DiffWatch: Watch Out for the Evolving Differential Testing in Deep Learning Libraries

Alexander Prochnow
University of Osnabrück
Osnabrück, Germany
aprochnow@uos.de

Jinqiu Yang
Concordia University
Montreal, Canada
jinqiu.yang@concordia.ca

ABSTRACT

Testing deep learning libraries is ultimately important for ensuring the quality and safety of many deep learning applications. As differential testing is commonly used to help the creation of test oracles, its maintenance poses new challenges. In this tool demo paper, we present *DiffWatch*, a fully automated tool for Python, which identifies differential test practices in DLLs and continuously monitors new changes of external libraries that may trigger the updates of the identified differential tests.

Our evaluation on four DLLs demonstrates that *DiffWatch* can detect differential testing with a high accuracy. In addition, we demonstrate usage examples to show *DiffWatch*'s capability of monitoring the development of external libraries and alert the maintainers of DLLs about new changes that may trigger the updates of differential test practices. In short, *DiffWatch* can help developers adequately react to the code evolution of external libraries. *DiffWatch* is publicly available and a demo video can be found at https://www.youtube.com/watch?v=gR7m5QQuSqE.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software libraries and repositories**.

KEYWORDS

Software Quality Assurance, Software Testing, Testing Deep Learning Libraries, Differential Unit Testing

ACM Reference Format:
Alexander Prochnow and Jinqiu Yang. 2022. DiffWatch: Watch Out for the Evolving Differential Testing in Deep Learning Libraries. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516835

1 INTRODUCTION

Deep Learning (DL) models play an important role in many life-critical software domains. Deep learning libraries (DLLs), the software libraries that support the creation and maintenance of DL models, are becoming more and more important in the maintenance of intelligent software systems. As illustrated by previous work [4], testing DLLs is different from traditional software in many aspects, one of which is the extensive use of differential testing. Developers of DLLs commonly utilize the output of other scientific libraries or even other DLLs, e.g., Numpy and Keras, as oracles when testing one DLL. The extensive use of differential testing helps ease the oracle challenge but also poses new maintenance challenges, i.e., developers/maintainers of the DLLs need to be constantly aware of new changes of the involved external libraries and how such changes may impact the use of differential testing in the DLL under maintenance. Failing to evolve with such changes results in outdated tests and flaky tests that may hinder the quality assurance effort of DLLs.

To help maintainers of DLLs be better alerted about the potentially affecting changes of external libraries, in this tool demo paper, we present *DiffWatch*, which continuously monitors new changes of external libraries upon automated detection of differential testing practice.

First, *DiffWatch* catalogs differential test cases, which are unit tests that compare the output of the code under test with the output of an equivalent function from an external library. This type of testing has a particular advantage for deep learning libraries because it allows an easy comparison of mathematical computation outputs which were generated by external libraries instead being hard coded in test cases by DLL developers.

Second, *DiffWatch* monitors external library repositories (e.g., other DLLs or scientific libraries) that are relevant to the differential testing practice of the DLL under test. Specifically, this tool shows relevant `git diffs` for each cataloged differential test case when a relevant function in an external library is changed. This should help developers make an informed decision on whether an update to their differential test cases as a result of this external code evolution is required.

We present an implementation of *DiffWatch* for Python and evaluate its accuracy of identifying differential test cases and monitoring impactful changes of external libraries. The evaluation is performed on a total of five versions of DLLs, i.e., TensorFlow 2.6.0, TensorFlow 1.12.0, PyTorch 1.9.0, Keras 2.6.0, and Theano 1.0.3. The evaluation shows that *DiffWatch* achieves a reasonable accuracy in identifying differential test cases We make an easy-to-run version publicly available using Jupyter notebook[1] and plan to release on the GitHub marketplace.

Paper organization. Section 2 presents a brief background of differential testing and related work. Section 3 illustrates the design and implementation of *DiffWatch*. Section 4 presents the evaluation

[1] https://github.com/AlexanderProchnow/DLL_Testing_Tool

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516835

of *DiffWatch* on four DLLs. Section 5 showcases usage examples of *DiffWatch*. Last we conclude in Section 6.

