electronics

MDPI

Article

Source File Tracking Localization: A Fault Localization Method for Deep Learning Frameworks

Zhenshu Ma, Bo Yang * and Yuhang Zhang

School of Information Science and Technology, Beijing Forestry University, Beijing 100083, China; mzs0822@bjfu.edu.cn (Z.M.); zhangyuhang306@bjfu.edu.cn (Y.Z.)
* Correspondence: yangbo@bjfu.edu.cn

Abstract: Deep learning has been widely used in computer vision, natural language processing, speech recognition, and other fields. If there are errors in deep learning frameworks, such as missing module errors and GPU/CPU result discrepancy errors, it will cause many application problems. We propose a source-based fault location method, SFTL (Source File Tracking Localization), to improve the fault location efficiency of these two types of errors in deep learning frameworks. We screened 3410 crash reports on GitHub and conducted fault location experiments based on those reports. The experimental results show that the SFTL method has a high accuracy, which can help deep learning framework developers quickly locate faults and improve the stability and reliability of models.

Keywords: fault report; fault localization; source file retrieval; sequence matching; fuzzy matching

Citation: Ma, Z.; Yang, B.; Zhang, Y. Source File Tracking Localization: A Fault Localization Method for Deep Learning Frameworks. *Electronics* **2023**, *12*, 4579. https://doi.org/10.3390/electronics12224579

Academic Editor: Antoni Morell

Received: 29 August 2023
Revised: 28 October 2023
Accepted: 7 November 2023
Published: 9 November 2023

1. Introduction

Deep learning is widely used in many fields, including but not limited to sentiment analysis, visual recognition, natural language processing, and so on [1–3]. The implementation of these functions basically follows the following three steps. Firstly, as many target images and other images as possible are collected as training data through field photography, web crawlers, or with the help of publicly available datasets. Secondly, the data are cleaned and segmented to create training and test sets. Finally, a suitable open-source deep learning framework is selected for building and training the neural network and optimizing the model's participation to achieve good results. A few open-source deep learning frameworks include TensorFlow [4] developed by Google, PyTorch [5] developed by Microsoft, Caffe [6] developed by BVLC, and Jittor [7] developed by Tsinghua University. After choosing the right deep learning framework, the process of building and training neural network models is often much easier.

However, as more and more cases, for example, autonomous driving errors and face recognition system errors, have occurred in recent years, research has shown that there are large and small vulnerabilities in mainstream deep learning frameworks, which may interfere with the computational process of the model and change its prediction results, thus affecting the judgment of the system and ultimately leading to system failures.

As can be seen from Figure 1, the deep learning model relies on the deep learning framework in the actual application process. The deep learning framework mainly consists of the computing part of the deep learning library, the underlying hardware part, and the interface part between the layers. The creation, integration, and invocation relationships between the framework interfaces are very complex, and with version changes, the interfaces are added and repealed more frequently, and many potential errors in the framework are due to the incorrect invocation of the interfaces.

The statistics of the number of reviews on GitHub about the major deep learning frameworks as of 18 February 2023 are shown in Table 1. Currently, PyTorch and TensorFlow, which are commonly used in academic research and industry, are the mainstream deep learning frameworks. However, the issues about these two frameworks account for a large

proportion of the table. Therefore, testing these two frameworks is very necessary. The issues on GitHub record the problems encountered by users when using the frameworks' interfaces or underlying arithmetics, which reflect the errors within the framework to some extent. Starting with these issues is a very feasible idea.

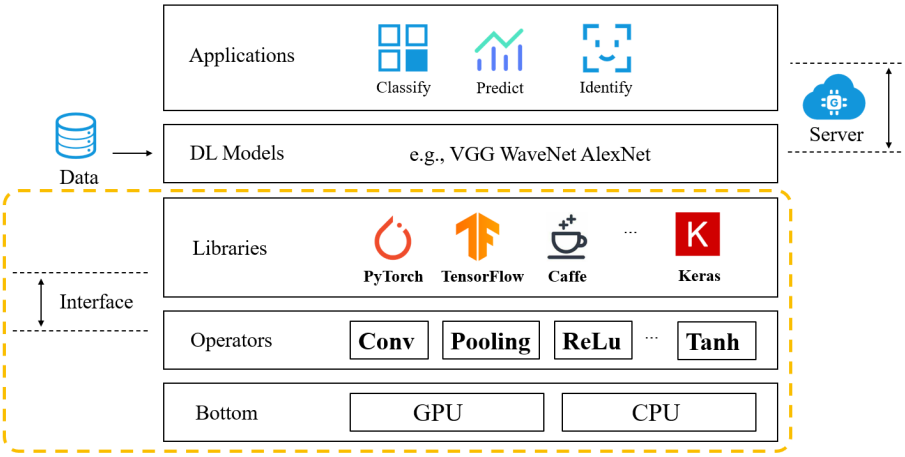

Figure 1. Schematic diagram of deep learning framework.

Table 1. Amount of issues in mainstream deep learning frameworks [1].

Framework	Open	Closed	Total
TensorFlow	2039	34,651	36,690
PyTorch	10,327	21,191	31,518
MXNet	1789	7758	9547
Caffe2	580	731	1311
Jittor	163	74	237
PaddlePaddle	99	37	136

[1] As of 18 February 2023.

In recent years, automated software fault localization methods have been proposed in the academic community. For example, Guo et al. [8] constructed Audee, a new method for DL framework testing and error localization. Audee was tested on four DL frameworks and proved its effectiveness in detecting inconsistencies, crashes, and NaN errors. Du et al. [9] conducted a comprehensive empirical study of three widely used deep learning frameworks (i.e., TensorFlow, MXNET, and PaddlePaddle). The study was based on 3555 bug reports from the GitHub repositories of these frameworks. Defects were classified according to fault-triggering conditions. The frequency distribution and evolutionary characteristics of different bug types were analyzed. In addition, correlations between bug types and fix times were investigated, and regression errors in deep learning frameworks were analyzed. At last, they revealed 12 important findings and provided 10 insights for developers and users based on experimental results.

Currently, there are many fault localization methods for program crashes and program logic errors [10–14]. However, there is less research on module loss errors and framework interface layer errors in runtime errors. The innovation of this paper is that we have summarized two corresponding automated fault location methods for missing module errors and GPU/CPU result discrepancy errors in the framework by analyzing a large number of fault reports and performing the manual location of multiple faults and verifying the feasibility and high accuracy of the methods through experiments.

The contributions of this paper are three-fold, as summarized in the following:

- We collect, analyze, and reproduce fault reports on deep learning frameworks on GitHub. These reports show that a lot of errors in TensorFlow and PyTorch have not been fixed so far.

- We propose a lightweight fault localization approach, SFTL (Source File Tracing Localization), for missing module errors and GPU/CPU result discrepancy errors in Pytorch and TensorFlow deep learning frameworks.
- We conducted experiments on the proposed approach. The experiment results show that SFTL has a good performance on missing module error and GPU/CPU result discrepancy error localization.

