This is the authors' version of the paper that has been accepted for publication in the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) 2022

Detect, Fix, and Verify TensorFlow API Misuses

Wilson Baker
University of Auckland
Auckland, New Zealand
wbak037@aucklanduni.ac.nz

Michael O'Connor
University of Auckland
Auckland, New Zealand
moco657@aucklanduni.ac.nz

Seyed Reza Shahamiri
University of Auckland
Auckland, New Zealand
admin@rezanet.com

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

***Abstract*—The growing application of DL makes detecting and fixing defective DL programs of paramount importance. Recent studies on DL defects report that TensorFlow API misuses represent a common class of DL defects. However to effectively detect, fix, and verify them remains an understudied problem. This paper presents the TensorFlow API misuses Detector And Fixer (TADAF) technique, which relies on 11 common API misuses patterns and corresponding fixes that we extracted from StackOverflow. TADAF statically analyses a TensorFlow program for identifying matches of any of the 11 patterns. If it finds a match, it automatically generates a fixed version of the program. To verify that the misuse brings a tangible negative effect, TADAF reports functional, accuracy, or efficiency differences when training and testing (with the same data) the original and fixed versions of the program. Our preliminary evaluation on five GitHub projects shows that TADAF detected and fixed all the API misuses.**

***Index Terms*—Deep Learning, TensorFlow, Static bug detection, APIs, Automated Program Repair, Differential Testing**

I. Introduction

Deep Learning (DL) has become an increasingly popular and promising Machine Learning (ML) method that often achieves human-like performance in a wide range of areas. DL is often used where failures or unexpected behaviors can have catastrophic consequences, such as self-driving cars [1] or medical diagnostic [2]. For this reason, exposing defects in DL programs is an extremely important problem [3].

Developers often rely on ML frameworks to efficiently design and train DL models. The TensorFlow library [4] is the most popular of such frameworks [5]. TensorFlow is continually extending its dominance over other DL frameworks (e.g., PyTorch and Theano). As in January 2022, the GitHub repository of TensorFlow is the top 10 most starred repositories in GitHub with over 160K stars [6].

TensorFlow exposes an Application Programming Interface (API) that developers use to implement a ML pipeline that trains and tests a DL model. Like any other library, developers can violate the (implicit) usage constraints of the TensorFlow API, introducing *API misuses*.

Figure 1 shows a TensorFlow API misuse and related fix, which was discussed in the StackOverflow question #38589255. The API misuse is calling `train_data.eval()` without calling the TensorFlow API `tf.train.start_queue_runners(sess)` before. This is a violation of an implicit usage constraint of the API, which requires that (for this particular input pipeline) developers specify when TensorFlow should start fetching the records into the buffers. Without the call at line 4, the buffers remain empty and the API call `eval()` at line 5 hangs indefinitely.

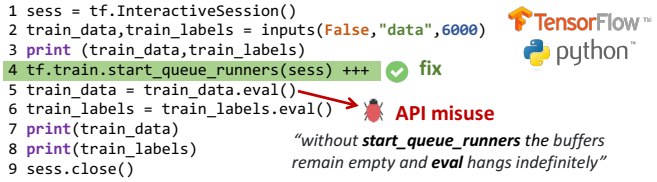

Fig. 1. TensorFlow API misuse - https://stackoverflow.com/q/38589255

TensorFlow users introduce these kind of bugs when they do not fully understand the assumptions made by the APIs [5], [7], [8]. Surprisingly, even popular and well-maintained DL programs suffer from TensorFlow API misuses [5], [7], [9]. Zhang et al.'s bug characteristic study classifies 175 TensorFlow program bugs collected from StackOverflow and GitHub. The results show that TensorFlow API misuses is a common type of TensorFlow bugs [5]. Further reinforcing this, Islam et al.'s study reports that most of the non-model related TensorFlow bugs are due to API misuses [9].