2 BACKGROUND AND RELATED WORK

Popular deep learning libraries such as TensorFlow, PyTorch and Keras as well as previous libraries such as Theano differ from more traditional software in the way their testing is done, specifically their unit testing [4]. This is, among other reasons, due to the so-called Oracle Problem. This problem is generally prevalent in the testing of numerical software. Since DLLs heavily encode mathematical formulas, the problem is encountered here as well. The Oracle Problem describes the fact that we oftentimes cannot find exact solutions to numerical computations. This is due to the limitations of computing machinery, which result in floating point errors due to the limited digital representation of floats as well as truncation errors due to the memory limitations (e.g. truncating infinite series) [1].

This problem derives its name from the test oracle. The oracle is the expected output which the actual output of the code under test is compared against. As the Oracle Problem describes, we might not always have an exact oracle to test the code against for equality. To solve this, deep learning libraries use oracle approximations. Instead of requiring exact solutions, oracle approximation unit tests check if the output of the code under test is within certain margins of the oracle [4].

Previous research [4] analyzes these oracle approximation testing practices in popular DLLs and characterizes the various types of oracles found. The identified types of oracles include defined oracles, naive implementation, same code under test (i.e. metamorphic testing), relevant internal function, and differential testing. We refer readers to the details in the original paper.

In this study, we focus on differential testing [3], which represents about 5-27% of all the testing practices in popular DLLs [4]. We focus on this type of testing because it can be used to compare the outputs of models from multiple frameworks to find inconsistencies and reveal bugs.

Previous work Audee [2] showcases this utilization of differential testing, which automatically generated differential tests and found 5 previously unidentified bugs in TensorFlow, CNTK and Keras. Similarly, CRADLE [5] was developed to identify the functions that caused the inconsistencies to arise. Both Audee and CRADLE are built to find bugs by leveraging differential testing. However, the challenges of maintaining existing differential testing is not yet addressed.

3 THE DESIGN AND IMPLEMENTATION OF DIFFWATCH

Figure 1 shows an overview of our tool. *DiffWatch* takes a set of inputs including the information of the DLL under test, as well as the information of the external libraries.

Here, we will use the example of a TensorFlow developer who would like to integrate the newest Keras version into TensorFlow. For this they need to check if the new Keras version impacts any of their differential tests. So they use our python package via the functions shown in **Listing 1** to run our tool, which automatically identifies differential test cases in their library and stores them in

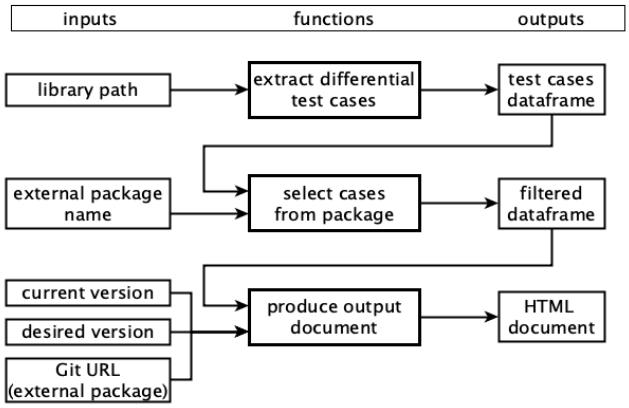

Figure 1: The Overview of DiffWatch

a Pandas dataframe (*line 1*). This dataframe will contain entries similar to **Table 1** for each identified test case.