The remainder of this paper is organized as follows. Section 2 briefly introduces the background of the fault localization of deep learning frameworks. Section 3 presents our proposed SFTL. Section 4 reports the empirical study for the evaluation of SFTL and describes the experiment's results. Section 5 briefly introduces the related works of the fault localization of deep learning frameworks. Section 6 summarizes this paper.

2. Background

2.1. Fault Report Collation and Analysis

Deep learning frameworks' program errors can be divided into two categories, which are runtime errors and logic errors. There are two typical types of runtime errors, missing module errors (MMEs) and GPU/CPU result discrepancy errors (GCRDEs). An MME is when a framework is calling a model or algorithm and the necessary modules or libraries are missing, resulting in the inability to run the program or implement specific functions. A GCRDE refers to the fact that when the framework calls the CPU and GPU for computation it sometimes produces different results. In general, this is because there is a difference in the implementation of the code on the CPU and GPU, resulting in differences in the results of the operation. For example, different versions of libraries or frameworks are used, different compilers or compilation options are used, or different levels of optimization are taken. These differences can lead to different ways of loading models, resulting in different results.

We analyzed 3410 reviews of TensorFlow and PyTorch on GitHub and selected those with a canonical fault report format, namely, run-alone code with error problems and an indication of the environment in which they were running. We collected these fault reports and summarized the common types of errors, as shown in Figure 2. We can find missing module errors and GPU/CPU result discrepancy errors account for a relatively high proportion: missing module errors account for 64.8% of runtime errors and GPU/CPU result discrepancy errors account for 71.7% of logic errors.

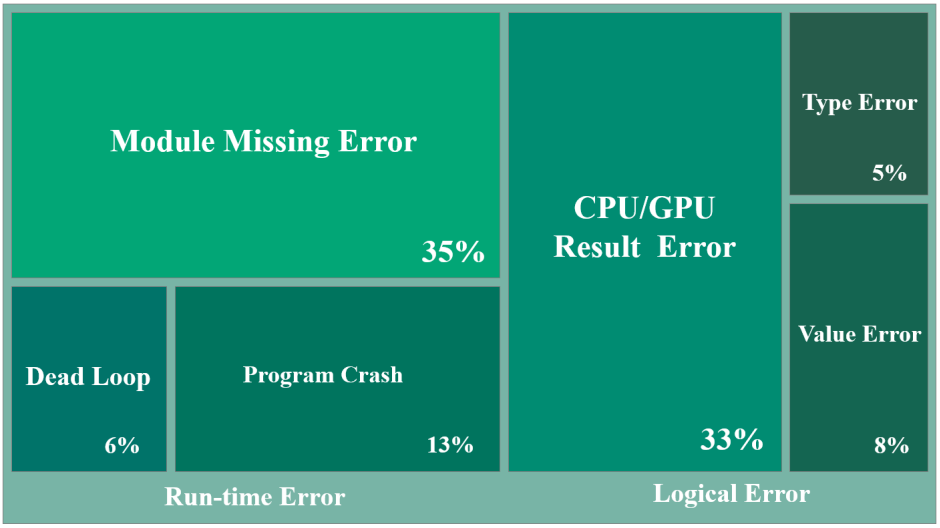

Figure 2. Common types of errors in deep learning frameworks.

2.2. Module Missing Error

When using TensorFlow or PyTorch for numerical operations and model building, users will inevitably encounter some system errors that are difficult to solve through re-

trieval and thought. Some users will document the process of system bug reporting and submit it to the TensorFlow (PyTorch) discussion forum on GitHub. To learn as much as possible about the specifics of what happens when errors occur in deep learning frameworks, we have analyzed hundreds of reports in detail and performed fault replication to summarize two types of errors that receive little attention.

The first type of error is a module missing error, which is reported as AttributeError: "XXX" object has no attribute "XXX" or ModuleNot-FoundError: No module named "XXX". This type of error is usually caused by a function in the program referring to a module that does not exist in that version, or by a function in the program referring to a function in the module that does not exist.

Take TensorFlow #issue 53149: https://github.com/tensorflow/tensorflow/issues/53149 (accessed on 21 November 2021), for example. This bug report documents an error that occurs when a user performs an irregular tensor conversion to numpy format; the test code provided in the bug report is shown in Figure 3.

```
import tensorflow as tf

def gen():
        ragged_tensor = tf.ragged.constant([[1, 2], [3]])
        yield 42, ragged_tensor

dataset = tf.data.Dataset.from_generator(
        gen, output_signature=(
        tf.TensorSpec(shape=(), dtype=tf.int32),
        tf.RaggedTensorSpec(shape=(2, None),
         dtype=tf.int32)))

iterator = dataset.as_numpy_iterator()
print(iterator.next())
```

Figure 3. Faulty codes in TensorFlow issue #53149.

Running the test code under the runtime environment provided in the bug report (Ubuntu18/Python 3.7/TensorFlow 2.6.0) yields the following error message in Figure 4.

```
File "/Users/xx/Library/Caches/pypoetry/virtualenvs/sandbox-
hasb3I3q-py3.6/lib/python3.6/site-packages/tensorflow/python/
data/ops/dataset_ops.py", line 4685, in to_numpy
    numpy = x._numpy()
AttributeError: 'RaggedTensor' object has no attribute '_numpy'
```

Figure 4. Traceback in TensorFlow issue #53149.

This bug reports a reference error in the framework source code: the RaggedTensor class is missing the _numpy attribute. The cause of this error was traced to the traceback error message:

It was found that when the Dataset class called the as_numpy_iterator() method, as_numpy_iterator() referenced the to_numpy() function, and in the process a method miss occurred; the error occurred with the statement numpy = x._numpy(). After manually checking the error path, it was found that the file defining the RaggedTensor class only has a definition for the numpy() method, but not for the _numpy() method. x does not have the method _numpy(), and x's call to the non-existent method _numpy() has caused the error.

2.3. GPU/CPU Result Discrepancy Error

In addition to errors caused by missing modules, we have found cases where the same operator behaves differently on the CPU and GPU, which is named GPU/CPU result discrepancy errors.

Take, for example, PyTorch #issue #87657: https://github.com/pytorch/pytorch/issues/87657 (accessed on 25 October 2022), which describes an inconsistency between THE CPU and CUDA on the "Histc" function. The function runs on the CPU and the GPU and returns slightly different results. The resulting study of the source code for the function's implementation on the CPU and GPU reveals that the main reason for the slightly different returned output is that the results are calculated in a slightly different way.

The code path for the "Histc" function on the CPU is pytorch/ATen/src/ATen/native/cpu/HistogramKernel.cpp.