Recently, the empirical study of Humbatova et al. [7] indicates that DL users recognize several TensorFlow API misuses to be common misconceptions. This is confirmed by the high community engagement of those StackOverflow posts that discuss TensorFlow API misuses [7]. This brings the opportunity to identify common TensorFlow API misuses from StackOverflow and GitHub to define patterns for the automated detection and fix of the misuses. Although there are static bug detectors for Python [10]–[13], including TensorFlow-specific static bug detectors [14], [15], none of these techniques target TensorFlow API misuses nor aim to fix or dynamically verify the detected bugs.

This paper presents the early research achievement *TensorFlow API misuses Detector And Fixer (TADAF)*, the first technique to automatically detect, fix, and verify TensorFlow API misuses. TADAF is grounded by 11 common API misuses patterns and related fixes that we mainly extracted from StackOverflow posts. TADAF statically analyses a TensorFlow program to identify instances of such patterns and automatically fix them. Then, it employs differential testing [16] to verify that the fixed version works properly and that there are indeed behavioural differences between the original and fixed versions. The API misuse in Figure 1 is an example of a misuse that TADAF successfully detects, fixes, and verifies. We evaluated our proposed approach on five GitHub projects, showing that TADAF correctly detected and fixed all misuses and successfully exposed behavioural differences in four projects.

TABLE I
11 PATTERNS OF TENSORFLOW API MISUSES CURRENTLY IMPLEMENTED IN TADAF

ID	Category	API misuse	Related fix
P1	Low Effectiveness	Call `decode_jpeg()` passing a `png` file as an argument	Replace the `decode_jpeg()` API call with the more inclusive `decode_image()` API call
P2	Low Effectiveness	Same as P1 but the code contains `image.resize()` API call, which has to be adjusted as well	Same fix as P1 plus replace `image.resize()` with `image.resize_image_with_crop_or_pad()`
P3	Deprecated API Error	Use of deprecated API call `tf.merge_summary()`	Replace it with the working version of the same API call `tf.summary.merge()`
P4	Deprecated API Error	Use of deprecated API call `tf.merge_all_summaries()`	Replace it with the working version of the same API call `tf.summary.merge_all()`
P5	Deprecated API Error	Use of deprecated API call `tf.train.SummaryWriter()`	Replace it with the working version of the same API call `tf.summary.FileWriter()`
P6	Error	Use of binary class mode but the final dense layer is set to a value which is not 2. For example, `Dense(3, activation='softmax')`	Find the last dense layer and change the value to 2. For example, `Dense(2, activation='softmax')`
P7	Low Efficiency	Use of `tf.nn.softmax()` in conjunction with a cross entropy formula leads to slowdowns	Combine both parts into a single function call `tf.nn.softmax_cross_entropy_with_logits()`
P8	Low Efficiency	Calling `@tf.function` many times in a for loop leads to slowdowns	Ensure that the parameters passed to the API are assigned using the `tf.range()` method as well as casting it using `tf.cast()`
P9	Deprecated API Error	Use of deprecated API call `tf.histogram_summary()`	Update to working version of same API call `tf.summary.histogram()`
P10	Deprecated API Error	Use of deprecated API call `tf.scalar_summary()`	Update to working version of same API call `tf.summary.scalar()`
P11	Low Efficiency	Calling the `eval()` API before calling `tf.train.start_queue_runners(x)` leads to hangs	Insert a call to `tf.train start_queue_runners(x)` before `eval()` and after the most recent assignment of variable `x`

II. TENSORFLOW API MISUSES PATTERNS

This section presents our methodology to identify an initial set of 11 TENSORFLOW API misuses patterns and related fixes. An important future work is to expand such a set to further improve the completeness of TADAF.

