Received: June 15, 2022. Revised: July 22, 2022.

INASS

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

Inside A Deep Learning Library, the Symptoms, Causes, and Fixes for Bugs

Anjali C[1]* Julia Punitha Malar Dhas[2] J. Amar Pratap Singh[1]

[1]*Department of Computer Science and Engineering, Noorul Islam Centre for Higher Education, Kumaracoil, Tamil Nadu, 629180, India*
[2]*Department of Computer Science and Engineering, Karunya Institute of Technology and Sciences, Coimbatore, Tamil Nadu, 641114, India*
* Corresponding author's Email: anjalic0753@gmail.com

Abstract: Deep learning has become a popular study area in recent years. While deep learning software generates tremendous positive results in some cases, faults in the programme can have fatal repercussions, in safety-critical applications, this is extremely significant. Researchers have conducted various empirical investigations on defects in deep learning systems to better understand the bug characteristic of deep learning software. Despite the fact that these studies are useful, none of them investigate the problem in TensorFlow, a deep learning framework. They feel that some fundamental concerns about deep learning library difficulties remain unsolved. The answers to these questions are crucial and helpful because they constitute the underlying library of many deep learning endeavours. Many deep learning initiatives are influenced by its flaws. In this paper we pre-process a dataset that is openly accessible by performing data normalization and log transformation. To prepare the data input for the deep learning model, we next undertake data modelling. Third, we send the modelled data to a deep neural network-based model that was specifically created to forecast the number of faults. We test the suggested method on two well-known datasets as well. The evaluation's findings show that the suggested strategy is reliable and can outperform cutting-edge methods. The suggested approach enhances the squared correlation coefficient by more than 8% while, on average, drastically reducing the mean square error by more than 14%.

Keywords: Bug analysis, Deep learning, Empirical study, Tensorflow.

1. Introduction

Deep learning has been a prominent study topic in recent years, and academics have employed deep learning approaches to solve challenges in a variety of fields, including software analysis. Instead, then recreating the wheel while developing deep learning applications, programmers frequently rely on existing libraries. Because deep learning libraries are so common, one bug in one can propagate to bugs in many applications, which can have severe implications. For example, due to faults in their deep learning algorithms, A Google self-driving car collided with a Tesla automobile. Researchers have undertaken empirical investigations on deep learning programme faults to better understand them. in particular, conduct an empirical investigation to better understand TensorFlow application issues.

In this method we proposed and implemented using tensor Flow applications are programs that leverage TensorFlow's APIs. Rather than focusing on TensorFlow applications, look into other deep learning tools like Caffe and Torch. Although their findings are beneficial in improving the quality of a single application, we are unaware of any other studies that have looked into the problems found in popular deep learning frameworks. Although TensorFlow flaws affect thousands of applications, many questions about these bugs remain unanswered. Numerous applications will benefit from a deeper understanding of such issues, However, because TensorFlow combines numerous sophisticated algorithms and is implemented in multiple programming languages, conducting the

needed empirical study is difficult. In earlier work, we conducted the first empirical analysis of TensorFlow flaws. In comparison to this work, our enhanced version adds two new features: Previously, they had just looked at the symptoms and causes of difficulties; however, in this expanded edition, we looked at bug fixes as well as a few linguistic flaws.

2. Related work

Programmers frequently base their deep learning applications on established libraries rather than creating the wheel. Tensor-Flow is the most well-liked of these libraries, and a recent analysis reveals that more than 36,000 Git Hub projects are based on Tensor Flow. Due to their widespread use, deep learning libraries are prone to defects that might spread to several applications and have fatal results. Deep learning has become a popular study area in recent years. While deep learning software generates tremendous positive results in some cases, faults in the programme can have fatal repercussions, especially when it is utilised in safety-critical applications [1]. From embedded systems to smart homes, IoT systems are fast gaining traction. Despite their increasing popularity and usage, no comprehensive study has been conducted to evaluate IoT development problems from the perspective of practitioners [2]. We introduce ARCANA, a novel approach, in this paper. We use ARCANA to figure out why an autoencoder is reporting anomalies. The reconstruction process is portrayed as an optimization issue with the goal of considerably reducing anomalous traits from an anomaly [3]. Deep neural networks (DNNs) are increasingly being used in software systems.