The kernel codes are as follows:

pos = static_cast<int64 t>((elt − leftmost edge[dim]) ∗ (rightmost edge[dim] − leftmost edge[dim]) ∗ (num bin_edges[dim] − 1)).

The overall formula for the core code is

$$(i - a)/(b - a) * N \tag{1}$$

The code path for the Histc function on the GPU is pytorch/ATen/src/ATen/native/cuda/SummaryOps.cu. The kernel code is as follows:

IndexType bin = (int)(((bVal − minvalue)) ∗ nbins /maxvalue − minvalue)).

The overall formula for the core code is

$$(i - a) * N/(b - a) \tag{2}$$

Comparing the two formulas shows that in the key code part of the implementation of this operator the division operation is performed first and then the multiplication operation on the CPU. While on the GPU, the multiplication operation is performed first and then the division operation. In some cases, especially when performing floating-point operations, the order of the arithmetic operations is extremely important and largely determines the outcome of the operation. After some research, it was finally confirmed that the inconsistent output of the Histc arithmetic from two different processors was a bug.

3. Methodology

3.1. The Framework of SFTL

The framework of the approach is given in Figure 5. We use the bug reports on GitHub about the deep learning frameworks as the source data. Then, we filter the bug reports reflecting specific error types from the source data, reproduce the faults, and extract the trace file according to the test code and version information provided in the bug reports. If the reported error is a module missing error, then the suspected fault file will be investigated using the module missing error process. If the reported error is a GPU/CPU result discrepancy error, then the source file will be traced using the GPU/CPU result discrepancy error process.

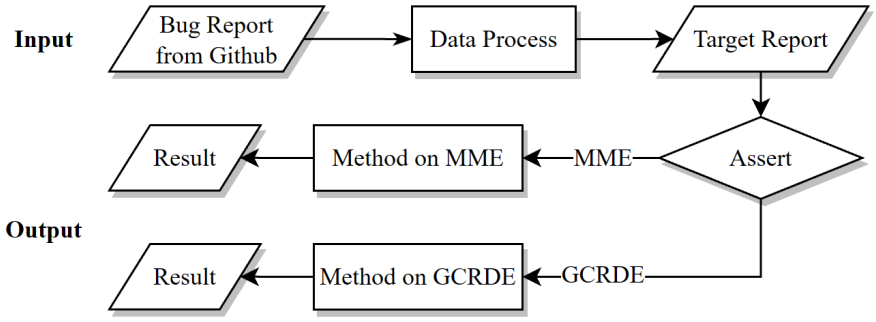

Figure 5. The framework of SFTL.

3.2. Data Pre-Processing

The experiments select 1776 PyTorch issues and 1634 TensorFlow issues, which respond to different issues and needed to be filtered for those issues on module missing class errors and GPU/CPU result discrepancy errors.

Our proposed approach is to extract the frequency of keywords in the title and body of the issue to determine whether the issue belongs to the one needed for the experiment. We use the TF-IDF (Term Frequency–Inverse Document Frequency) algorithm to preprocess data. The principle of the TF-IDF algorithm can be summarized as follows: if a keyword appears more often in one article and less often in others, it is more representative of the topic of that paper:

$$TF_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{3}$$

where $n_{i,j}$ indicates the number of occurrences of keyword t_i in issue d_j and $TF_{(i,j)}$ is the frequency of occurrences of keyword t_i in issue d_j. When filtering for missing module errors, the keywords selected are "Attribute", "Not Found", "no", etc. When filtering for GPU/CPU result discrepancy errors, the keywords selected are "CPU", "GPU", "Result", "Output", etc., which express the topic of issue. The weight of each word is also designed according to its ability to extract the issue topic.

$$IDF_i = log\frac{|D|}{1 + |j : t_i \in d_j|} \tag{4}$$

$$TF - IDF = TF \cdot IDF \tag{5}$$

where $|D|$ denotes the number of all issues and $|j : t_i \in d_j|$ denotes the number of issues containing keyword t_i. After calculating the TF-IDF of each issue according to the above formula, the issues whose sum of keyword-weighted scores exceeds the threshold value are the target issues to be filtered. However, since it was not known how many relevant issues were missed, 1176 PyTorch issues and 1634 TensorFlow issues were read and annotated one by one to ensure that only a few relevant issues were missed. In addition, we found that not all reports can be used as source data for automated detection. To this end, reports need to meet the following conditions:

- The report clearly describes the process by which the problem occurred during the user's use of the framework;
- The report provides code that can be run independently to reproduce the original problem;
- The report provides a specific version of the framework and the dependent environment;
- The problem documented in the report was caused by a potential vulnerability in the framework itself and not by a user error.

If any of the above conditions are not met, the sample provided in the issue will not run in the corresponding experimental environment and no valid error feedback from the framework will be available.

3.3. Method for Missing Module Errors

We propose a missing file tracking method (Figure 6) that allows the tester to simply enter the number of the bug report and obtain an automatic result for the error in the report.

The missing file tracking process for missing module type bugs is divided into four main steps:

1. Crawl the corresponding bug report by number or URL.

The first step is to crawl the corresponding issue directory of TensorFlow (PyTorch) on GitHub and obtain the issue number through regular expression processing. After that, we use the Webdriver module [15] in Selenium to crawl the web pages using the number as an index for bug reports. Crawling a single fault report web file took 3.5 s.

The crawling of GitHub's issue pages is unstable due to the large number of pages and it is prone to connection timeouts. Therefore, when crawling a single page fails, the crawler is made to throw an exception and disconnect from the GitHub server within 2 s to avoid IP blocking. After 2 s, the connection continued to be attempted until the page was successfully crawled.

Through the crawler, 1776 issues of PyTorch and 1634 issues of TensorFlow were crawled.

Input: Number ***No*** of the Issue reflecting the missing module error
Output: The location in the framework of the file that caused the missing module error ***path***

Process

1 Crawl the source code of the Issue ***No***.
2 Extract the specific environment in which the fault occurred in Issue based on the source code, including the framework version, system type and dependency package version.
3 Extract the fault code provided by the source code and process it so that it can throw an exception message when run;
4 Run the fault code to get the Traceback file, and match the header file of the missing class and the suspect reference file in the Traceback to locate the ***path*** of the fault file in the source file.

Figure 6. Missing file tracking location process.

2. Framework versions, dependency package versions, and system types are automatically identified based on bug reports.

In order to select a local runtime environment that matches the code runtime environment in the issue, the framework version, dependency package version, and system type where the error occurred were extracted from the bug report and entered into a similar local environment to test the code. If a valid framework version, dependency package version, or system type cannot be extracted, the issue is determined to be an irregular bug report.

3. Automatically identify and run test code based on bug reports.

We extract the test code from the crawled page that can be run independently. Then, we run it and record the traceback information returned by the program. If no valid test code can be extracted, the issue is judged to be an unstandardized bug report and cannot be tested.

