On the Stability and Applicability of Deep Learning in Fault Localization

Viktor Csuvik*, Roland Aszmann*, Árpád Beszédes*, Ferenc Horváth*, Tibor Gyimóthy*
*Department of Software Engineering
University of Szeged, Szeged, Hungary

{csuvikv,aszmann,beszedes,hferenc,gyimi}@inf.u-szeged.hu

Abstract—**Numerous Deep Learning (DL)-based fault localization (FL) methods are developed with the aim of leveraging the code coverage matrix and failure vector to identify the connection between program elements and defects. The imbalanced data on which these approaches train their models poses a substantial challenge to the effectiveness of fault localization techniques. This study explores the stability of fault localization models in deep learning, specifically, their performance when trained repeatedly using the same input but varying random initializations. Using the Defect4J benchmark, we trained deep learning models (MLP, CNN, and RNN) independently and found that 86 cases resulted in (partly) consistent rankings among all five models and versions, while 621 exhibited varying outcomes, meaning that 90% of the produced ranks were different in subsequent trainings. The models showed significant variability in ranking results, with maximum ranks sometimes five times that of the minimum. We also adapted the churn metric from DL research to evaluate models, confirming their instability. To improve stability, meta-parameter optimization, model simplification and resampling has been applied. Although some of these techniques proved effective, even with the improvements, the models remained insufficiently stable to produce reliable results.**

Index Terms—**Spectrum Based Fault Localization, Deep Learning, Prediction Churn, SBFL, DL**

I. Introduction

Efficient program fault localization techniques (FL) play a crucial role in software development due to the time-consuming and resource-intensive nature of identifying and fixing defects in large, complex programs. Sophisticated methods are designed to streamline this process by analyzing code coverage and failure data to pinpoint the potential sources of errors. Spectrum-Based Fault Localization (SBFL) approaches are among the most widely adopted ones [1].

In recent times, there have been results suggesting Deep Learning (DL) based solutions, from which coverage matrix-based approaches play a prominent role. Several noteworthy outcomes in this direction have been achieved by, for instance, authors of papers [2]–[9]. These researchers utilize deep learning models, expecting them to reflect the complex nonlinear relationship between the statements and test results. The suspiciousness of each statement is obtained by training a deep learning model on coverage information and then by constructing a special input that makes the model to estimate the faultiness of a single statement. This approach is often referred as DLFL (Deep Learning Fault Localization), and to the best of our knowledge it was introduced by Wong *et al.* in 2009 [10]. Since then several other research groups have investigated DLFL, achieving moderate success [8], [11], [12].

In DL approaches, stability is essential because only reporting a good DL performance may threaten the experimental conclusions [13]. The basic requirement of a stable model is that after re-training it, the obtained results are similar (ideally nearly the same) as before using the original dataset. In our study, we investigated the extent to which DLFL models fulfill the stability criterion on the Defect4J benchmark. When training what we call a *stable model* multiple times on the same input, we expect that despite different initialization values, the resulting models provide similar or identical results for the test cases. By examining publicly available DLFL models, we found that the result rankings exhibited significant variation between independent model trainings. To asses the stability of these models, we adopted *churn*, a metric commonly used in the ML literature [14], [15]. It measures the probability that the output of two independently trained models will be different. If this probability is high the models are likely to produce different outputs, suggesting an error in model design or implementation.

In our research, we used 230 versions from the Defecs4J benchmark and applied DLFL by training each deep learning model (Multi-Layer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN)) five times independently, using different random initializations. Surprisingly, we found that in the case of the examined versions, only in 86 instances did all five models produce partly identical results, whereas for 621 cases, all five models yielded different results. Notably, the result rankings exhibited significant variation, with a substantial spread in some cases, where the maximum rank value was five times as large as the minimum value. On average, 90% of the produced ranks were different in subsequent trainings. In terms of the churn results, we found that, by the original definition the metric, it showed a value of 0.95 on average (the probability that ranks between two trainings differ is 95%). Our churn metric defined for the FL problem showed a more nuanced picture, but still high probabilities can be observed.

We attempted to enhance stability by making parameter adjustments, but significant improvements proved elusive. However, through more substantial modifications to the model, such as reducing the number of layers, we achieved some improvement in stability. We also explored whether improving the coverage matrix through resampling could enhance stability. It was found that there was a significant improvement, but the model still remained insufficiently stable to provide reliable and reproducible results. Combining these two methods led to better stability results but still not acceptable.

In addition to the instability issues, the performance of deep learning models barely approach that of standard methods. Not mentioning the resources needed: while deep learning models are very expensive to teach and evaluate, standard approaches can be run quickly, calling into question the validity of the former. The following points summarize our main contributions:

- **Model evaluation using different metrics**: to test the stability of DLFL models, we used several metrics: mean, standard deviation and churn also showed that the results of independent trainings are very different from each other. By conducting a large-scale training we investigate in detail the stability of DL-based FL method.
- **Proposal of potential improvements**: based on the DL literature, we applied metaparameter optimization, model simplification and oversampling. Although metaparameter optimization is more demanding on the resource, this technique did not improve the stability of the models. On the other hand, with the combination of model simplification and oversampling we achieved moderate success: stability improved, but still remained unreliable.

In summary, our experiments show that DLFL approaches are not stable, and furthermore, this may imply that similar issues may arise in other DL application domains in software engineering. Our approach to measure model stability could be employed in many such other areas as well.

II. Research Objectives

We organize our experiment along the following:

RQ1: *How stable is the training process of DLFL approaches and how does it affect reproducibility?*

By conducting a large-scale training we investigate in detail the stability of DL-based FL method. If the outputs of successive model trainings are very different from each other, it makes impossible to reproduce previous research findings, and makes these approaches unusable in practice (since we cannot provide reliable rankings).

RQ2: *Can the stability and applicability be increased?*

Stability is a prerequisite for reproducibility – using standard techniques from DL literature (metaparameter optimization, model simplification and resampling) we attempted to address the stability problem of the models.