DNNs have been demonstrated to have problems in the past. Due to a lack of understanding of model behaviour, conventional debugging tools do not assist localising DNN issues [4]. A rising number of mobile software applications are incorporating deep learning (DL). Mobile DL apps combine large-scale data DL models with DL programs in a variety of software apps [5]. A security mentality must be applied to all software engineering methods in order to produce secure software [6]. Statistical fault localization is a simple technique for identifying candidates for defective code places fast [7-8].

Researchers have conducted empirical investigations on these flaws in order to better understand deep learning program bugs. Conduct an empirical investigation in particular to comprehend Tensor Flow applications' bugs. A program that uses Tensor Flow's APIs is referred to as an application of Tensor Flow in this context. In today's manufacturing operations, monitoring rotating machinery is critical. Several machine learning and deep learning-based modules have demonstrated good defect identification and diagnosis outcomes [9-11]. Flaky tests have received a lot of attention in recent years, and with good reason. These tests are time and money consuming, and they reduce the reliability of the test scripts and build processes that they affect [12, 13]. The automatically generated crash reports can examine the root of the fault that caused the crash (also known as the crashing fault), which is an important aspect of software quality assurance [14-16].

In this method we proposed and implemented using subjective traditional procedures have been employed often in recent decades [17]. To ascertain the cause of the recall, i.e., the defect type, each incidence was studied using root cause analysis [18]. These diagnosticians frequently work without adequate user instructions or comprehensive topic knowledge. In literature and practice, software problem diagnosis as performed in the field is underrepresented [19, 20].

A contribution of this paper is

- In contrast to our previous work, which solely examined bug symptoms and causes, the extended version examined both bug fixes and bugs that occurred across deep learning method.
- We compared the symptoms, causes, and repair patterns we had found. We discover that TensorFlow contains type confusions depending on the comparison, which the earlier studies had not mentioned.
- Additionally, we discover that TensorFlow suffers dimension mismatches just like deep learning applications. Ten repair templates are found in Tensor Flow problems. We discover that correcting deep learning defects takes substantially the same repair steps as fixing bugs in other types of projects, despite the fact that fixing TensorFlow bugs requires specialized knowledge. Additionally shown is the relationship between typical mending patterns and their root causes.

3. Proposed method

3.1 Tensorflow implementation

The calculations and steps of a machine learning process are represented by dataflow graphs in TensorFlow. Each node in a dataflow graph represents a single mathematical operation (for

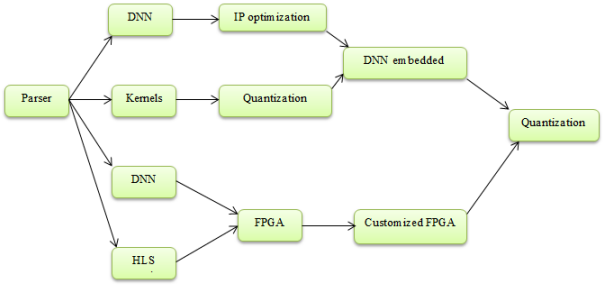

Figure. 1 TensorFlow implementation

example, matrix multiplication), and each edge indicates a data dependency, as shown in Eq. (1).

$$d = \sqrt{\sum_{i=1}^{n}(xi - yi)}\,2 \quad (1)$$

The data format of data exchanged between two nodes at each edge is defined by a tensor (n-dimensional arrays) show in Fig. 1.

3.2 The tensorflow bug remediation process

TensorFlow's source code is available on GitHub, where users can report bugs and track commits. When a user faces problems (such as a bug), she often submits an issue, which in this article is referred to as a bug report. This report's contents are used to figure out what's wrong. The bug report includes basic information like the operating system, the broken TensorFlow version, and code samples that can be used to reproduce the issue. Aside from that, the bug report includes a summary that details the problem. In addition, the reporter may be able to offer a practical bug fix. After getting an error, TensorFlow developers explain the possible causes and how to fix them. Developers can also refer to previous bug reports and pull requests when discussing a more complex bug. To label bug reports as "resolved" or "fixed," other open-source communities (e.g., Jira) employ more advanced issue trackers. On the other hand, GitHub's issue tracker is more basic, and its status is usually erroneous.