4. Combine the import module and traceback to automatically locate the faulty class or faulty interface (take TF#53149 as an example).

First, as can be seen in Figure 7, we extracted all the files involved in the traceback path, which were dataset_ops.py and nest.py in this case, which constitute a collection of suspect reference files, and extracted the name of the missing class, which is "RaggedTensor" in this case, and the name of the missing attribute, which is "_numpy" in this case.

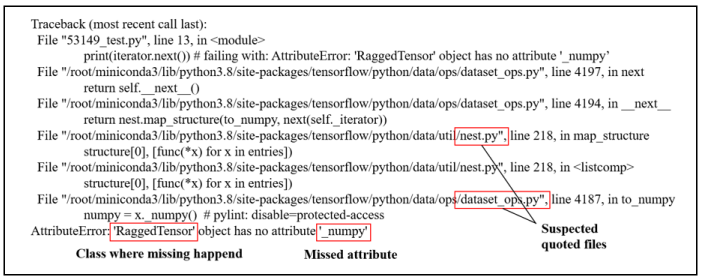

Figure 7. Key arguments in traceback.

As can be seen in Figure 8, files in the suspect reference file collection are extracted one by one, the similarity between the missing module name and the referenced module name is compared, and the similarity is ranked. Then, the module name with the highest similarity is selected as the suspect fault file. Here, the ragged_tensor file introduced from ops. ragged has the highest similarity to the missing class.

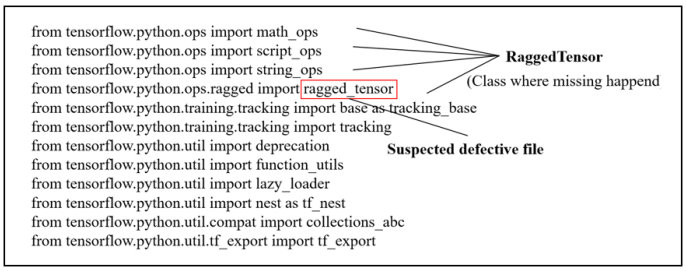

Figure 8. Matching of suspected files with missing classes.

To calculate the similarity between the occurrence of the missing error class and the suspect file, we used the Needleman–Wunsch algorithm commonly used for sequence comparison [16].

From Figure 8, the class name of the missing class is RaggedTensor, while the suspect files include math_ops, scripts_ops, string_ops, ragged_tensor, tf_export, etc. The Needleman–Wunsch algorithm was used to compare the class names of the missing classes with the suspect files one by one. The filenames are matched one by one, and the matching process is shown in Figure 9.

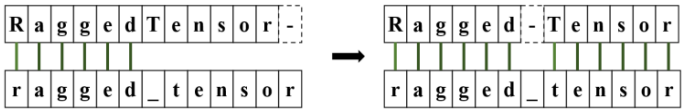

Figure 9. Matching-based Needleman–Wunsch algorithm.

The maximum match score is calculated during the matching process. The formula for calculating the maximum match score is as follows:

$$F(i,j) = max \begin{cases} F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) + p \\ F(i,j-1) + p \end{cases} \quad (6)$$

We calculate the score $F(m, n)$ for the missing class name M of length m and the suspect file N of length n. Then, we start from $F(0,0)$, match the characters one by one, add s for a successful match, and penalize p for matching the empty space.

The top 3 suspect files are selected for the matching score, i.e., the top 3, and all the methods (beginning with def) in the code of the file definition class are extracted from these files to see if the missing attributes are in these methods. If the missing attribute is not in these methods, the suspect fault file is identified as having a missing module error and the system automatically generates a fault report.

3.4. Method for GPU/CPU Result Discrepancy Error

The experiments were preceded by research and the collation of a large number of bug reports for the TensorFlow and PyTorch frameworks. Then, a summary of the various types of errors was reported, as well as corresponding localization and resolution methods for the different error types.

We can find that within the same framework a single deep learning operator may have multiple implementations that are independent of each other. For example, in TensorFlow there are about 10 separate code implementations related to conv2d using different processors (e.g., CPU, GPU) and different data types (e.g., float, double, uint16, etc.).

The focus here is on the overall PyTorch framework, and the overall PyTorch code architecture is shown in Figure 10.

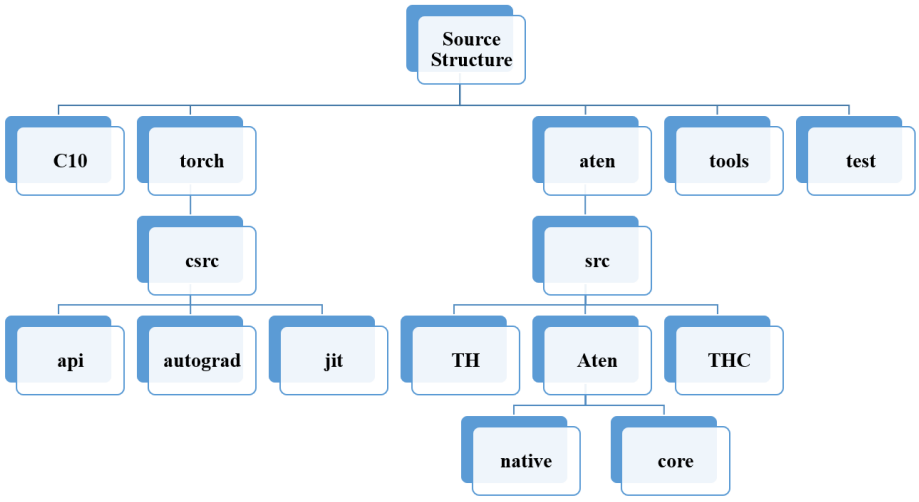

Figure 10. Structure of PyTorch.

"c10/" is an abbreviation for "Caffe" and "A Tensor Library", PyTorch's core library files, which contain implementations of basic torch functions suitable for use in settings where the binary size is important, as well as support for use on servers and mobile devices.

The code files in "torch/" contain Python code modules for PyTorch, following the front-end module structure; you can view the function interface information in the torch/ folder, including the relevant parameters and comments, by Ctrl+right-clicking in the editor.

"torch/csrc/" encapsulates some of Pytorch's core libraries and contains the C++ files that make up the PyTorch library, a mix of Python bindings and C++ refactorings, many of which are only generated when the code is compiled.

"aten/src/ATen/", "ATen", stands for "A Tensor Library" and is the location where many of the PyTorch functions are implemented, including C++ or C implementations of the operators in torch.

"src/Aten/core" implements the core functionality of ATen. However, as the official PyTorch documentation states, the code is gradually being migrated to "c10/";

"src/Aten/native" is PyTorch's arithmetic library, mainly a C++-style implementation and a modern implementation of operators, where most of the surface implementation files

for CPU arithmetic are located, while some operators that require special compilation are placed in its subdirectory src/Aten/native/cpu.