III. Related Work

A. Related Work on Deep Learning Fault Localization

In the realm of DLFL, coverage matrix-based approaches have emerged as pivotal contributions. Zhang *et al.* in their seminal work [2] utilized three deep learning architectures (MLP, CNN, RNN) for fault localization. Subsequent research expanded on this foundation, employing oversampling [3] and test generation [4] to address class imbalance. The results demonstrated that these approaches enhanced fault localization effectiveness, surpassing previous DLFL approaches. The group later proposed Aeneas, synthesizing failing test cases from a reduced feature space [5]. This approach statistically outperformed baselines. To address class imbalance further, subsequent works introduced cost-effective data augmentation approaches, such as between-class learning [6] and the Lamont approach [7]. Additionally, the use of Generative Adversarial Networks (GANs) in CGAN4FL [8] demonstrated their efficacy in constructing a class-balanced dataset for fault localization. While these publications share a common theme, each introduces a unique improvement in fault localization. Unfortunately, not all listed papers have online appendices or repositories, but only one [5] provides source code. Also, the lack of predefined seeds in the training process poses challenges to the reproducibility of their experiments.

The issue of data imbalance in intelligent fault diagnosis methods has garnered extensive attention, leading to numerous publications [16]–[18]. For instance, Fang et al. [19] employ a conditional variational autoencoder (CVAE) for synthesis and apply fault localization techniques. GNet4FL [20] combines static and dynamic features for more precise fault localization. Other techniques, such as DeepFL [21] and FLUCCS [22], leverage additional information, with DeepFL automatically learning latent features and FLUCCS using Genetic Programming and linear rank Support Vector Machines (SVMs) for learning fault localization formulae.

B. Related Work on the Stability of Deep Learning

Conventional training methods for neural networks incorporate various sources of randomness, such as initialization, mini-batch order, and data augmentation. Since neural networks tend to be significantly over-parameterized in practical applications, this inherent randomness can lead to issues [23]. Situations when two models independently trained by the same algorithm produce differing predictions for the same input are referred to by literature as *churn* [14], [15]. Churn represents the proportion of test samples where predictions of the models do not match. One approach to mitigate churn involves eradicating all forms of randomness within the training configuration. However, even if one manages to control the seed used for random initialization and the data ordering, which itself presents challenges, it remains difficult to evade the inherent non-determinism present in contemporary computing platforms [24]. Distillation [25] transfers knowledge from larger to smaller neural networks to reduce churn, while this paper does not explore other properties of neural networks [26]. Having stable models capable of generating predictions unaffected by the random training factors is essential for developers to trust deep learning approaches – a challange modern machine learning still have to tackle with [27].

Liu *et al.* [13] examines the reproducibility of DL models in the field of SE. The study reveals that only 10.2% of the reviewed publications address replicability or reproducibility, with over 62.6% not sharing high-quality source code or complete data. Experimental results underscore the importance of reproducibility and replicability, demonstrating challenges in reproducing DL model performance due to an unstable optimization process, non-convergence in model training, and sensitivity to vocabulary and testing data size. While sharing a reproduction package in deep learning supports reproducibility, the inherent randomness in model initialization and optimization makes it challenging to guarantee that models trained by different researchers will produce identical experimental results even when re-running the provided source code and data. Thus stable models are fundamental to have reliable and applicable DL models in any domain.

In our work, we did not concentrate on general DL models, but observed a special field: fault localization. We did not seek to assess the actual performance of such models, we only test how stable and reproducible these methods are. Our methodology differs from what we saw in previous literature: in addition to the classical statistical measures, we also used the churn metric to support our findings. Furthermore all of our experiment data is available in the online appendix [28], [29] with fixed seed values and reproducible measurements.

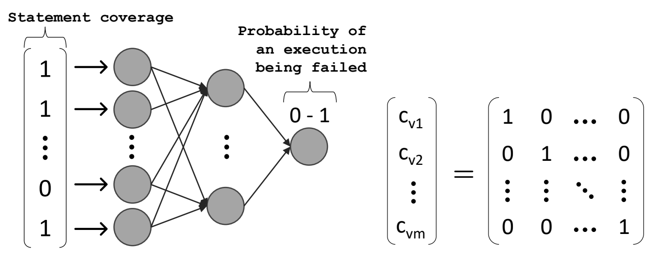
IV. Background

A. Coverage Matrix-Based DL Model

To the best of our knowledge, DL-enhanced coverage matrix-based fault localization was introduced in 2009 by Wong *et al.* [10]. In this section, we introduce their proposed method in short. Suppose we have a program with *m* executable statements and exactly one fault. Suppose also that there are *n* executable test cases of which *k* tests are successful and *n - k* are failed. This data is then organized into an *n x m* sized matrix, where inside the matrix there is 1 if the test covers the statement and 0 otherwise. An example of such a matrix can be seen in Figure 1. The result of the test execution is also organized into a vector form that has *n* rows, denoting whether the test was successful (0) or failed (1).

$$\overbrace{\begin{pmatrix}0&0&0&0&0&1&1&0&0\\1&1&1&0&1&0&1&1&1\\1&1&1&0&1&1&1&1&1\\0&0&1&0&0&1&1&1&0\\1&1&1&1&0&1&0&1&1\end{pmatrix}}^{\texttt{Statement coverage}} \longrightarrow \begin{array}{l}\left.\begin{pmatrix}0\\0\\0\end{pmatrix}\right\}\texttt{3 successful tests}\\ \left.\begin{pmatrix}1\\1\end{pmatrix}\right\}\texttt{2 failed tests}\end{array}$$

Fig. 1: Coverage matrix with 9 statements and 5 test cases.



(a) High-level illustration of the neural network used for fault localization.

(b) Virtual test simulating coverage of a single statement.

Fig. 2: Components of DL-enhanced SBFL.