To obtain a relevant set of initial patterns, we aimed at having at least one pattern for each type of bugs that can occur from TENSORFLOW API misuses. Zhang et al.'s study classifies the root causes of TENSORFLOW program bugs (including API misuses) into three categories [5]:

- *Error*: Any bug that causes the program to crash often raising an exception while doing so.
- *Low Effectiveness*: Any bug that negatively affects the accuracy of the model or reduces the quality of the result.
- *Low Efficiency*: Any bug that causes the training or testing phase to take a considerably longer time than expected. This includes bugs that lead to infinite executions.

Any TENSORFLOW API misuses that has a behavioral effect on a DL programs must fall into these three categories [5], [9]. We also considered an additional sub-category of error bugs:

- *Deprecated API Error*: Errors bugs that are documented by TENSORFLOW as there exists a documentation on the change as well as warnings leading up to the change.

In the context of API misuses, we thought it was best to separate deprecated API errors from general ones since TENSORFLOW developers already specified in the API documentation how to detect and fix the API misuses.

We used the following process to find at least one API misuse for each of the four categories:

First, we examined what TENSORFLOW API misuses had been studied before. For this, we looked to the Zhang et al.'s [5] and Humbatova et al.'s [7] studies and listed down every API misuses that they identified, which were collections of StackOverflow posts or GitHub issues. This totalled 15 bugs. From there, we selected those involving TENSORFLOW, narrowing the number of API misuses from 15 to 7.

Then, we turned to searching StackOverflow for collecting additional TENSORFLOW API misuses. We retrieved all the questions with the tag `[tensorflow]` and then sorted in terms of views with the idea that the most commonly experienced bugs would have the highest number of views. We then inspected the search results and took care to note: (i) if it was a TENSORFLOW API misuse and which of the four categories it falls into by reading the comments and fitting the behaviour of the bug based on the category definitions posed above, (ii) when the original question was asked (to ensure relevancy in terms of time), and (iii) number of answers (another measure of community engagement).

We then selected the most popular TENSORFLOW API misuses in each of the categories based on our labelling of the data obtained in above (i). We only accepted those with at least one popular answer (5+ upvotes) and a minimum number of 5,000 views to the post. We found these thresholds by looking at all 14 TENSORFLOW bugs that were located and choosing the bugs in descending order based on number of views, number of upvotes and popular answers. We used the publish date of the StackOverflow question to help understand popularity lines. If two posts had the same number of views then the post with the earliest date would have higher popularity since the new post has accumulated more views in a shorter time.

After inspecting the collected API misuses we defined 11 TENSORFLOW API misuses patterns. Table I summarizes the pattern with a description of the misuse and its related fix.

III. TADAF

TADAF aims at automatically detect and fix TensorFlow APIs misuses. Figure 2 overviews the logical architecture of our proposed approach. TADAF takes as input a Python TensorFlow program that implements a DL pipeline, and a dataset for training and testing the generated DL model. If TADAF detects API misuses, it outputs the fixed version of the program and a report of the behavioral differences between the fixed and original versions.

TADAF has three main components: Finder, Fixer, and Verifier. The Finder statically analyses the program to identify instances of the 11 API misuse patterns. Then, the Fixer works together with the Finder to take those found instances and apply a common fix to the detected misuses creating a fixed version of the program. Finally, the Verifier performs differential testing using the dataset in input to verify that the fixed version works properly and that behavioural differences exist between the original and fixed versions.

A. Finder

The Finder component statically analyses the Abstract Syntax Trees (AST) [17] representation of the Python TensorFlow program in input, to identify instances of any of the API misuse patterns. By analyzing the AST, TADAF can unambiguously identify language constructs (e.g., function calls, method arguments), and thus precisely detect API misuses. We implemented each pattern of Table I as a dedicated AST analysis that takes the AST source tree of the program and returns a list of faulty AST nodes that contain all the relevant information needed to perform the fix. Going back to the example of Figure 1, we can break down how Finder detects the API misuse (P11 in Table I) step-by-step.