"src/TH" is also PyTorch's arithmetic library, which is mainly a C-style implementation and is relatively old. However, in PyTorch 1.7, PyTorch 1.8, PyTorch 1.9, PyTorch 1.10, etc., "native/" covers most of the tensor operations, confirming that the code in "/TH" is gradually migrating to "native/" as the versions are updated.

"aten/src/ATen/native/cpu/" is the implementation file for the CPU operator.

"src/Aten/native/cuda" is the implementation part of the CUDA operator.

In summary, if you want to see the implementation of a particular torch function, the first place to look is the kernel code in the "aten/src/ATen/native/" section.

To see what the function does in "aten/src/ATen/native", first look in the "ate/src/ATen /native/native_functions.yaml" registry file to find the overall framework of the function and determine if it is implemented in "native/".

Native_functions.yaml is a configuration file for the forward operators, which contains a general description of each operator, where each operator has a number of fields, including the func, variants, dispatch, etc.

The func field indicates the name of the operator and the type of input and output.

The variants field indicates how the advanced method should be called. The method indicates that a tensor can be defined and called with an absolute (), and the function indicates that the absolute method of the torch can be automatically generated and called with torch.absolute().

The dispatch field indicates the type of device being dispatched and can be called with different kernels for different backends.

Based on the above introduction, an automatic locating method for vulnerability files is proposed, as is shown in Figure 11.

When locating where the function is implemented on the CPU, first go to the pytorch/ATen/src/ATen/native directory and find the native_functions.yaml file. In it, look for information about the function and, if it exists, read the dispatch field of the function's architecture code to find the field where the device type distributed is the CPU. Again, since the underlying implementation file for the arithmetic is in the aten/src/ATen/native/cpu/ directory and usually exists as xxx_kernel.cpp, the resulting dispatch field is stitched with the kernel to obtain a prediction filename that implements the function. After that, go into the aten/src/ATen/native/cpu/ directory and read all the filenames in that directory and put them into a list. Finally, call Python's string fuzzy comparison function difflib.get_close_matches to fuzzy match the predicted filenames with all the filenames in the list.

Difflib.get_close_matches internally uses the SequenceMatcher module to compute the longest common subsequence between two strings. Afterward, for each candidate string, the function calculates a value to indicate its similarity to the target string. The calculation process takes into account four factors simultaneously:

- The number of identical characters: the higher the number of identical characters, the higher the similarity.
- The relative distance between the positions of the same characters: the closer the positions of the same characters, the higher the similarity.
- The number of missing characters: the fewer the missing characters, the higher the similarity.
- Number of new characters: the fewer the new characters, the higher the similarity.

Input: the name of the function to be located, fun, and the path to the PyTorch file, ***path***
Output: the name of the source file where the function is located ***result***

Algorithm:
1 Initialize the prediction file with the name prename.
2 Go to ***path*** + /aten/src/ATen/native.
  2.1 Read the native_functions.yaml file line by line.
  2.2 Find the information about the fun function and find the dispatch field key of the device type distributed as CPU (GPU).
  2.3 Assign the value of key + kernel to prename to get the predicted function name.
3 Initialize the file list file.
4 Go to ***path*** + aten/src/ATen/native/cpu(cuda)/path.
  4.1 Call the os.listdir() function to read all the filenames under that path and add them to the file list file.
5 Call the difflib.get_close_matches function to match file with prename.
6 Locate the file name with the greatest match ***result***

Figure 11. Process of locate operators in GPU/CPU.

User-defined weighting factors can be used to adjust the effect of these factors on similarity. The default weighting factors are [0.6, 0.3, 0.1, 0.0], with the number of matching blocks having the greatest influence followed by the character length, and the number of transpositions having a factor of 0.

The string similarity is calculated as follows:

$$\frac{len(match\_blocks) * (len(target\_string) + len(other))}{2 * len(target\_string) * len(other)} \quad (7)$$

Match_blocks indicates the longest common subsequence of the target string and the candidate string. *len*(*target_string*) is the length of the target string, and *len*(*other*) is the length of the candidate string.

The filename with the highest degree of match is the most likely filename to be implemented by this function. When calling this function, be careful to set the cutoff, the match size, to a small value if possible; if the value is too large, the result may be null.

It should be noted that we illustrate our algorithm in terms of PyTorch type error localization, similar to the TensorFlow type error algorithm.

4. Experiments and Results Analysis

4.1. Research Questions

To evaluate the effectiveness of STFL and compare it on some particular tasks, our experiments were particularly designed to answer the following four research questions:

RQ1: How effective is the target report filtering?

RQ2: What is the accuracy of automated fault location methods for module missing-type errors?

RQ3: What is the accuracy of automated fault location methods for GPU/CPU result discrepancy errors?

4.2. Experimental Subjects

We collated 1776 recent PyTorch issues and 1634 TensorFlow issues on the GitHub official website, and the issues reflected by these issues varied. The reports were filtered by automated scripts, and we obtained 431 issues that reported missing module errors in

TensorFlow and 81 issues that reported GPU/CPU result discrepancy errors in TensorFlow. In addition, 284 issues that reported missing module errors in PyTorch and a total of 178 issues reflecting GPU/CPU result discrepancy errors in PyTorch were used as the subjects of this study.

To ensure the reproducibility and accuracy of the experimental results, the identified issues must include independent test code that can be run seamlessly in the corresponding experimental environment. Additionally, we provide detailed information on the specific versions of both the framework and dependent environments, such as Python, which is crucial.

We followed the aforementioned requirements and conducted a thorough screening of the entire population of issues under consideration. This screening process led to the identification of 821 reports that met the necessary criteria and were deemed appropriate experimental subjects. By utilizing these valid reports as our experimental subjects, we were able to ensure that consistent and reliable results could be obtained in our study.

For RQ2 and RQ3, we compare STFL with VSM and Bert models.

4.3. Evaluation Metrics

A sorted list of source files is generated for each fault report and its performance is evaluated using the following metrics:

- Accuracy

This metric is the accuracy of the test method for the subject and is calculated as the number of correctly identified errors/total number of samples.

- Precision

This is the accuracy rate of the test method against the subject, calculated as the number of errors found by the method/and the total number of samples identified as errors by the method.

- Recall

This indicator is the recall rate of the test method for the experimental object, calculated as the number of errors found by the method/and the number of true errors in the sample.

- F1-Score

The F1-Score takes into account both the accuracy and recall of the classification model.