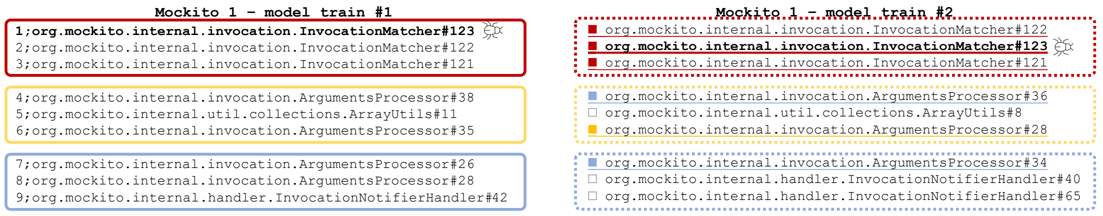
Next, a neural network is being constructed. The input layer has *m* neurons, as a single row from the above matrix (*i.e.* 'statement coverage vector') forms the input of the model. The output is a single neuron, and the desired outcome is to predict whether the test execution result was successful or not. The inner structure of the neural network is arbitrary, in the literature one can find diverse architectures, including Feedforward Neural Network, Convolutional Neural Networks, Recurrent Neural Networks and even Graph Neural Networks [12], [20], [30]. The outcome can be interpreted as an estimation of the test execution result. Note that there is no train-test-validation split of the training data, as the training objective is to learn to predict whether a test was successful or not, based only on the test coverage matrix as we depicted this process on Figure 2 (a). Thus the training objective is to perfectly learn when will a test case fail and when will succeed, no generalization needed.

The actual fault localization part comes after the neural network has been trained. Assume there is a set of so called *virtual test cases* whose coverage vectors are $c_{v1}, ..., c_{vm}$. The execution of a virtual test case covers only one statement and if we organize these into a matrix form again, the result is a diagonal matrix as shown in Figure 2 (b). The execution of such a virtual test case is interpreted as a *test that only covers one statement*. If a statement is contained in a lot of failed test executions, the output of such a virtual test case is expected to be high. This implies that during the fault localization, we should first examine the statements whose output values are high. The output value of the neural network is between 0 and 1, the larger the value is the more likely it is that the corresponding statement (in which in the coverage vector the value was 1) contains a bug. This output can be treated as the suspiciousness of a given statement in terms of its likelihood of containing the bug.

The fault localization process continues from this step the usual way: the statements are ranked based on their suspiciousness – more suspicious statements are sorted at the top, while less suspicious ones to the bottom of the list.

B. Churn Measurement

We interpret churn as defined by Cormier *et al.* as the expected disagreement between the predictions of two models [14]. Churn is zero if both models provide the same output for the same input, while large churn values mean



```
Mockito 1 - model train #1
1;org.mockito.internal.invocation.InvocationMatcher#123
2;org.mockito.internal.invocation.InvocationMatcher#122
3;org.mockito.internal.invocation.InvocationMatcher#121

4;org.mockito.internal.invocation.ArgumentsProcessor#38
5;org.mockito.internal.util.collections.ArrayUtils#11
6;org.mockito.internal.invocation.ArgumentsProcessor#35

7;org.mockito.internal.invocation.ArgumentsProcessor#26
8;org.mockito.internal.invocation.ArgumentsProcessor#28
9;org.mockito.internal.handler.InvocationNotifierHandler#42
```

```
Mockito 1 - model train #2
■ org.mockito.internal.invocation.InvocationMatcher#122
■ org.mockito.internal.invocation.InvocationMatcher#123
■ org.mockito.internal.invocation.InvocationMatcher#121

■ org.mockito.internal.invocation.ArgumentsProcessor#36
□ org.mockito.internal.util.collections.ArrayUtils#8
■ org.mockito.internal.invocation.ArgumentsProcessor#28

■ org.mockito.internal.invocation.ArgumentsProcessor#34
□ org.mockito.internal.handler.InvocationNotifierHandler#40
□ org.mockito.internal.handler.InvocationNotifierHandler#65
```

Fig. 3: Statement boxing including 3 statements each. Boxes are highlighted with colors, the faulty statement is bold and the list is ordered by the suspiciousness assigned by each model. The churn is calculated as follows: $Churn = 1 - 1/12 = 0.917$, $BoxChurn = 1 - 5/12 = 0.583$, while $FlBoxChurn = 1 - 1/1 = 0$.

that models independently trained disagree on most of the test data – suggesting that an error could have occurred in model design, or in learning and evaluation phase. This probability is essentially implemented in practice as:

$$1 - \frac{\#SamplesOnModelsAgreed}{\#AllSamples} \tag{1}$$

where $\#SamplesOnModelsAgreed$ is the number of samples on which the two independently trained models agree, while $\#AllSamples$ is the number of samples in the dataset. In Figure 3 one can observe that most of the assigned ranks differ between independent trainings.

C. *Churn Adaptation*

Churn is defined for classification tasks, for example image recognition [15]. The straightforward adaptation of this metric for fault localization would be to consider every rank as a class label and models need to produce the exact same list of suspicious statements. However, two key observations can be made: (1) statements on similar positions are counted as disagreement and (2) not all ranks are of equal importance - buggy statements are more important. For example, consider the case of 100 statements on which the two models rank the faulty statement on rank 1, while the remainder in random order. It is clear that in this case the worst case scenario is that the models only agree on the rank of the faulty statement, resulting in a high churn of 0.99 (although they ranked the faulty statement as intended). On the other hand, let us consider two models and define the ranks of the second model as the rank assigned by the first model + 1. Now the two models disagree on every sample, however the produced lists are very similar. To overcome these limitation we define two revised versions of churn for fault localization.

1) Statement Boxing: First we introduced what we call *boxing*, the process that groups statements into fixed sized boxes and then churn is measured not on the actual ranks, but on the assigned boxes. For example, in Figure 3 we defined a box size of 3. It can be observed that although the two independent trainings placed the faulty statement on different ranks, they are mapped in the same box, thus reducing churn value. In our experiments we used a predefined box size, but it can be considered as an input parameter. It is expected that $BoxChurn$ values are lower compared to regular $Churn$, as fewer classes are created in this new setting. The implementation of the boxing function is arbitrary, in our experiments we used fixed box sizes of 5, 10 and 50.