Listing 1: Tool example

```
1 df_test_cases = extract_diff_test_cases(library_path)
2 filtered_df = select_cases_from_package(df_test_cases,
      'keras')
3 produce_output_doc(filtered_df, current_date,
      desired_date, package_git_url, save_to)
```

Table 1: Extracted test case

File_Path	kernel_tests/distributions/gamma_test.py
Line_Number	77
Found_in_Function	testGammaLogPDF
Function_Definition_Line_Number	62
Assert_Statement_Type	assertAllClose
Oracle_Argument_Position	2
Differential_Function_Line_Number	76
Differential_Test_Function	stats.gamma.logpdf

Of particular interest here is the last field, which contains the extracted external function. In *line 2* we filter those entries for ones using Keras functions. The tool can then produce a HTML report (*line 3*) which includes all the chosen test cases as well as the `git diffs` relevant to each case. These `git diffs` compare the desired version of the external library (Keras in this case) with the current version used by the deep learning library (TensorFlow in this case). **Figure 2** shows an example of a generated output document, which we will go into detail on in the results section. Further examples can be found in the *repository accompanying this demo*. To view this, download the HTML and open it in a browser.

Now we examine the specifics, beginning with the automatic cataloging of differential test cases and afterwards moving on to the production of the output document.

3.1 Identifying differential test cases

In order to better understand what our tool should catalog, we will define differential test cases as (1) being denoted by an assert statement and (2) having the oracle, i.e. the 'expected' value, be

derived from an external function or influenced by an external function at some point in the computation. Take this excerpt of a simple differential test case from a recent version of TensorFlow (version 2.6.0):

Listing 2: TensorFlow's gamma_test.py example

```
69 gamma = gamma_lib.Gamma(concentration=alpha, rate=beta)
70 log_pdf = gamma.log_prob(x)
   ...
   expected_log_pdf = stats.gamma.logpdf(x, alpha_v, scale=1
       / beta_v)
   self.assertAllClose(self.evaluate(log_pdf),
       expected_log_pdf)
```

In **Listing 2** we can see our two previously defined criteria fulfilled: The assert statement in the last line compares `log_pdf`, which contains TensorFlow's implementation of a gamma distribution's logarithmic probability density function with `expected_log_pdf`, which contains the implementation found in the external library SciPy, specifically in its `stats` module. Both distributions are initialized with equivalent shape and scale parameters. They then receive an array of floats `x` as input and return the log probability density at those values. Since the returned values might deviate slightly due to e.g. floating point errors, `assertAllClose` checks if all value pairs are close within a certain default margin.

Now that we have seen our first differential test case, let us examine the algorithm by which our tool identifies cases like this automatically. The algorithm uses the Abstract Syntax Tree (AST) generated from a given source code file. An abstract syntax tree represents source code as a tree, where each node is an element of the source code, e.g. a function definition, function call, if-else statement, while-loop, variable assignment etc. This structure makes identifying differential test cases easier in comparison to identifying them in the raw source code. To get an idea what these AST nodes look like, **Listing 3** shows the AST of the last line, i.e. the assert call, of our previous test case example.

Listing 3: The last line of Listing 2 in AST-representation

```
Call(
    func=Attribute(
        value=Name(id='self', ctx=Load()),
        attr='assertAllClose',
        ctx=Load()),
    args=[
        Call(
                func=Attribute(
                        value=Name(id='self', ctx=Load()),
                        attr='evaluate',
                        ctx=Load()),
                args=[
                        Name(id='log_pdf', ctx=Load())],
                keywords=[]),
        Name(id='expected_log_pdf', ctx=Load())],
    keywords=[])
```

To parse the entire source code files and traverse the trees, we use the `ast` python package, which also produced the printed AST in **Listing 3**. We built a custom `NodeVisitor` on top of the `ast` library, which traverses a tree in the manner described in **Algorithm 1**.

The essential part of **Algorithm 1** is found in *lines 5 and 6*. How the algorithm does this part is dependent on what types of nodes it finds as the arguments for the assert statement. A detailed explanation of how the algorithm handles different node types and an annotated version of the algorithm in its entirety can be found as a *Jupyter Notebook in the GitHub repository*. Lastly we run this algorithm on all test files of a given library, creating a catalog of all of its differential test cases. In the following subsection, we will now examine how this catalog can be used to find relevant changes in external libraries and produce an output document linking these changes to the test cases that they affect.