While labels on bug reports do not represent the state of an issue, A pull request marked "ready to pull" has been fixed and is ready to combine. The label makes it simple to spot fixed pull requests. Some pull requests include links to associated bug reports, making finding bugs much easier. Several pull requests, however, are missing issue reports and are not provided by users. We look for problem fixes in pull requests using keywords (for example, bug) (Section 3.1).

3.3 Dataset

TensorFlow was the topic of our investigation after it was discovered that TensorFlow APIs are used in over 36,000 GitHub projects. TensorFlow issues have affected thousands of apps as a result. To extract authorised pull requests, we follow the steps below:

Step 1: Labels are used to group pull requests. Begin with closed pull requests marked "ready to pull" to avoid cosmetic bugs. Because completed pull requests earlier to a specific date aren't recognized, As described in Step 2, By scanning keywords, we collect samples from previously closed pull requests.

Step 2: Using keywords to find pull requests. To track down people who fix problems in closed pull requests, they utilize phrases like "bug," "fix," and "error." We remove bug fixes that remedy surface issues by using terms like "typo" and "doc." We are actively examining the remaining bug fixes, carefully evaluating their pull requests in order to identify genuine changes.

Step 3: It is extracted bug reports and code changes. Using postings and commit, for each of our bug fixes, they receive an issue report as well as code updates. Symptoms, root causes (RQ1), and locations are identified using the data collected and the pull requests that go with it (RQ2).

3.4 Manual analysis

All bugs in our investigation will be manually inspected by two graduate students. Both students are computer science majors who have worked with deep learning techniques before. They've created at least two Tensor-Flow-based deep learning application projects in the last two years (for example, business data mining) The two students examine the bugs separately and compare their findings, following our technique. They discuss TensorFlow bugs at our weekly group meetings when they can't agree on a solution. Our first-agreement rate is 92.57 percent. By dividing the consistent examples by the total cases, the initial agreement rate is calculated.

3.5 RQ1's protocol

Using taxonomies from previous studies, they built their own taxonomy of bug symptoms and causes. If they identify a TensorFlow issue that meets these requirements, they expand their taxonomy by adding an existing category.

$$lift(A,B) = \frac{P(A \cap B)}{P(A) \cdot P(B)} \quad (2)$$

P(A), P(B), and P(A+B) are the probabilities that a bug belongs to category A, B, or both A and B. (C). (AB). If the lift value is larger than one, a symptom is associated with a root cause; otherwise, it is not.

3.6 Model and software reliability

Software reliability modeling is a mathematical approach in which the parameter estimates used have a direct impact on software reliability and fault prediction accuracy. In the G-O software dependability model, this study will estimate the properties of a representative model. The cumulative failure rate of the software system was calculated to be:

$$m(t) = a(1 - e_{bt}) \quad (3)$$

Where m (t) denotes the predicted function of the cumulative number of failures till time t, and a denotes the total rate of failure that the software expects to find when the test is completed. and b is a proportionality constant with a range of 0; and (0, 1).

3.7 Fitness function construction

Creating a suitable fitness function, or objective optimization function, and changing the parameterization problem into a function optimization problem is the key to using intelligent optimization algorithms to estimate the software reliability model. Based on the characteristics of the software reliability model and the least square approach, the fitness function generated is as follows:

$$fit = \frac{\sqrt{\sum_{i=1}^{n}[m(ti)m^0(ti)]^2}}{n} \quad (4)$$

where fit denotes the proportional difference between the observed and expected number of software failures. The lower the fit value, the more precise the model fitting and, as a result, the better the parameter estimate result. During the testing

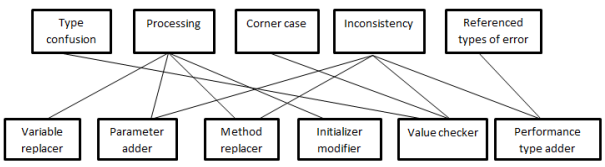

Figure. 2 The relationship between repair methods

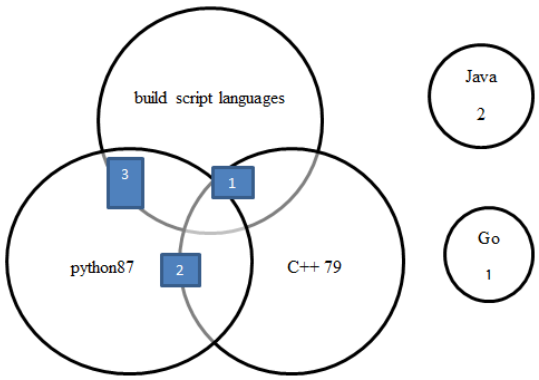

Figure. 3 The variety of programming languages that are available

period, a total of m (ti) failures were discovered (0,t_i). For the test period (0, ti), the model predicts a total of m0 failures (t_i). The i-th failure happens at time t_i.