2) Faulty Box: As the rank of the faulty statement is the most important in fault localization applications, we decided to adapt churn to meet this criteria. In the previous metrics disagreement between correct statements is also measured. To overcome this limitation we define $FlBoxChurn$ as $BoxChurn$ calculated only on the box containing faulty statements. Note that there might be more faulty statements in a system, thus the value of $FlBoxChurn$ is not necessarily 0 or 1. In Figure 3 this amendment results in a churn value of 0, which gives a better indication of the problem.

D. *SFL measures*

SBFL's effectiveness can be measured in various ways [31], [32], but most rely on estimating the effort programmers need to identify the faulty element using the tool. The rank list serves as a proxy for this property, with the number of elements before the first faulty element, often collectively called the *Expense*.

1) Expense metric: Most often, the absolute version of the Expense metric is used which means that we simply count the number of code elements in the rank list in front of the faulty one. One complication with this method are rank ties [33], i.e. situations when different code elements share the same suspiciousness scores. Typically, all elements in a rank tie are assigned the same rank value, based on one of these approaches [34]: *minimum*, which refers to the top most position of the elements sharing the same suspiciousness value (optimistic or the best case); *maximum*, where the bottom most position is used (pessimistic or the worst case); and the *average* strategy, where the medium position of the elements sharing the same suspiciousness value is used (average case).

Equation 2 shows the *absolute average rank* calculation [35], [36], where i and f are code elements, the latter being the faulty one, while s_i and s_f are the respective suspiciousness score values.

$$E(f) = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2} \tag{2}$$

Another issue arises when a program has multiple faults, which is common. Typically, the E value linked to the element with the highest suspiciousness score is used ($min(E(f))$, where $f \in$ {faulty elements}). We will use this as the expense measurement in the following.

2) Top-N: Several studies report that developers investigate only the first 5 or 10 elements in the ranking list by fault localization algorithms before giving up [37], [38]. The family of metrics that distinguishes bugs where the minimum rank of faulty elements is less than or equal to N is commonly referred to as *Top-N* or *acc@N* [39]. This metric represents the number of successfully localized bugs within the top-n elements of the ranking lists. Higher values are better for this metric, and the typical values used for N are 1, 3, 5, and 10. Code elements ranked behind N, are referred to as the *Other* category.

V. Experiment Setup

In preparation for our research, we found the seminal work of Xie *et al.* [5] whose implementation is publicly available [1] and on which our implementation was based. To the best of our knowledge, the research group published several other papers based on this code, with some improvements [2]–[4], [6]–[9]. They used three deep learning architectures: Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). All of these models are trained on the same dataset, computing architecture and with same metaparameters (200 epochs, 0.01 learning rate, batch size of 40 (except CNN – 10), MSE loss function and SGD optimizer with momentum of 0.9).

Our experiment was conducted on a 64-bit Linux server with Ubuntu 20.04.6 LTS. operating system, 10 Intel(R) Xeon CPUs, 128G RAM and one 12G GPU of NVIDIA GeForce RTX 2080 Ti. We used the 2.0.1 version of PyTorch library to build and examine the neural networks. To make our results reproducible, we fixed the random seed for each run of the different neural network models on the input matrices, by using the torch *manual_seed* and the cuda subpackage's same named function. The seed values are published with our source code at [28], [29].

A. Investigated Models

1) MLP: it consists of multiple layers of interconnected nodes, each of which performs a weighted sum of its inputs and applies an activation function to produce an output. MLP is a special case of Feedforward Neural Network (FNN) where every layer is a fully connected layer, and the number of nodes in each layer is the same. The architecture is built dinamically: the number of units in the hidden layers depends on the number of statements present in the program ($hidden\_units = max(30, \lfloor num\_statements/30 \rfloor) * 10$). Three hidden layers are present in the architecture, between which dropout of 25% has been applied to stabilize the learning process. The output layer is a sigmoid layer, which provides the network's output.

2) CNN: it is a specialized deep learning architecture designed for processing and analyzing grid-like data, such as images and matrices. It processes data through a series of convolutional layers, which apply filters to the input matrix to extract features. These features are then passed through one or more fully connected layers, which perform classification or regression tasks. In the observed repository two convolutional layers are present, the first with 15, while the second with 30 output layers. A kernel of size 1×3 is used, with a step size of 2, and max-pooling, resulting in a vector-like convolution. The explanation behind this is that while the arrangement of statements might be useful information (order of the columns), the order of the tests is certainly not (rows in the matrix). After the convolution, two fully connected layers are defined, between which dropout of 25% has been applied. The output again is produced by a single neuron using the sigmoid function.

3) RNN: it incorporates feedback loops, allowing it to maintain a form of memory about previous inputs. RNNs consist of a series of interconnected nodes, each of which takes an input and produces an output. The output is then fed back into the network as input for the next node in the sequence. The observed RNN architecture consists of 2 hidden layers with the same number of neurons as the number of statements and a single classification layer that applies the sigmoid function as before.

TABLE I: Main Properties of Programs Used from Defects4J (KLOC, Tests and No. Bugs Columns Data from [40])

Project	**KLOC**	**Tests**	**No. bugs**	**Avg. no. statements**	**No. suitable bugs**
Chart	96	2 205	26	1 374	21
Lang	22	2 245	65	804	65
Math	85	3 602	106	1 967	100
Mockito	20	1 379	38	1 778	37
Time	28	4 130	27	3 811	7
Total	**251**	**13 561**	**262**	**1 610**	**230**

B. Dataset

For the evaluation, just like in [5], Defects4J (v1.4.0)[2] was selected. In FL research [40], Defects4J is a well-known, widely used collection of Java programs and curated bugs. We evaluated all the models and their different improved versions (described in Section V-A and VII respectively) five times on the benchmark. Since our experiments relied on repeated executions, which is a highly hardware intensive and time consuming task, and because we aimed at comparing the results for all investigated models, we had to exclude certain parts of the benchmark for the experiments. We excluded the Closure program altogether and some versions from the other programs. As a result, a total of 230 defects were included in the final dataset whose properties are shown in Table I. Columns 2-4 show program sizes, the number of tests and available bugs. Column 5 contains the average statement count, while in column 6 are the number of used bugs (the exact list of bugs can be found in the supplemental material).