Algorithm 1: Extracting differential test cases from one source code file

Input : rootASTNode that represents the source code file
Output: dataFrame that contains the information of differential test cases in the source code file

```
1  foreach astNode ∈ rootASTNode.children do
2      if astNode.isAssertStatement then
3          arguments ← astNode.getArguments()
4          foreach argument ∈ arguments do
5              value ←
                 rootASTNode.lastAssigned(astNode, argument)
                 ▷ lastAssigned function identifies the last value
                 assigned to argument before astNode;
6              dataFrame.append(value, getType(rootASTNode,
                 value))
7                ▷ getType function checks the origin of value,
                 whether it is from computed results, an
                 external function or an internal function.
8          end foreach
9      end if
10 end foreach
```

3.2 Monitoring external library evolution

In order to monitor external library changes and display relevant ones to the user, our tool requires the `git URL` of the repository that it should monitor (e.g. 'https://github.com/tensorflow/tensorflow.git'). It then requires this library to be loaded in order to identify in which file an extracted differential testing function is defined. We do this using the python package `inspect`, specifically its `inspect.getsourcefile` function, to which we pass the value of the `Differential_Test_Function` field for each test case. For our example in **Listing 1**, this returns `scipy/stats/gamma_dist.py`. With respect to this file, we then compare the current version of the package used in the deep learning library with the desired version the developer would like to upgrade to by comparing their respective commits via a `git diff`. We then format its output and write it to our output document.

Figure 2 shows the general structure of an output document generated by our tool with annotations 1 through 4. At the beginning of the document, we show the commit ids corresponding to the current and desired version as well as their commit messages *(annotation 1)*. The document is then split up into sections. Each section contains information on all differential test cases whose external functions are defined in the same source file. Within a section, we firstly list all of these test cases together with the information stored about them in our dataframe *(annotation 2)*. Secondly the section then contains a collapsible subsection *(annotation 3)* which shows the formatted output of our `git diff` *(annotation 4)*.

Table 2: Differential test case extraction results

	TensorFlow 2.6.0	TensorFlow 1.12.0	PyTorch 1.9.0	Keras 2.6.0	Theano 1.0.3
Total data entries	45,336	43,568	33,111	9,961	4,024
Of those unsupported	4619 (10.19%)	1238 (2.84%)	2856 (8.63%)	918 (9.22%)	445 (11.06%)
Correctly identified entries					
After filtering for external functions	54% (= $\frac{27}{50}$)	72% (= $\frac{36}{50}$)	96% (= $\frac{48}{50}$)	92% (= $\frac{46}{50}$)	96% (= $\frac{48}{50}$)

In the following two sections, we will examine and evaluate the extracted test cases from various deep learning libraries. Additionally, we will have a look at examples from the past where our tool could have aided developers in managing external code evolution.

4 EVALUATION

In order to evaluate the tool, we will present data on the differential test case extraction results for TensorFlow (TF) version 2.6.0 as well as version 1.12.0, when TensorFlow and Keras were still developed independently. We selected version 1.12.0 as well because it was previously studied in the Oracle Approximation paper [4]. In this version especially, we find good examples of differential testing (TF using Keras functions as oracles), which will be covered in detail in the results section. We will also quickly examine the results of our differential test case extraction in Keras 2.6.0, PyTorch 1.9.0 and Theano 1.0.3.

Table 2 presents collected statistics about the test case extraction done by our tool. The first row shows the number of entries in our dataframe's CSV file, with each entry containing information as seen in **Table 1**. The second row shows how many of those test cases were identified, but the correct extraction of the differential testing function is not yet covered by **Algorithm 1**'s logic. This is either denoted by an "UNSUPPORTED ..." statement in the `Differential_Test_Function` column of the data or by an empty string in this column, i.e. NaN. This is an area for future improvement. Detailed statistics about this can be found in the *Data_evaluation notebook* or by using the `coverage_analysis` function of our python package.