3.8 Bug category correlation

Fig. 2 depicts the root causes and repair patterns. In this diagram, rectangles indicate fundamental causes. Mending patterns can be seen in the rounded rectangles. Repair patterns with less than three issues are not considered [21]. Correlations are represented by lines, and correlations with values larger than one are highlighted. We rule out isolated fixes since not all bug patches follow the same repair pattern. Fig. 2 depicts the following relationships:

In conclusion, our gathered fixes yield ten repair templates. they discover two novel templates when compared to previous studies, Although the majority of the templates we found overlap with those already in use.

The distribution is depicted in Fig. 3.

3.9 Scheme of MR categorization

The following criteria were checked for each MR in the chosen sample: 'phase identified,' 'defect kind,' "real defect position," 'defect trigger,' and 'barrier analysis details,' among others.

Information about phase detection when it comes to finding problems sooner, it's critical to know when the defect was discovered. The flaw can be discovered in any of the ten process phases see Table 1 that the analyst chooses. The analyst may also explain why the problem was not identified earlier.

Types of defects Implementation, interface, and external defects were split into three categories. There are a variety of fault kinds within each of these classes.

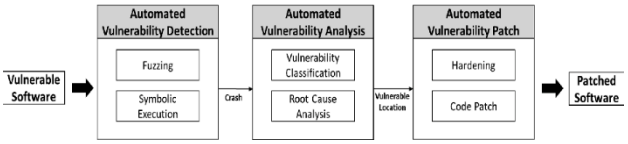

Figure. 4 Vulnerability analysis using automated root cause

3.10 Automated root cause analysis of vulnerabilities

To correctly fix the vulnerability, they must first determine the source of the problem and then apply the patch. To identify risky places, several investigations are being carried out [22]. Fault localisation, pattern-based evaluation, the three types of automated vulnerability root cause analysis approaches addressed in this section are and taint analysis show in Fig. 4.

3.10.1. Fault localization

Fault localization is a technique that uses test scenarios to determine the location of a vulnerability. To detect faults, four approaches can be used: similarity-based, statistics-based, artificial intelligence-based, and program analysis-based fault localization [23]. The frequency with which sentences are executed in both successful and unsuccessful tests is used in similarity-based fault localization.

Statistics-based fault localization calculates the chance of discovering an alternate path while applying conditional probability to select one. When a program is transformed into a behavior graph using graph mining tools, the AI-based defect localization methodology is a method of looking for a subgraph. The program-based fault localisation technique computes the edges in the control-flow graph connecting questionable code blocks to assess the suspicion of a code block.

$$suspiciousness_{(i)} = f\ ailed_{(i)} / \sqrt{\left(totalf\ ailed_{(i)}\right) \times \left(f\ ailed_{(i)}\right) + \left(passed_{(i)}\right)} \quad (5)$$

3.10.2. Similarity analysis using signatures

Similarity analysis based on signatures compares bug signatures to those of other software to see if the target software has the same signature [24]. Although there are numerous functions in source code, comparing code to binary has severe limitations, such involves code cloning based on functions, instructions, and code similarity analysis. The main issue is that each CPU architecture creates its own binary code.

$$S(X,Y) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{xi-Xn}{\emptyset x}\right)\left(\frac{yi-Yn}{\emptyset x}\right) \quad (6)$$

If X axis is (xi, xn) and Y axis (yi, yn)

$$\text{Where } \emptyset G = \sqrt{\sum_{i=1}^{n}\left(\frac{Gi-Gn}{n}\right)2} \quad (7)$$

A number of multi-platform research have been conducted to address this issue, depending on whether a weak code pattern is explored with an intermediary language or a weak code similarity inquiry is compared with an object, different analysis approaches are used. Because the executable file generated is unique, various studies have been conducted to develop a multi-platform environment for pattern similarity analysis.