4.4. Experimental Environment

The operating system used for the experiments on the Linux side was Ubuntu 18.04, with an Intel(R) Xeon(R) Platinum 8255C CPU and RTX 2080 Ti GPU. The operating system used on the Windows side was Windows 11, with an i7-11700 CPU and RTX 3070 GPU. A virtual environment with different deep learning framework versions was created via Anaconda3 before the experiments, including TensorFlow (2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.4.0, 2.5.0, 2.6.0, 2.7.0, 2.8.0, 2.9.0, 2.10.0, 2.11.0) and PyTorch (1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 1.9.0, 1.10.0, 1.11.0). By setting up different versions of the virtual environment, we can ensure that the test code in the bug report runs under the version described in the bug report, ensuring that the test code feedback error is true and valid.

We also installed the usual dependency packages in each environment to ensure that the test code runs without interruptions, thus obtaining the expected traceback.

4.5. Experimental Results

4.5.1. Answer to RQ1 and Results Analysis

The experiment selected 1776 PyTorch issues and 1634 TensorFlow issues, which responded to a variety of issues and needed to be screened for errors in the missing module category and GPU/CPU result discrepancy category.

Each of the 1176 PyTorch issues and 1634 TensorFlow issues were read and annotated to ensure that only a few relevant issues were missed.

Table 2 shows the results of the issue screening in the experiments. The experiments extracted 284 issues from 1776 PyTorch issues on mode-missing class errors and 178 issues on GPU/CPU result discrepancy errors. In addition, the experiments extracted 431 issues from 1634 TensorFlow issues on mode-missing class errors and 81 issues on GPU/CPU result discrepancy errors.

Table 2. Issue screening experiment results.

Dataset	Error_Type	issue_Num	Related_issue	Accurancy	Precision	Recall	F1-Score
PyTorch	MME	1776	284	0.999	0.996	1.000	0.998
TensorFlow	MME	1634	431	0.998	0.993	1.000	0.997
PyTorch	GCRDE	1776	178	0.999	0.994	1.000	0.997
TensorFlow	GCRDE	1634	81	0.998	0.976	0.988	0.982

We analyzed the accuracy of both frameworks' failure report screening, which was close to 100%. It turns out that this is because we use word frequency statistics, and the topic of the fault report can often be directly displayed through keywords.

4.5.2. Answer to RQ2 and Results Analysis

After filtering out valid issues from the relevant issues, a sample of 243 issues related to module-like errors (MMEs) in PyTorch and 375 issues related to missing module-like errors in TensorFlow were used as experimental samples for automated localization. Whether the official response to the issue was valid or whether the error was acknowledged by other users under the issue was taken as whether the issue truly reflected the missing module error in the framework.

Table 3 shows the performance of the automated fault localization approach for missing module errors in the experiments. For the 243 valid samples in the PyTorch issues, the method located 183 valid errors with an accuracy of 0.84, precision of 0.93, recall of 0.88, and F1-Score of 0.91. For the 375 valid samples in the TensorFlow issues, the method was able to locate 298 valid errors with an accuracy of 0.86, precision of 0.93, recall of 0.90, and F1-Score of 0.92.

Table 3. MME experiment results.

Dataset	Method	Accurancy	Precision	Recall	F1-Score
PyTorch	VSM	0.59	0.70	0.63	0.66
PyTorch	BERT	0.76	0.82	0.86	0.84
PyTorch	SFTL	0.84	0.93	0.88	0.91
TensorFlow	VSM	0.63	0.75	0.78	0.77
TensorFlow	BERT	0.78	0.82	0.88	0.85
TensorFlow	SFTL	0.86	0.93	0.90	0.92

Our method achieves accuracy values of 0.84 and 0.86 on the PyTorch and TensorFlow frameworks, respectively, which is better than the VSM method and BERT method. But, actually, it is not high enough. In the actual operation process, the accuracy of the module missing error is very related to the call path of the module. But the call path consists of several calls, and at each call we use the Needleman–Wunsch algorithm to match the reference library to the missing module, so if the path is too long, our path-seeking effect will deteriorate, thus deviating from the ideal situation. Currently, in the vast majority of cases, we can find the completed call path.

4.5.3. Answer to RQ3 and Results Analysis

Similarly, 178 GPU/CPU result discrepancy error (GCRDE)-related issues from PyTorch and 81 GPU/CPU result discrepancy error-related issues from TensorFlow were used as experimental samples for automated localization. Whether the official response to the issue was valid or whether the error was acknowledged by other users under the issue was taken as a true reflection of the GPU/CPU result discrepancy problem in the framework.

Table 4 shows how the automated fault location approach for GPU/CPU result discrepancy errors (GCRDEs) was performed in the experiments. For the 132 valid samples in the PyTorch issues, the method located 83 valid errors with a locality accuracy of 0.74, precision of 0.86, recall of 0.80, and F1-Score of 0.83. For the 71 valid samples in the TensorFlow issues, the method located 49 valid errors, with localization accuracy.

The method achieved a 0.72 accuracy, 0.89 precision, 0.78 recall, and 0.83 F1-Score for the 71 valid samples in the TensorFlow issues.

Table 4. GCRDE Experiment Result.

Dataset	Method	Accurancy	Precision	Recall	F1-Score
PyTorch	VSM	0.59	0.52	0.45	0.48
PyTorch	BERT	0.68	0.80	0.82	0.81
PyTorch	SFTL	0.74	0.86	0.80	0.83
TensorFlow	VSM	0.57	0.64	0.60	0.62
TensorFlow	BERT	0.70	0.80	0.82	0.81
TensorFlow	SFTL	0.72	0.89	0.78	0.83

Our method has accuracy values of 0.74 and 0.72 for PyTorch and TensorFlow, which is better than the VSM and BERT methods. But our accuracy is still not good enough, so we conduct a detailed analysis of this case. We found that this is because methods for CPU/GPU result discrepancies can only locate possible source files in the module tree in the framework. Because some methods are implemented through the CPU and GPU, they do not use the Python language. Therefore, this method cannot be matched inside the source file, like the method for module missing errors, and the complete call path can be found, so it cannot achieve a more ideal effect.

4.6. Discussion

4.6.1. Implications for Interface Layer Fault Localization

The interface layer is the part of the deep learning framework that directly interacts with the user and is responsible for receiving user input data and parameters, as well as returning model outputs. Therefore, faults in the interface layer may directly affect the user's use of the model and understanding of the results. Locating the interface layer error is important to ensure the stability and reliability of the model.

4.6.2. Significance of Automated Fault Detection

Automated fault detection in deep learning frameworks is important for improving the efficiency of fault detection, reducing labor costs, improving the accuracy of fault detection, promoting the continuous optimization of the model, and enhancing the robustness of the model.

Our fault detection and automated testing method is used on module missing errors and GPU/CPU result discrepancy errors. Thus, the experiments also address only these two types of errors. However, in fact, our approach can be applied to other error types as well.

There are complex referencing relationships in deep learning frameworks that constitute a complex network of file references. When an error occurs, the core idea of our proposed method is to find an explicit module reference path that this error corresponds to in the framework. Therefore, logical errors and runtime errors caused by file reference

relationship errors at runtime can be detected by the proposed method in the paper. In order to improve the generalizability of our approach, we plan to validate our approach on PaddlePaddle, Caffe2, and MXNet in future work.