[1]https://github.com/ICSE2022FL/ICSE2022FLCode/

[2]https://github.com/rjust/defects4j/tree/v1.4.0

TABLE II: Number of same *Expense* appearing in the five separate runs of models

	All same	At least 4	At least 3	At least 2	All different
MLP	0	0	7	29	201
CNN	0	0	1	13	217
RNN	1	2	6	27	203

VI. Results

A. Statistical Measure-based Evaluation

To examine the stability of the models, we ran the various models on every selected program-versions 5 times in a row. The only difference between each 5 runs on the same input and model is the selected random seed, fixed at start. The separate runs have to produce same, or similar outputs, if the models are stable.

Table II illustrates the extent of variability observed in expenses across the 5 runs. Notably, only one model and program version consistently yielded the same expense in each run. Column 5 reveals that merely 12%–6%–12% of versions resulted in the same expense at least twice, underscoring the significant diversity in outcomes. Approximately 90% of versions exhibited different results for each unique random seed, emphasizing the substantial impact of seed selection on model outcomes.

Table III shows a summary of results by programs and total. We can see in the first 3 columns the mean expense of the models on the benchmark, by programs. Columns 4–6 show the average of standard deviation, presenting the disparity from mean values of the five separate run by each version. These are quite high values, showing the high variety of the outputs of the separate runs, and it strengthens our conclusions of previous table. From columns 7–12 are the average maximum and minimum expense values of the five runs of models. As we can see there are really high differences between them, in most of the cases the maximal expense is 4–5 times bigger than the minimal one. The RNN model on Chart has proportionally the smallest distinctness, however that has still a double difference quotient. We also used statistical significance testing, by using Wilcoxon sign-rank test [41], complemented with Cliff's Delta effect size measure [42], which proved that maximum values are significantly larger than minimals, with large to medium magnitude of effect size in all cases. Detailed statistical test results appear in the online appendix [28], [29].

TABLE IV: DLFL *Top-N* choosing Min or Mean or Max rank of the 5 seperate runs of each version by different models

		Top-1	*Top-3*	*Top-5*	*Top-10*	*Other*
MIN	MLP	23	49	58	78	152
	CNN	2	4	8	17	213
	RNN	8	23	29	42	188
MEAN	MLP	0	7	13	21	209
	CNN	0	0	0	1	229
	RNN	1	2	10	19	211
MAX	MLP	0	3	7	14	216
	CNN	0	0	0	1	229
	RNN	1	1	5	12	218

We examined the *Top-N* of the DLFL as well, and the outcomes are shown in Table IV. The rows present the selection of model results from the 5 runs, as minimal, mean and maximal expense. The minimal selection makes the *Top-N* results significantly better than the other two. Notice that the Top-1 case is almost zero if not selecting the best case, while Top-3-5-10 are at least 2 times, even 7 times worse than the minimal, mean and maximal selection. There is not so high difference comparing mean and maximal, which is due to the really high maximal values that have high effect on means. Our results presented how much effect the selection of random seed has on DLFL, which makes the applicability of this technique questionable. There are almost no cases where the difference is negligible between the five runs in a version. Detailed box plots depicting standard deviation of projects can be found in the online appendix [28], [29].

B. Churn-based Evaluation

As described in Sections IV-B and IV-C, we measured churn in three settings: (1) by the original definition, (2) by aligning statements into boxes and (3) by filtering out boxes that do not contain faulty statements. In Figure 4, we can observe histograms of the measured churn values. These values were measured using the 5 trained MLP models, while other models show similar trends. Churn is measured on model variant-pairs, for example on version 1 between training attempt 1 and 2, next between training attempt 1 and 3, etc. These pairwise churn values are then averaged so that the histogram can more

TABLE III: DLFL Average *Expense* results of 5 seperate runs of each version by different models

	Mean			**Standard deviation**			**Minimum**			**Maximum**		
	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN
Chart	419.54	460.50	521.47	312.28	225.47	232.54	124.81	223.69	293.00	853.17	760.52	830.00
Lang	201.89	329.80	306.34	122.63	179.35	139.67	82.19	134.65	162.25	385.27	569.00	493.98
Math	511.06	899.46	896.25	283.61	480.60	283.29	199.81	321.86	565.00	866.07	1454.44	1254.11
Mockito	522.48	707.43	679.70	310.54	402.02	295.12	184.43	259.93	315.62	906.22	1219.76	1038.05
Time	1618.37	1681.39	1422.41	571.25	941.65	506.16	871.86	638.21	912.07	2263.71	2857.71	2153.00
Average	**450.87**	**691.29**	**676.50**	**253.82**	**373.56**	**246.75**	**177.70**	**259.65**	**396.79**	**778.01**	**1145.81**	**993.17**

concisely represent all the measured data. It is clear from the figure that $Churn$ values are highest, $FlBoxChurn$ values the lowest and $BoxChurn$ somewhere between. Note that the size of boxes plays a crucial role here: larger box sizes are more forgiving to model errors. In our experiments we used fixed box sizes of 5, 10 and 50 (in Figure 4 we only included 5 and 10). More detailed histograms can be found in the online appendix of this paper [28], [29]. It is clear that these $Churn$ values are considerably large, even if the measure is adapted to fault localization.