As APIs are all interacted through function calls, the first step to detect a pattern is to find every instance of a specific function. In the case of the motivating example, it is important to know all locations of the `eval()` and `start_queue_runners()` function calls. This is achieved through utility methods which "walk" through the nodes in the AST and return all nodes that match the right node type (`AST.Call`) and name (target function as a String).

Further analysis of the nodes is done to determine if they meet the criteria for the pattern. This could include comparing the order of certain functions or the type of information being passed as their arguments. For our example, Finder checks if there is an AST `eval()` function node and there are no `start_queue_runners()` function nodes before.

Finally, some extra information may be needed for the Fixer component to successfully make the fix. This could include locations of assignment nodes for certain variables or values given in arguments. The process for collecting this information is similar as above but instead focuses on detecting certain assignment nodes in the AST. For our example, we need to know where is the most recent assignment of the session variable (`sess` defined at line 1 in Figure 1). The Fixer needs such a variable to call `tf.train.start_queue_runners(sess)` and fix the misuse.

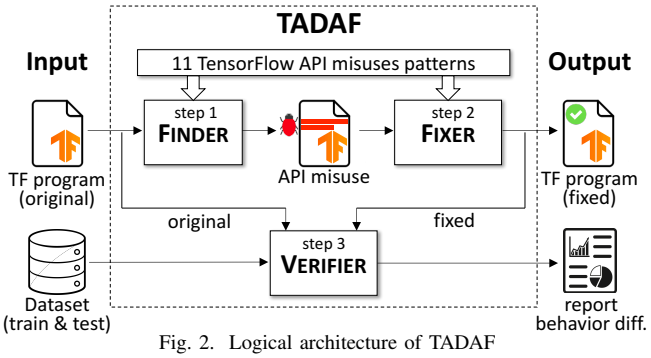

Fig. 2. Logical architecture of TADAF

B. Fixer

The Fixer component goes through the list of API misuses that Finder identified. For each fix pattern in Table I, we implemented a function that takes the AST of the program and the faulty node and returns the transformed AST. We considered three different AST transformations: (i) alter an existing node of the AST, (ii) remove an existing node of the AST, and (iii) add a new constructed node to the AST. Each of these transformations can be used in different combinations as many times that is needed to fix an API misuse.

In the example in Figure 1, The Fixer component creates a new `AST.Call` node that invokes the function `tf.train.start_queue_runners(sess)` and inserts the node in the AST right before the faulty node. TADAF automatically converts the transformed AST into Python source code, creating a fixed version of the TensorFlow program.

C. Verifier

The Verifier component checks that the fixed version works correctly and that the detected/fixed API misuses are real TensorFlow bugs that negatively affect the performance of the original program. We achieve this with *differential testing* [16], [18], which compares metrics before and after the fix. These comparison metrics are changes in accuracy of the model, changes in training/testing time, and any changes to the presence of exceptions/time-outs. TADAF collects these metrics by running the original and fixed version of the TensorFlow program on the same training and test dataset.

In the example of Figure 1, the call `eval()` hangs forever in the original version. TADAF reports either a time-out or an infinite training time depending on the underlying system.

IV. Preliminary Evaluation

To evaluate our proposed approach, we implemented a prototype tool in Python that relies on the `ast` module [17] to analyze and transform ASTs. We collected evaluation subjects by searching on GitHub for Python projects with the following characteristics: (i) it uses the TensorFlow framework to define, train, and test a DL model; (ii) there is at least one targeted misuse in the current or past versions; and (iii) it contains a dataset for training and testing the DL model. We used as search query the API calls related to the misuses, or a natural language description of the misuses. We found five projects that meet the requirements, covering seven unique

TABLE II
EVALUATION RESULTS OF TADAF

ID	FINDER			FIXER	VERIFIER (original version)			VERIFIER (fixed version)		
	# Misuses	Category	Time (ms)	Time (ms)	# Crashes	Time (s)	Accuracy	# Crashes	Time (s)	Accuracy
S1	1 (P1)	Low Effe.	2	21	0	6,869	low	0	7,094	high
S2	4 (P3, P5, P9)	Deprecated	12	3	4	-	-	0	<1	92%
S3	1 (P7)	Low Effi.	3	1	0	30	99%	0	28	99%
S4	1 (P11)	Low Effi.	8	7	0	∞	-	0	$\sim$1	98%
S5	1 (P6)	Error	3	6	1	-	-	0	5	78%

misuse patterns. Subject S2 contains four misuses, while other subjects one each. For S2 we considered four distinct copies of the program, each with a single misuse. To provide more information about the subjects and to facilitate future work in this area, we release our experimental data [19].