4. Experimental result

4.1 Root causes

Table 1 summarizes the main findings from deep learning applications. Some of the issues they observed aren't found in deep learning libraries or are unusual. They discover problems like improper model parameters and structure inefficiency in TensorFlow applications, for example. Crashing accounts for 92% of TensorFlow difficulties, with performance bugs accounting for the remaining 8%. When we compare TensorFlow flaws to the distribution, we find that they are more diverse.

4.2 Identifying crucial root issues that can be improved or eliminated

The distributions of MRs by defect type were evaluated as a preparatory step in determining dominating contributions, with the result that type algorithm and type functionality flaws vastly outweighed all other defect kinds in Table 1. A close look into MRs with the defect types 'algorithm' and 'functionality.' The discovery phase has been related to the MRs for the defect types "algorithm" or "functionality" and the defect class "implementation":

The data demonstrates that more than 60% of faults are discovered late in the process, after systems development and system testing, for all defect classifications. The fact that the average is 60% suggests that the remainder 40% of all MRs are typically diagnosed early show in Fig. 5 and Table 2.

Table 1. They display the results for all examined MRs as well as subgroups of MRs classed as "algorithm" or "functionality" flaws

Defect type	SW planning (percent)	Integration of software (percent)	Integration of systems (percent)	System evaluation (percent)
All MRs	15	27	39	14
Algorithm	2	32	78	13
Functionality	17	25	49	18
All other MRs	35	22	36	8

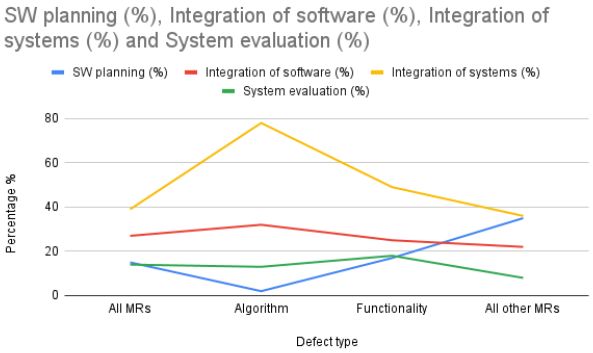

Figure. 5 The results are given for all MRs examined, as well as "algorithm" or "functionality" defects in subsets of MRs

Table 2. displays the percentage of components with faults of less than N

Defect type	Architecture	High-level planning	Design and specification of components (percent)	Implementation of components (percent)
All MRs	5	7	29	38
Algorithm	1	1	47	49
Functionality	3	12	23	45

Because the average is 60%, the remainder 40% of all MRs are usually detected early. The connections between numerous root causes were studied to this purpose. There is no unusual behavior associated with the phase when the problem was introduced. Algorithm flaws appear during the design, specification, and implementation phases, and they deviate greatly from the standard defect detection distribution. Functional defects arise at about the same rate as the average.

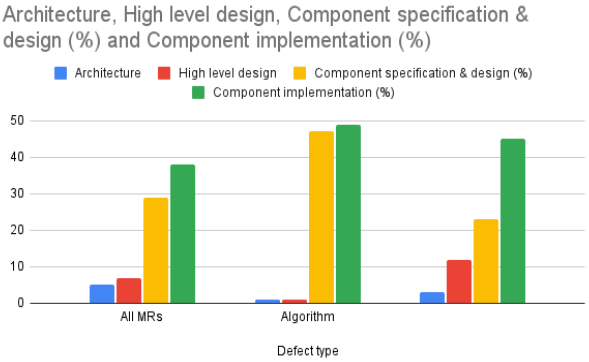

Figure. 6 Software process compliance metric

Table 3. The number of faulty versions developed for each amount of defects, the size of the testing procedure, and the number of test runs for each are all depicted in this diagram

	Gzip	Replace	Space	Total
1 faults	21	33	37	98
2 faults	179	179	179	539
3 faults	419	419	419	1257
4 faults	540	540	540	1620
5 faults	540	540	540	1620
6 faults	540	540	540	1620
7 faults	540	540	540	1620