It is application dependent what is an acceptable churn, but having an average value of 0.99 means that if the model is being trained again using different seeds, the statements will receive a different rank than before with 99% probability, which is clearly disadvantageous. One can argue that only the rank of the faulty statement is important, and that minor differences between ranks are still acceptable. To this end, we defined $FlBoxChurn$ and by measuring it on the subject programs we still see considerably high values. Notably, using a box size of 5, 65.08% of statements have a churn value higher than 0.5. For box size of 10 and 50, this is somewhat reduced to 64.94% and 60.94%, respectively. Average $BoxChurn$ value of box size 10 is 0.96, while for $FlBoxChurn$ it is 0.41.

Answer to RQ1: *In the 5 independent trainings we performed, we found that the output of the same model varies greatly due to the effect of random factors during training phase. Standard deviation of ranks is 291.37, with 606.22 mean values on average, implying 48.06% relative standard deviation, while based on boxed churn measurements the probability that two statements are going to end up in different boxes is 99.89%. These high values are good indicators of system instability.*

VII. Potential improvements

A. Metaparameter optimization

In our study, we diligently applied metaparameter optimization techniques to fine-tune the three observed models, with the hope of achieving significant performance enhancements. However, despite exhaustive experimentation and resource allocation, the outcomes proved disappointing. We used grid search to optimize the following parameters: learning rate, epochs, batch size, loss function and optimizer. This experience underscores the importance of a nuanced approach to hyperparameter tuning, as well as the acknowledgment that not all models may benefit from such optimization strategies.

B. Model simplification

Model simplification in deep learning refers to the process of reducing the complexity and size of a neural network while maintaining acceptable performance levels. This technique aims to achieve several objectives, including improved model interpretability, reduced computational resource requirements, and enhanced generalization on limited data [43]. On the observed models we made the following architectural adjustments:

- **MLP**: reduced the number of hidden layers
- **CNN**: the number of channels in convolution layers has been decreased to the 2/3 of the baseline
- **RNN**: the original two recurrent layers have been replaced by a single one

TABLE V: Effect of Resampling and Simplified models on the results in Table III

	All same	**At least 4**	**At least 3**	**At least 2**	**All different**
MLP	17	27	51	125	105
CNN	0	0	0	5	225
RNN	4	9	34	93	137

By eliminating unnecessary model components, we streamlined the network, making it more efficient and stable. The runs with the explained simplifications produced a bit more stable results with the MLP and RNN models and had negligible impact on CNN values. However, these models still do not seem to produce reliable outcomes.

C. Resampling

Resampling techniques can be roughly categorized into three commonly used types: oversampling, undersampling, and sampling with the creation of artificial data [3]. In our experiments, we applied oversampling, which is simple yet effective and incurs a little cost. The approach first identifies failing test cases; then iteratively resamples failing test cases into original test cases; finally stops the iterative resampling

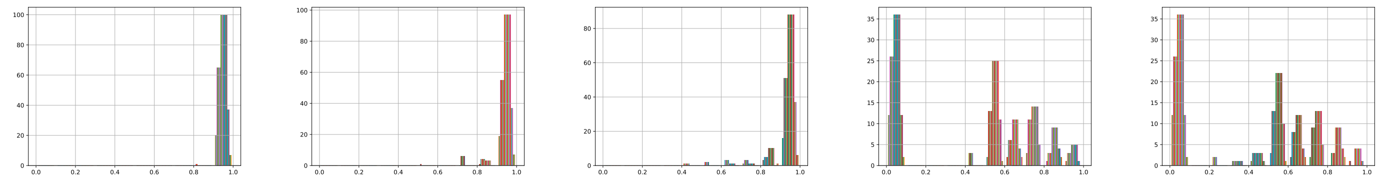

(a) Origin churn (b) BoxChurn, box size = 5 (c) BoxChurn, box size = 10 (d) FlBoxChurn, box size = 5 (e) FlBoxChurn, box size = 10

Fig. 4: Histogram of churn values measured using the MLP model on the observed programs.

TABLE VI: Effect of using the Resampling and the Simplified models combined on Average *Expense* results

	Mean			**Standard deviation**			**Minimum**			**Maximum**		
	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN	MLP	CNN	RNN
Chart	188.72	595.78	209.87	67.42	302.65	55.67	85.33	204.33	143.33	246.48	952.48	263.81
Lang	28.00	354.93	94.39	13.01	146.31	12.57	14.80	182.28	80.32	45.54	541.61	111.25
Math	124.05	860.38	209.13	51.32	432.69	58.57	67.88	361.98	150.30	193.82	1396.74	289.84
Mockito	336.86	617.54	278.98	117.34	298.77	103.02	193.19	270.32	146.16	468.57	985.00	404.30
Time	997.94	1945.97	337.43	413.92	1050.84	162.31	503.14	595.71	135.71	1514.43	3120.79	535.86
Total	163.64	687.35	191.91	63.62	337.15	55.61	87.88	289.17	128.78	241.11	1100.74	262.89
Diff	-287.23	-3.94	-484.58	-190.20	-36.41	-191.14	-89.82	29.51	-268.01	-536.89	-45.06	-730.28
Diff %	-63.71	-0.57	-71.63	-74.94	-9.75	-77.46	-50.55	11.37	-67.54	-69.01	-3.93	-73.53

process until obtaining a balanced test suite, where the number of failing test cases is the same as that of passing test cases.

By applying resampling, we found that the stability of the models increased significantly with MLP and RNN models, it had a bit of positive impact on CNN, but they still do not provide reliable results.

Due to limited space, our detailed results of using simplified models and resampling in DLFL appear in the online appendix [28], [29].

D. Resampling and Model simplification

In this section, we present results on the combination of model simplification and resampling.

In Table V, we can see the variety of each five runs of the simplified models with resampling technique usage. The MLP and RNN model values are in some case more similar to each other compared to the baseline in Table III, however there are still high variety in results. Still around half of the runs produce different expense every time in MLP and RNN models. The other 50% cases the models mostly produces at least 2–3 times same expense values from the 5 separate runs, which is an improvement compared to the baseline. There are also more cases when these 2 architectures produced same results each time from the 5 case. CNN model seems to have no improvement in this point of view, in 97% of cases it produced different results each time.