We then run TADAF on each of these five projects. It is important to mention that the VERIFIER component is still in the initial prototyping phase. It needs manual intervention to launch the original and fixed programs and to collect the performance metrics. Because most of our subjects contain their own metrics for testing models, such as printing test accuracy or saving outputted images, we decided to run and compare the performance metrics manually. An important future work is to make the VERIFIER fully automated. Also, we ran the VERIFIER one time only. To handle non-deterministic DL programs [3], the VERIFIER should run multiple times and report median results. We run our experiments on a NVIDIA RTX6000 GPU, AMD 3990X CPU, with 256GB RAM.

Table II shows the results. Column "FINDER" indicates the number of misuses detected and the corresponding patterns, their category, and the time taken by the FINDER component. Notably, we report the time to search on all the Python files of the projects. Column "FIXER" gives the time taken to create the fixed version of the faulty TENSORFLOW program. In just a few milliseconds, the FINDER and FIXER components successfully detected and fixed all the expected misuses.

Columns "VERIFIER (original version)" and "VERIFIER (fixed version)" show the behavioral differences between the executions of the original and fixed versions. We show three key performance metrics: the number of runtime exceptions, the training and testing time, and the accuracy results on the test set. Notably, we do not report the accuracy for the subjects S2, S4, S5 as the TENSORFLOW programs either crash or hang indefinitely. Subject S1 is a generative adversarial network that generates images. Developers did not compute any accuracy metric. As such, we report a qualitative judgment on the images: we observe that the fixed version leads to clearer images.

The results show that for all except one subject (S3), TADAF successfully confirmed that the detected misuse has a negative impact on the performance and functional correctness of the DL program. In such cases, VERIFIER exposed the expected category of the TENSORFLOW bug as documented in Table I. For S3, while TADAF does not expose any clear behavioral difference, we believe this could be due to the small size of the dataset. There may not have been enough time for differences in efficiency to show clearly in our results.

V. RELATED WORK

Research on exposing DL program bugs mostly focuses on testing [3], [20]–[22], while static bug detection for DL programs remains an understudied problem [3]. Python is the de facto language for implementing DL programs. Being a dynamically-type language, there are several challenges when statically analyzing Python code [15], [23]. In the context of static detection of TENSORFLOW API misuses, our evaluation results suggest that many of such challenges do not affect the precision and completeness of TADAF.

There are several static bug detectors to detect general Python bugs: MYPY [10], PYLINT [11], PYFLAKES [12], and PYTYPE [13]. Although these tools can find bugs in TENSORFLOW programs, they cannot detect TENSORFLOW API misuses but only generic Python bugs.

Static bug detection of TENSORFLOW-specific program bugs is limited to TENSORFLOW shape-related bugs [14], [15]. ARIADNE [14] is a static shape analysis for TENSORFLOW, which relies on WALA for statically analyzing Python programs. PYTHIA [15] relies on datalog-based static analysis to identify TENSORFLOW shape-related bugs. Differently, TADAF targets the detection of generic TENSORFLOW API misuses. Moreover, ARIADNE and PYTHIA do not aim to fix and dynamically verify the detected TENSORFLOW bugs.