Overall, our proposed automated fault detection approach for two types of errors bridges the gap of automated detection for deep learning frameworks to some extent. Both the missing module error and the GPU/CPU result discrepancy error locate the interface layer, and due to the large number of fault reports submitted by users, we propose automated fault localization methods to improve efficiency and reduce the error rate.

4.7. Threats to Validity

There are two main threats to validity in our study. Firstly, there is a threat to internal validity concerning our implementations. Although we followed the method given in the corresponding original study to replicate the related techniques, there may still be some differences in the results. Secondly, the most obvious threat to external validity is that the selected datasets are only available in GitHub, which might affect the accuracy and statistical reliability of the experimental evaluation.

5. Related Work

Testing of deep learning frameworks has been carried out in recent years. Many bugs in deep learning frameworks have been discovered. How these errors are located is a focus of attention, and the industry has paid much attention to the fault localization of deep learning framework errors. For example, Guo et al. [8] constructed Audee, a method for testing DL frameworks and localization errors, which is able to detect three types of errors: logic errors, crashes, and Not-a-Number errors. It also proposes a test method for each error type. They applied Audee to test four DL frameworks and the results showed that Audee was effective in detecting inconsistencies, crashes, and NaN errors. In total, 26 unique unknown errors were found, 7 of which have been identified or fixed by the developers. Du et al. [9] conducted a comprehensive empirical study of fault-triggering conditions in three widely used deep learning frameworks (TensorFlow, MXNET, and PaddlePaddle). The team collected 3555 bug reports from the GitHub repositories of these frameworks. Gu et al. [17] proposed the Muffin model, which can generate different DL models to explore a target library and does not rely on already trained models but uses an automatic model generation algorithm to obtain different test inputs. To perform variance testing, Muffin relies on data tracking analysis during the training phase. The model training phase is divided into three stages, as forward computation, loss computation, and gradient computation. A set of metrics is designed on the data-trace to measure the consistency of the results across the different DL libraries. This allows Muffin to detect more inconsistencies in a comparable testing time, and inconsistencies can be indicative of potential errors. They applied Muffin to test 15 releases of three widely used DL libraries, TensorFlow, CNTK, and Theano. The results show that Muffin detected 39 new errors (including 21 crash errors) in the latest releases of these libraries. Wang et al. [18] proposed a new approach called LEMON (Deep Learning Library Testing via Guided Mutation), which can generate valid DL models to trigger and expose errors in DL libraries. They designed a series of model mutation rules (both full-layer mutation rules and internal-layer mutation rules) to maximize the ability to explore the DL library code space by altering existing models to automatically generate DL models. At the same time, they proposed a heuristic strategy in LEMON to guide the process of model generation towards amplifying the degree of inconsistency of real bugs, improving the differentiation between real bugs and uncertain effects. Jia et al. [19] proposed an idea to inject bugs into a deep learning library and check to what extent existing test cases could detect the bugs they injected. With a mutation tool, they constructed 1545 flawed versions (i.e., mutants). By comparing the test results of clean and flawed versions, their study yielded 11 findings, which they summarized as answers to three research questions. They chose a mutation tool for Python called output to inject errors and published eleven relevant revelations for four questions. Wei et al. [20] proposed

a method for fuzzing DL libraries by mining information from open sources, which obtains code/models from three different sources, code snippets from library documentation, library developer tests, and DL models in the wild. They automatically run all collected code models and use tools to track FreeFuzz, which will use the traced dynamic information to perform fuzz tests for each covered API. Through extensive research on FreeFuzz on PyTorch and TensorFlow, FreeFuzz is able to automatically track valid dynamic information for 1158 popular APIs, 9× more than the LEMON and with a 3.5× lower overhead than LEMON. To date, FreeFuzz has detected 49 errors for PyTorch and TensorFlow.

Deng et al. [21] proposed DeepREL, an approach based on the automated inference of relational API and thus more effective DL library fuzzing. It is a fully automated end-to-end relational API inference and fuzzing technique. The whole idea of the approach is that in DL libraries there may be many APIs sharing similar input parameters and outputs. Based on this idea, the called API can be used to test other relational APIs. Evaluations of PyTorch and TensorFlow have shown that DeepREL can cover 157% more APIs than the state-of-the-art FreeFuzz. To date, DeepREL has detected a total of 162 vulnerabilities, of which 106 have been identified by developers as previously unknown vulnerabilities. Additionally, in addition to the 162 code bugs, the method has found 14 documentation bugs (all confirmed). Zhang et al. [22] constructed the Duo model, which combines fuzzy techniques and difference testing techniques to generate inputs and evaluate the corresponding outputs while implementing the mutation-based fuzzy generation of tensor inputs to evaluate the correctness of multiple operator instance outputs using nine mutation operators derived from genetic algorithms and difference testing. In contrast to existing DL library testing techniques, Duo performs the testing task directly on DL operators rather than tracing the hidden output of the model. Duo was used in experiments to evaluate seven operators in TensorFlow, PyTorch, MNN, and MXNet, and the results show that Duo can expose flaws in DL operators and enable the multidimensional evaluation of DL operators from different DL libraries. Pei et al. [23] constructed DeepXplore, the first white-box framework for systematically testing real-world DL systems. DeepXplore effectively found thousands of incorrect corner behaviors in state-of-the-art DL models with thousands of neurons trained on five popular datasets. For all DL models tested, DeepXplore generated a test input demonstrating incorrect behavior within one second on average, further showing that the test input generated by DeepXplore can also be used to retrain the corresponding DL model to improve the accuracy of the model by up to 3%. Ma et al. [24] proposed DeepGauge, a set of multi-granularity testing criteria for DL systems, aiming to present a multi-faceted depiction of the testing platform. The set of test criteria is evaluated in depth on two well-known datasets, five DL systems, and four state-of-the-art adversarial attack techniques against DL. Adversarial samples were generated by four adversarial data generation algorithms (Fast Gradient Notation Method (FGAM), BIM, JSMA, CW). For the original test data and adversarial samples, the coverage criteria are used as metrics to quantify the ability of the test data to detect errors. The potential role of DeepGauge provides insight into the construction of more general and robust DL systems.

6. Conclusions

To address issues in the use of deep learning frameworks, we searched for bug reports on forums, such as GitHub and StackOverflow, and performed a replication of the relevant issues. After examining a large number of bug reports, the types of errors on TensorFlow (PyTorch) were divided into two categories: running errors and program logic errors. The focus is then on missing program module-type errors and GPU/CPU result discrepancy errors.