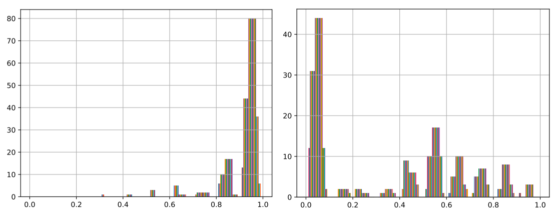
In Table VI, we can see the mean values of DLFL using the two new improvement technique. The structure of table is same as for Table II, but there are two extra rows, which show the absolute and percentage difference compared to the baseline. We can see that all models improved in average expense and in standard deviation as well. There are 70% improvements at MLP and RNN model in standard deviation, which indicates that stability seems to be improved. The maximal and minimal expense values both are highly decreased, however their ratios still seems to be too high to call models stable. Like in the previous section, CNN seems to be not really improved, the combined technique does not make it more stable. We used statistical significance testing here as well, like in Section VI-A: the maximal values, despite our improvements, were still significantly greater than minimals in all cases, however the effect sizes decreased a bit in some cases.

In Table VII, we examined the effect of our improvements on the DLFL *Top-N*. As we can see, there are still big differences between the selection of expenses from the five runs. The best case, the minimal selection, has higher values than the baseline had in Table IV, so we improved results in terms of fault localization effectiveness. However, if we check the mean and maximal selection, we can see that those values are still smaller than the minimal, indicating that random seed has a large effect on these results. The ratio of *Top-N* between the selections are better than at the baseline, but models still seems to be too unstable to give reliable results.

TABLE VII: Effect of Resampling and Simplified models on DLFL *Top-N*

		Top-1	*Top-3*	*Top-5*	*Top-10*	*Other*
MIN	MLP	59	102	121	147	83
	CNN	2	9	17	29	201
	RNN	37	68	83	109	121
MEAN	MLP	10	52	70	112	118
	CNN	0	0	0	1	229
	RNN	3	31	44	75	155
MAX	MLP	10	41	60	84	146
	CNN	0	0	0	0	230
	RNN	3	21	33	55	175

Churn measurements support the statistical analysis. In the following discussion, we continued our experiments with a box size of 10. In Figure 5, we depicted the histograms of boxed churn values using the MLP model. The histograms therefore represent the same data as in Figure 4, but the bars on the left side of the x-axis are higher than before. This essentially means that churn has been somewhat reduced. To quantify this improvement: the churn value decreased by 0.0063 on average, resulting an average value of 0.98 in case of $Churn$, 0.93 in case of $BoxChurn$ and 0.32 in case of $FlBoxChurn$. Although the improvement is measurable, this can only be considered as moderate success.



(a) BoxChurn, box size = 10 (b) FlBoxChurn, box size = 10

Fig. 5: Histogram of churn values measured using the simplified MLP model and resampling.

Answer to RQ2: *Enhancement of stability and applicability can be achieved to a certain extent; however, the improvements may not suffice to ensure consistently reliable outcomes. Both statistical and churn measurements confirmed that stability improved, but the models remained insufficiently stable to produce reliable results in practical applications.*

VIII. Discussion and Threats to Validity

We only examined the issue of stability in the FL domain; however, this might be a general problem in SE research using DL. DL reproducibility can be largely supported by sharing a reproduction package, however without examining the stability of the proposed model, its applicability is limited. As we have seen, standard statistical measures serve as a good indicator of whether a model is stable or not, and the adapted churn metric can further support such arguments. By applying our method, we believe better practices can be built for publishing DL applications and testing their stability.

Another question that arises is what can we conclude about the reasons for the instability and how it can be mitigated. As we have seen in RQ2, the major problems which we managed to moderate were the imbalanced data, subomptimal network architecture and parameterization, and in general, suitability of DL for SBFL. In particular, Zhang *et al.* [2] concluded in their findings that CNN performs the best in locating faults, even achieving better results than Dstar. Although in this study we did not concentrate on FL performance, rather on stability, we found that CNN was the most unstable of the models under test. CNNs might have demonstrated remarkable performance in image classification, object detection, and image segmentation tasks [44] where the order of the data carries important information, but in the FL domain we found MLP to provide more reliable results.

A possible threat to validity of our research relates to the benchmark we used. We only utilized the Defects4J bug database for evaluation, however it is the most commonly employed dataset in the literature for conducting similar studies. We believe that this choice does not impact the assessment of stability. Also, not all bugs from the benchmark were used in our experiments, due to hardware limitations. This does not limit the validity of the results, as if a model proves unstable even on this subset, it implies instability across the entire dataset. In this study we did not investigate the impact of low convergence levels, posing a threat to the validity of the findings. In scenarios when the primary objective is not generalization to new data, omitting a traditional train-test-validation split and convergence levels may align with the goals. Our online appendix package [28], [29] ensures full reproducibility of the research.

IX. Conclusions

In this paper, we examined the stability and suitability of DLFL models to provide reliable solutions to the fault localization problem. By selecting 230 versions from the Defects4J benchmark, we trained deep learning models 5 times, independently. We evaluated the stability of the observed models using mean, standard deviation and churn, which showed that the produced ranks are very different from each other. Our results show that, on average, 90% of the produced ranks were different in subsequent trainings. Furthermore, we proposed potential improvements to this problem, achieving moderate success: metaparameter optimisation techniques did not improve stability, but model simplification and oversampling reduced the mean of ranks by 45.3% and standard deviation by 54.04% on average. Churn also has been reduced, but even with these improvements, the models remained insufficiently stable to produce reliable results.

The implications of our research are twofold. First, SBFL research employing machine learning techniques should pay much higher attention on model stability because it decisively impacts the soundness and significance of the results. Second, other SE fields employing DL could also benefit from our stability measurement approach.