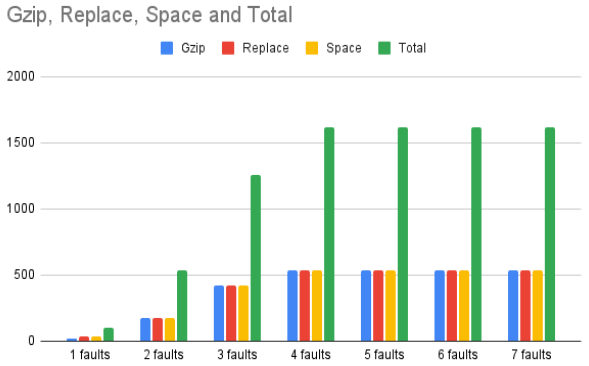

Figure. 7 For each quantity of mistakes, test suite size, and test executions, the number of incorrect versions created is shown

4.3 Interference with fault-location

The ability of a fault to obstruct the location of another problem. The presence of a defect causes the fault-localization approach to be inefficient in locating another problem, which we define as fault-localization interference. While small amounts of ineffectiveness may be overlooked by developers, Fault localization interference refers to any decrease in fault-localization efficacy induced by the presence of another defect. In the example in Fig. 6, if either bug1 or bug2 had been deleted, the other would have been better located (as indicated by the

results after eliminating the test case failures that they generated, respectively). As a result, each issue caused fault localization interference for the other in this scenario.

Failure clustering is a technique for troubleshooting numerous errors at the same time, while previous research implies that failure clustering can reduce interference, the addition of another automated technique adds to the computational and operational costs. Previous studies focused on the localizability of certain individual flaws; However, developers are frequently unaware of the number or nature of issues that lead to failures show in Fig. 7. In such cases, the ability to locate any problem could aid in an iterative debugging process that results in a fault-free program show in Table 3.

5. Conclusion

While empirical research has been carried out to properly appreciate deep learning concerns, it has generally focused on defects in its applications, leaving the source of errors within a deep library largely unexplained. They investigate 202 TensorFlow vulnerabilities in order to have a better understanding of them. Our findings suggest that (1) the causes are more essential than the symptoms; and the causes are more important than the symptoms. (2) There are several similarities between regular software defects and TensorFlow bugs; (3) Inconsistent defects are widespread in other supporting components in contrast; API implementations with inadequate data formatting (dimension and type) are prone to problems. They hope to explore flaws in other deep-learning libraries in the future in order to obtain a deeper understanding of deep-learning framework issues, as well as develop automated ways for detecting defects in deep-learning libraries. Furthermore, they investigate repair patterns by focusing on source files while ignoring configuration files. Read issue reports and their solutions to learn more about deep learning issues. The runtime behaviors of some bugs are not hidden and are difficult to discover via static analysis. In the future, they plan to use dynamic analysis to investigate the runtime behavior of deep learning difficulties. In our upcoming research, we want to look at how many software modules actually have defects by including more projects that were created using various programming languages as well as industry-sponsored projects. Additionally, rather than estimating it from the module level, we are more interested in estimating the amount of flaws.

Conflicts of Interest

Authors do not have any conflicts.

Author Contributions

The Paper conceptualization, methodology, software, validation, formal analysis, investigation, resources, data curation, writing—original draft preparation, writing—review and editing have been done by 1st author. The visualization, supervision have been done by 2nd author. The project administration has been done by 3rd author.

Acknowledgments

The Author with a deep sense of gratitude would thank the supervisor for his guidance and constant support rendered during this research.

References

[1] A. Makhshari and A. Mesbah, “IoT Bugs and Development Challenges”, In: *Proc. of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 460-472, 2021.

[2] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, “The Symptoms, Causes, And Repairs of Bugs Inside a Deep Learning Library”, *Journal of Systems and Software*, Vol. 177, p. 110935, 2021.

[3] C. M. Roelofs, M. A. Lutz, S. Faulstich, and S. Vogt, “Autoencoder-Based Anomaly Root Cause Analysis for Wind Turbines”, *Energy and AI*, Vol. 4, p. 100065, 2021.

[4] M. Wardat, W. Le, and H. Rajan, “DeepLocalize: Fault Localization for Deep Neural Networks”, In: *Proc. of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 251-262, 2021.

[5] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, “An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications”, In: *Proc. of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 674-685, 2021.