We propose SFTL, an automated vulnerability detection method based on fault reporting. For program modules missing class errors, a combination of the import module and traceback is used to automatically locate the faulty class or faulty interface. The class names of the occurring missing classes are compared with the suspect filenames one by one by the Needleman–Wunsch algorithm, and a match score is calculated, from which the

corresponding errors are located. For GPU/CPU result discrepancy errors, the predicted filenames are formed using the dispatch field of the corresponding function in the configuration file native_functions.yaml of the forward operator, and fuzzy matching is performed using Python's difflib.get_close_matches function, from which the suspect files are located.

Finally, experiments were carried out with 1776 recent PyTorch issues and 1634 TensorFlow issues on the official GitHub website, and the experimental results show that the SFTL automatic localization method has a high accuracy rate. It can help deep learning framework developers to quickly locate errors in deep learning frameworks and thus solve problems faster.

The deep learning framework fault location is important for improving the performance and reliability of deep learning models and the reliability of their applications in a preliminary exploratory attempt. In future work, a more comprehensive and in-depth study of error types in deep learning frameworks will be carried out to propose more advanced and reliable fault location methods to cover more deep learning frameworks, such as PaddlePaddle, Caffe2, and MXNet.

Author Contributions: Methodology, Z.M., B.Y. and Y.Z.; Validation, Z.M., B.Y. and Y.Z.; Writing—original draft, Z.M. and Y.Z.; Writing—review & editing, B.Y. All authors have read and agreed to the published version of the manuscript.

Funding: National Key R&D Program of China: 2022YFF1302700.

Data Availability Statement: Data is contained in the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Chai, Y.; Du, L.; Qiu, J.; Yin, L.; Tian, Z. Dynamic Prototype Network based on Sample Adaptation for Few-Shot Malware Detection. *IEEE Trans. Knowl. Data Eng.* **2022**, *35*, 4754–4766. [CrossRef]
2. Qiu, J.; Tian, Z.; Du, C.; Zuo, Q.; Su, S.; Fang, B. A Survey on Access Control in the Age of Internet of Things. *IEEE Internet Things J.* **2020**, *7*, 4682–4696. [CrossRef]
3. Qiu, J.; Du, L.; Zhang, D.; Su, S.; Tian, Z. Nei-TTE: Intelligent Traffic Time Estimation Based on Fine-Grained Time Derivation of Road Segments for Smart City. *IEEE Trans. Ind. Inform.* **2020**, *16*, 2659–2666. [CrossRef]
4. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv* **2016**, arXiv:1603.04467.
5. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv* **2019**, arXiv:1912.01703.
6. Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. In Proceedings of the 22nd ACM International Conference on Multimedia (MM '14), Orlando, FL, USA, 3–7 November 2014; Association for Computing Machinery: New York, NY, USA, 2014; pp. 675–678.
7. Hu, S.-M.; Liang, D.; Yang, G.-Y.; Yang, G.-W.; Zhou, W.-Y. Jittor: A novel deep learning framework with meta-operators and unified graph execution. *Sci. China Inf. Sci.* **2020**, *63*, 222103. [CrossRef]
8. Guo, Q.; Xie, X.; Li, Y.; Zhang, X.; Liu, Y.; Li, X.; Shen, C. Audee: Automated Testing for Deep Learning Frameworks. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), Virtual, 21–25 December 2020.
9. Du, X.; Sui, Y.; Liu, Z.; Ai, J. An Empirical Study of Fault Triggers in Deep Learning Frameworks. *IEEE Trans. Dependable Secur. Comput.* **2022**, *20*, 2696–2712. [CrossRef]
10. Qiu, J.; Chai, Y.; Tian, Z.; Du, X.; Guizani, M. Automatic Concept Extraction Based on Semantic Graphs From Big Data in Smart City. *IEEE Trans. Comput. Soc. Syst.* **2020**, *7*, 225–233. [CrossRef]
11. Zhou, L.; Li, J.; Gu, Z.; Qiu, J.; Gupta, B.B.; Tian, Z. PANNER: POS-Aware Nested Named Entity Recognition Through Heterogeneous Graph Neural Network. *IEEE Trans. Comput. Soc. Syst.* 2022, *early access*. [CrossRef]
12. Li, J.; Cong, Y.; Zhou, L.; Tian, Z.; Qiu, J. Super-resolution-based part collaboration network for vehicle re-identification. *World Wide Web* **2023**, *26*, 519–538. [CrossRef]
13. Tian, Z.; Luo, C.; Qiu, J.; Du, X.; Guizani, M. A Distributed Deep Learning System for Web Attack Detection on Edge Devices. *IEEE Trans. Ind. Inform.* **2020**, *16*, 1963–1971. [CrossRef]
14. Tian, Z.; Gao, X.; Su, S.; Qiu, J.; Du, X.; Guizani, M. Evaluating Reputation Management Schemes of Internet of Vehicles Based on Evolutionary Game Theory. *IEEE Trans. Veh. Technol.* **2019**, *68*, 5971–5980. [CrossRef]
15. Gojare, S.; Joshi, R.; Gaigaware, D. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Comput. Sci.* **2015**, *50*, 341–346. [CrossRef]

16. Needleman, S.B.; Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* **1970**, *48*, 443–453. [CrossRef] [PubMed]
17. Gu, J.; Luo, X.; Zhou, Y.; Wang, X. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 25–27 May 2022; pp. 1418–1430. [CrossRef]
18. Wang, Z.; Yan, M.; Chen, J.; Liu, S.; Zhang, D. Deep Learning Library Testing via Effective Model Generation. In Proceedings of the ESEC/FSE 2020: 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual, 8–13 November 2020; pp. 788–799.
19. Jia, L.; Zhong, H.; Huang, L. The Unit Test Quality of Deep Learning Libraries: A Mutation Analysis. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 47–57. [CrossRef]
20. Wei, A.; Deng, Y.; Yang, C.; Zhang, L. Free Lunch for Testing:Fuzzing Deep-Learning Libraries from Open Source. In Proceedings of the ICSE '22: 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022.
21. Deng, Y.; Yang, C.; Wei, A.; Zhang, L. Fuzzing deep-learning libraries via automated relational API inference. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022.
22. Zhang, X.; Liu, J.; Sun, N.; Fang, C.; Liu, J.; Wang, J.; Chai, D.; Chen, Z. Duo: Differential Fuzzing for Deep Learning Operators. *IEEE Trans. Reliab.* **2021**, *70*, 1671–1685. [CrossRef]
23. Pei, K.; Cao, Y.; Yang, J.; Jana, S. DeepXplore: Automated whitebox testing of deep learning systems. In Proceedings of the SOSP '17: 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; pp. 1–18. [CrossRef]
24. Ma, L.; Xu, F.J.; Zhang, F.; Sun, J.; Xue, M.; Li, B.; Chen, C.; Su, T.; Li, L.; Liu, Y.; Zhao, J.; Wang, Y. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In Proceedings of the ASE '18, Montpellier, France, 3–7 September 2018.