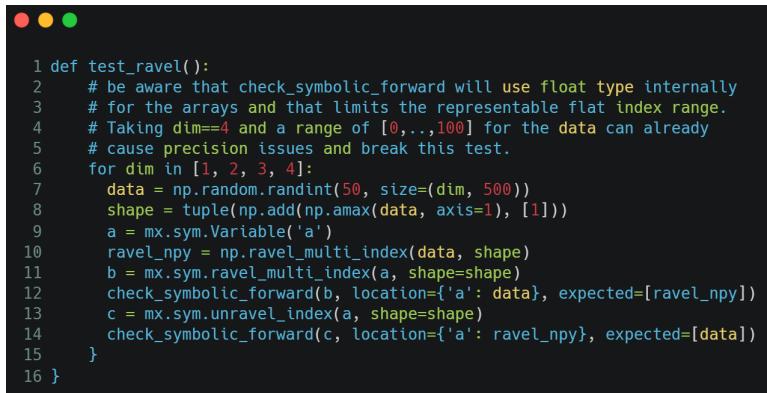
```
 1 def parse_args():
 2     """Parse arguments."""
 3     parser = argparse.ArgumentParser(
 4         formatter_class=argparse.ArgumentDefaultsHelpFormatter,
 5         description='Diagnose script for checking the current system.')
 6     choices = ['python', 'pip', 'mxnet', 'os', 'hardware', 'network']
 7     for choice in choices:
 8         parser.add_argument('--' + choice, default=1, type=int,
 9                             help='Diagnose {}.'.format(choice))
10     parser.add_argument('--region', default='', type=str,
11                         help="Additional sites in which region(s) to test. \
12                         Specify 'cn' for example to test mirror sites in China.")
13     parser.add_argument('--timeout', default=10, type=int,
14                         help="Connection test timeout threshold, 0 to disable.")
15     args = parser.parse_args()
16     return args
17
18 URLS = {
19     'MXNet': 'https://github.com/apache/incubator-mxnet',
20     'Gluon Tutorial(en)': 'http://gluon.mxnet.io',
21     'Gluon Tutorial(cn)': 'https://zh.gluon.ai',
22     'FashionMNIST': 'https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/fashion-mnist/train-labels
   idx1-ubyte.gz',
23     'PYPI': 'https://pypi.python.org/pypi/pip',
24     'Conda': 'https://repo.continuum.io/pkgs/free/',
25 }
26 REGIONAL_URLS = {
27     'cn': {
28         'PYPI(douban)': 'https://pypi.douban.com/',
29         'Conda(tsinghua)': 'https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/',
30     }
31 }
```

Fig. 18: Code snippet of the incorrect source file for the tensor bug retrieved by BugLocator (`https://bit.ly/3Dr5LA9`)

```
 1 def test_ravel():
 2     # be aware that check_symbolic_forward will use float type internally
 3     # for the arrays and that limits the representable flat index range.
 4     # Taking dim==4 and a range of [0,..,100] for the data can already
 5     # cause precision issues and break this test.
 6     for dim in [1, 2, 3, 4]:
 7         data = np.random.randint(50, size=(dim, 500))
 8         shape = tuple(np.add(np.amax(data, axis=1), [1]))
 9         a = mx.sym.Variable('a')
10         ravel_npy = np.ravel_multi_index(data, shape)
11         b = mx.sym.ravel_multi_index(a, shape=shape)
12         check_symbolic_forward(b, location={'a': data}, expected=[ravel_npy])
13         c = mx.sym.unravel_index(a, shape=shape)
14         check_symbolic_forward(c, location={'a': ravel_npy}, expected=[data])
15     }
16 }
```

Fig. 19: Code snippet of the ground truth for the API bug [43] (`https://bit.ly/3JT1JE8`)

```python
def build(self, hp):
    """Build the HyperModel into a Keras Model."""
    tf.keras.backend.clear_session()
    self._register_hps(hp)
    self.compile()
    real_nodes = {}
    for input_node in self.inputs:
        node_id = self._node_to_id[input_node]
        real_nodes[node_id] = input_node.build()
    for block in self.blocks:
        temp_inputs = [real_nodes[self._node_to_id[input_node]]
                       for input_node in block.inputs]
        outputs = block.build(hp, inputs=temp_inputs)
        outputs = nest.flatten(outputs)
        for output_node, real_output_node in zip(block.outputs, outputs):
            real_nodes[self._node_to_id[output_node]] = real_output_node
    model = tf.keras.Model(
        [real_nodes[self._node_to_id[input_node]] for input_node in
         self.inputs],
        [real_nodes[self._node_to_id[output_node]] for output_node in
         self.outputs])

    return self._compile_keras_model(hp, model)
```

Fig. 20: Code snippet of the ground truth for the GPU bug [44] (`https://bit.ly/44llZXc`)