Acknowledgement

This work was supported in part by the ÚNKP-23-3-SZTE-435 New National Excellence Program of the Ministry for Culture and Innovation, by the national project TKP2021-NVA-09 implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the source of the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme and by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, and D. Li, *Software Fault Localization: an Overview of Research, Techniques, and Tools*, 2023, pp. 1–117.

[2] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, “A study of effectiveness of deep learning in locating real faults,” *Information and Software Technology*, vol. 131, p. 106486, 2021.

[3] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, “Improving deep-learning-based fault localization with resampling,” *Journal of Software: Evolution and Process*, vol. 33, no. 3, p. e2312, 2021.

[4] Z. Zhang, Y. Lei, X. Mao, M. Yan, and X. Xia, “Improving fault localization using model-domain synthesized failing test generation,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 199–210.

[5] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 48–60. [Online]. Available: https://doi.org/10.1145/3510003.3510136

[6] Y. Lei, C. Liu, H. Xie, S. Huang, M. Yan, and Z. Xu, "Bcl-fl: A data augmentation approach with between-class learning for fault localization," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 289–300.

[7] J. Hu, H. Xie, Y. Lei, and K. Yu, "A light-weight data augmentation method for fault localization," *Information and Software Technology*, vol. 157, p. 107148, 2023.

[8] Y. Lei, T. Wen, H. Xie, L. Fu, C. Liu, L. Xu, and H. Sun, "Mitigating the effect of class imbalance in fault localization using context-aware generative adversarial network," *arXiv preprint arXiv:2303.06644*, 2023.

[9] L. Fu, Y. Lei, M. Yan, L. Xu, Z. Xu, and X. Zhang, "Metafl: Metamorphic fault localisation using weakly supervised deep learning," *IET Software*, 2023.

[10] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, pp. 573–597, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:14477519

[11] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 664–676. [Online]. Available: https://doi.org/10.1145/3468264.3468580

[12] A. M. Mohsen, H. Hassan, R. Moawad, and S. H. Makady, "A review on software bug localization techniques using a motivational example," *International Journal of Advanced Computer Science and Applications*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247211009

[13] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, and X. Yang, "On the reproducibility and replicability of deep learning in software engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, oct 2021. [Online]. Available: https://doi.org/10.1145/3477535

[14] Q. Cormier, M. M. Fard, K. Canini, and M. R. Gupta, "Launch and iterate: Reducing prediction churn," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 3179–3187.

[15] S. Bhojanapalli, K. Wilber, A. Veit, A. S. Rawat, S. Kim, A. K. Menon, and S. Kumar, "On the reproducibility of neural network predictions," *ArXiv*, vol. abs/2102.03349, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231839490

[16] Z. Ren, T. Lin, K. Feng, Y. Zhu, Z. Liu, and K. Yan, "A systematic review on imbalanced learning methods in intelligent fault diagnosis," *IEEE Transactions on Instrumentation and Measurement*, 2023.

[17] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 664–676.

[18] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[19] X. Fang, X. Gao, Y. Wang, Z. Liao, and Y. Ma, "Improving fault localization using conditional variational autoencoder," *IEICE TRANSACTIONS on Information and Systems*, vol. 105, no. 8, pp. 1490–1494, 2022.

[20] J. Qian, X. Ju, and X. Chen, "Gnet4fl: Effective fault localization via graph convolutional neural network," *Automated Software Engg.*, vol. 30, no. 2, apr 2023. [Online]. Available: https://doi.org/10.1007/s10515-023-00383-z

[21] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[22] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[23] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," *IEEE Transactions on software engineering*, vol. 47, no. 7, pp. 1368–1380, 2019.

[24] M. Morin and M. Willetts, "Non-determinism in tensorflow resnets," *ArXiv*, vol. abs/2001.11396, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:210965994

[25] H. Jiang, H. Narasimhan, D. Bahri, A. Cotter, and A. Rostamizadeh, "Churn reduction via distillation," *arXiv preprint arXiv:2106.02654*, 2021.

[26] M. Klabunde, T. Schumacher, M. Strohmaier, and F. Lemmerich, "Similarity of neural network models: A survey of functional and representational measures," *arXiv preprint arXiv:2305.06329*, 2023.

[27] O. E. Gundersen, K. L. Coakley, and C. R. Kirkpatrick, "Sources of irreproducibility in machine learning: A review," *CoRR*, vol. abs/2204.07610, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.07610

[28] "Supplemental material for on the stability and applicability of deep learning in fault localization," 2023. [Online]. Available: https://github.com/sed-szeged/deepfl-stability

[29] R. Aszmann, "sed-szeged/deepfl-stability: On the Stability and Applicability of Deep Learning in Fault Localization," Jan. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10496189

[30] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, and D. Li, "Software fault localization: an overview of research, techniques, and tools," *Handbook of Software Fault Localization: Foundations and Advances*, pp. 1–117, 2023.

[31] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. Van Hoorn, A. Filieri, and D. Lo, "An evaluation of pure spectrum-based fault localization techniques for large-scale software systems," *Software: Practice and Experience*, vol. 49, no. 8, pp. 1197–1224, 2019.

[32] A. Zakari, S. P. Lee, and I. A. T. Hashem, "A single fault localization technique based on failed test input," *Array*, vol. 3, p. 100008, 2019.

[33] X. Xu, V. Debroy, W. E. Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, pp. 803–827, 2011.

[34] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, 2007, pp. 449–456.

[35] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[36] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.

[37] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[38] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 10 2016, pp. 267–278.

[39] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209.

[40] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[41] W. J. Conover, *Practical nonparametric statistics*. John Wiley & Sons, 1998, vol. 350.

[42] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[43] M. M. Bejani and M. Ghatee, "A systematic review on overfitting control in shallow and deep neural networks," *Artificial Intelligence Review*, pp. 1–48, 2021.

[44] S. Cong and Y. Zhou, "A review of convolutional neural network architectures and their optimizations," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 1905–1969, 2023.