[6] S. O. Slim, M. M. Elfattah, A. Atia, and M. S. M. Mostafa, “IoT System based on Parameter Optimization of Deep Learning using Genetic Algorithm”, *International Journal of Intelligent Engineering and Systems*, Vol. 14, No. 2, pp. 220-235, 2021, doi: 10.22266/ijies2021.0430.20.

[7] R. R. Althar and D. Samanta, “The Realist Approach for Evaluation of Computational Intelligence in Software Engineering”,

Innovations in Systems and Software Engineering, Vol. 17, No. 1, pp. 17-27, 2021.

[8] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating Faults with Program Slicing: An Empirical Analysis", *Empirical Software Engineering*, Vol. 26, No. 3, pp. 1-45, 2021.

[9] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An Empirical Analysis of Ui-Based Flaky Tests", In: *Proc. of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 1585-1597, 2021.

[10] Z. Xu, T. Zhang, J. Keung, M. Yan, X. Luo, X. Zhang, and Y. Tang, "Feature Selection and Embedding Based Cross Project Framework for Identifying Crashing Fault Residence", *Information and Software Technology*, Vol. 131, p. 106452, 2021.

[11] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs", In: *Proc. of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 187-199, 2021.

[12] X. Lei, P. Sandborn, R. Bakhshi, A. K. Pour, and N. Goudarzi, "PHM Based Predictive Maintenance Optimization for Offshore Wind Farms", In: *Proc. of the 2015 IEEE Conference on Prognostics and Health Management*, pp. 1-8, 2015.

[13] A. Stetco, F. Dinmohammadi, X. Zhao, V. Robu, D. Flynn, M. Barnes, and G. Nenadic, "Machine Learning Methods for Wind Turbine Condition Monitoring: A Review", *Renewable Energy*, Vol. 133, pp. 620-635, 2019.

[14] G. Helbing and M. Ritter, "Deep Learning for Fault Detection in Wind Turbines", *Renewable and Sustainable Energy Reviews*, Vol. 98, pp. 189-198, 2018.

[15] R. Chalapathy and S. Chawla, "Deep Learning for Anomaly Detection: A Survey", *arXiv Preprint arXiv:1901.03407*, 2019.

[16] M. Jeevanantham and J. Jones, "Extension of Deep Learning Based Feature Envy Detection for Misplaced Fields and Methods", *International Journal of Intelligent Engineering and Systems*, Vol. 15, No. 1, pp. 563-574, 2022, doi: 10.22266/ijies2022.0228.51.

[17] M. A. Lutz, S. Vogt, V. Berkhout, S. Faulstich, S. Dienst, U. Steinmetz, and A. Ortega, "Evaluation of Anomaly Detection of an Autoencoder Based on Maintenace Information and SCADA-Data", *Energies*, Vol. 13, No. 5, p. 1063, 2020.

[18] N. Renström, P. Bangalore, and E. Highcock, "System-Wide Anomaly Detection in Wind Turbines using Deep Autoencoders", *Renewable Energy*, Vol. 157, pp. 647-659, 2020.

[19] M. Aurisicchio, R. Bracewell, and B. L. Hooey, "Rationale Mapping and Functional Modelling Enhanced Root Cause Analysis", *Safety Science*, Vol. 85, pp. 241-257, 2016.

[20] D. Ballabio, F. Grisoni, and R. Todeschini, "Multivariate Comparison of Classification Performance Measures", *Chemometrics and Intelligent Laboratory Systems*, Vol. 174, pp. 33-44, 2018.

[21] B. Boehmke and B. Greenwell, *Hands-on Machine Learning with R*, 1st Ed., *Chapman and Hall/CRC*, 2019.

[22] B. Brown, M. Chui, and J. Manyika, "Are You Ready for the Era of Big Data", *McKinsey Quarterly*, Vol. 4, No. 1, pp. 24-35, 2011.

[23] Y. H. Chang, C. H. Yeh, and Y. W. Chang, "A New Method Selection Approach for Fuzzy Group Multicriteria Decision Making", *Applied Soft Computing*, Vol. 13, No. 4, pp. 2179-2187, 2013.

[24] H. Suryotrisongko and Y. Musashi, "Hybrid Quantum Deep Learning and Variational Quantum Classifier-Based Model for Botnet DGA Attack Detection", *International Journal of Intelligent Engineering and Systems*, Vol. 15, No. 3, pp. 215-224, 2022, doi: 10.22266/ijies2022.0630.18.