Although Zhang et al. study [5] does not provide a solution for fixing TENSORFLOW program bugs, the approach they used for fixing the bugs manually is very similar to the approach that TADAF automatically implements. They mention in the discussion of their paper that *"analysing the root causes could be useful for further developing automated repair approaches"* [5]. However, it should be noted that they targeted more broadly with TENSORFLOW program bugs whereas we have specifically targeted TENSORFLOW API misuses. Automatic Program Repair (APR) [24] for DL programs is still an unexplored issue (interested readers can refer to the recent study of Islam et al. [25]). TADAF automatically fixes a common and critical type of DL bugs, and thus making the first steps towards APR for DL programs.

VI. CONCLUSIONS AND FUTURE WORK

This paper presented TADAF, the first technique to automatically detect, fix, and verify TENSORFLOW API misuses. We implemented TADAF in a prototype tool for Python programs that use TENSORFLOW, and evaluated it on five open-source programs found in GitHub. Our preliminary results

are promising, giving evidence of the TADAF's effectiveness. In the future, we will conduct a rigorous evaluation to give more evidence of the effectiveness of our proposed approach.

TADAF is still at an early stage of inception. There are several promising future works and novel research directions that will likely turn TADAF into a fully-fledged automated static bug finder and fixer for TENSORFLOW programs.

I. While the FINDER and FIXER components of TADAF are already fully automated, the VERIFIER requires some manual intervention. More specifically, the VERIFIER relies on logging statements in the analyzed TENSORFLOW program to collect behavioral information (e.g., classification accuracy, execution time) when running the DL pipeline. At the moment, these statements are manually added. In the future, they should be automatically added by TADAF.

II. Once the VERIFIER is fully automated, we could automatically discover new API misuse patterns by mining TENSORFLOW programs from GitHub. In particular, given a commit that changes TENSORFLOW APIs calls, the VERIFIER could automatically recognize if the program versions before and after the commit manifest behavioral differences. Statistical analysis on many of such commits might help to distinguish common API misuses from project-specific faults.

III. Each API misuse pattern is unique. Thus, we had to implement a dedicated AST analysis for detecting and fixing each pattern. However, we noticed that the different implementations share common operations on the ASTs. For example, searching for an API call, checking the values of parameters, ensure the presence of certain statements. This brings an important opportunity: creating a domain specific language to represent patterns and related fixes [26]. Given such a representation, TADAF could automatically map language constructs to parameterized ASTs operations. This will facilitate the definition of new API patterns and avoid the manual cost of implementing new patterns into TADAF.

IV. Currently, TADAF assumes that a train and test dataset is both available and adequate to verify the detected API misuses and related fixes. This might not be the case. An important future work is to study data generation [27] and adequacy criteria [28] in the context of TENSORFLOW API misuses.

REFERENCES

[1] Q. Rao and J. Frtunikj, "Deep learning for self-driving cars: Chances and challenges," in *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*, 2018, pp. 35–38.

[2] L. C. Crossman, "Leveraging deep learning to simulate coronavirus spike proteins has the potential to predict future zoonotic sequences," *bioRxiv*, 2020.

[3] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

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

[6] Google Brain Team, "GitHub repository of TensorFlow," https://github.com/tensorflow/tensorflow.

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

[8] X. Du, G. Xiao, and Y. Sui, "Fault triggers in the tensorflow framework: An experience report," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 1–12.

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

[10] "mypy," http://mypy-lang.org/.

[11] "pylint," https://www.pylint.org/.

[12] "pyflakes," https://github.com/PyCQA/pyflakes.

[13] "pytype," https://google.github.io/pytype/.

[14] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: analysis for machine learning programs," in *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 1–10.

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

[16] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[17] Python, "AST module," https://docs.python.org/3/library/ast.html#module-ast.

[18] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[19] "TADAF - Experimental Data," https://doi.org/10.5281/zenodo.5831266.

[20] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: mutation testing of deep learning systems based on real faults," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 67–78.

[21] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[22] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[23] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 470–481.

[24] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

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

[26] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177.

[27] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 486–498.

[28] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1039–1049.