The last row contains statistics on a manual evaluation we performed. We randomly sampled 50 data entries from the data our tool produced for each deep learning library filtered for functions from external libraries that our users would commonly filter for

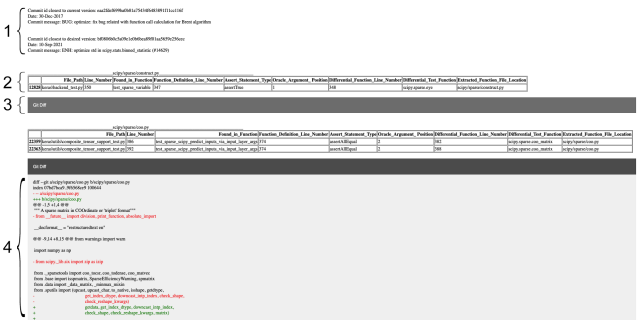

Figure 2: Excerpt of a tool output document, shown to provide a general overview of the document's structure

(i.e. SciPy, Keras etc.). For each sampled data entry, we then manually decided if it correctly identified a differential test case and a differential testing function as defined above. Our evaluations are stored in an `Evaluation` column in the data. The *evaluation data* for each library can be found in the repository.

Lastly, we evaluated if the developed differential test case extraction algorithm would generalize to other libraries. For this we tested the algorithm on the libraries NumPy and SciPy without making any adjustments to it. The algorithm was able to extract 26695 data entries from NumPy and 285 from SciPy, of which 9132 were unique test cases for NumPy and 69 for SciPy. This might be a first indication that the tool could also be useful for other software libraries, however this requires more evaluation.

5 USAGE EXAMPLES

Overall, we found that every DLL we tested does differential testing in some capacity with NumPy and SciPy. For example in PyTorch 1.9.0, we found *17 test cases* and in TensorFlow 2.6.0 we found *57 test cases* that use SciPy functions. Most notably, this is done in order to compare probability distribution implementations, similar to the example given previously (Table 1 and Listing 2). NumPy, on the other hand, is mostly used for array related tests. Here, we found 573 test cases using NumPy functions in PyTorch 1.9.0 and 2894 test cases in TensorFlow 2.6.0.

Differential tests focused on e.g. the neural network architectures and computations were only found rarely. One prominent file, however, that included such differential test cases was found in TensorFlow version 1.12.0. Here TF tested its Recurrent Neural Network implementation against that of Keras in the kernel_tests/rnn_test.py file e.g. the test case that concludes with the assert statement in line 574. All extracted information about the differential test cases the tool identifies in this file can be found in the *rnn_test output document* in the repository. `Git diffs` similar to the ones included in this document may be of use for managing future differential testing.

6 CONCLUSIONS

As shown earlier, deep learning libraries can be improved by including differential testing, since this might detect errors that would not be found by regular unit tests. The tool we created could then help developers manage their differential test cases and aid in maintaining them during external code evolution.

There are still challenges ahead in the development of this tool. Namely, decreasing the amount of identified test cases that are not differential test cases. For this a more extensive evaluation of the identified test cases could yield valuable insights into how this may be accomplished. Additionally, between 3-11% of found AST node

structures are currently unsupported (Table 2). To improve this, an evaluation of these test cases and a subsequent expansion or generalization of the algorithm's logic could be done. Lastly, the limitation to Python source files could be removed by using a general Abstract Syntax Tree format that can be generated from other programming languages as well. This would expand the applicability of the tool to other deep learning and software libraries.

REFERENCES

[1] T.Y. Chen, Jianqiang Feng, and T.H. Tse. 2002. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings 26th Annual International Computer Software and Applications*. 327–333. https://doi.org/10.1109/CMPSAC.2002.1045022

[2] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 486–498. https://doi.org/10.1145/3324884.3416571

[3] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[4] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 785–796. https://doi.org/10.1109/ASE.2019.00078

[5] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107