DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization

Xia Li
UT Dallas, USA
xli124730@utdallas.edu

Wei Li
SUSTech*, China
liw7@mail.sustech.edu.cn

Yuqun Zhang
SUSTech, China
zhangyq@sustech.edu.cn

Lingming Zhang
UT Dallas, USA
lingming.zhang@utdallas.edu

ABSTRACT

Learning-based fault localization has been intensively studied recently. Prior studies have shown that traditional Learning-to-Rank techniques can help precisely diagnose fault locations using various dimensions of fault-diagnosis features, such as suspiciousness values computed by various off-the-shelf fault localization techniques. However, with the increasing dimensions of features considered by advanced fault localization techniques, it can be quite challenging for the traditional Learning-to-Rank algorithms to automatically identify effective existing/latent features. In this work, we propose DeepFL, a deep learning approach to automatically learn the most effective existing/latent features for precise fault localization. Although the approach is general, in this work, we collect various suspiciousness-value-based, fault-proneness-based and textual-similarity-based features from the fault localization, defect prediction and information retrieval areas, respectively. DeepFL has been studied on 395 real bugs from the widely used Defects4J benchmark. The experimental results show DeepFL can significantly outperform state-of-the-art TraPT/FLUCCS (e.g., localizing 50+ more faults within Top-1). We also investigate the impacts of deep model configurations (e.g., loss functions and epoch settings) and features. Furthermore, DeepFL is also surprisingly effective for cross-project prediction.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging.

KEYWORDS

Fault localization, Deep learning, Mutation testing

ACM Reference Format:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

* SUSTech is short for Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.

1 INTRODUCTION

It has been reported that debugging software faults can take up to 80% of the total software cost for some projects [1]. Therefore, it is essential to develop automated techniques to help reduce the manual efforts during the software debugging process. In the literature, various fault localization techniques [2–10] have been proposed to automatically localize potential faulty code locations. Fault localization techniques usually analyze various dynamic execution information to compute the suspiciousness (i.e., probability to be faulty) of each program element (such as statement or method). Then, a ranked list of program elements (based on the descending order of suspiciousness values) can be either provided to developers for manual fixing or serve as the first step of automated program repair [11–17]. Please refer to a recent survey for more details [18].

Among the existing fault localization methodologies, spectrum-based fault localization (SBFL) has been intensively studied due to its lightweightness and effectiveness. Typical SBFL techniques (such as Tarantula [19], Ochiai [20], DStar [21], Jaccard [5], Kulczynski [22]) simply apply statistical analysis on the coverage data of failed/passed tests to compute the suspiciousness of code elements. The basic intuition is that code elements executed by more failed tests are more suspicious. Despite widely studied, SBFL has clear limitations in design, i.e., elements executed by failed tests may not have caused the test to fail and faulty elements may also be executed by passed tests coincidentally. To bridge the gap, mutation-based fault localization (MBFL) was proposed to mutate source code to check the actual impact of each code element on the outcomes of tests [23–26]. Typical MBFL techniques (such as FPIF [27], Metallaxis [26] and MUSE [23]) first apply mutation testing [28] to generate mutants for the original program under test. Each mutant is exactly the same with the original program but with one syntactic change based on the predefined rules (called mutation operators, such as changing $1f(a>b)$ into $1f(a<b)$). Then, MBFL techniques use mutants to check the impacts of code elements on the test outcomes for precise fault localization.

Although MBFL techniques consider the impact information, they may still perform poorly for some cases, e.g., some elements may not have any mutant to simulate its impact [29]. Actually, to date, there is no optimal fault localization technique that can perform the best for all cases. Therefore, recently, researchers started to combine the strengths of various traditional fault localization techniques via machine learning. Learning-to-Rank [30], a supervised machine learning technique for solving ranking problems in the field of information retrieval, has been widely used to combine the suspiciousness values computed by various fault localization techniques for more effective fault localization [8, 9, 29, 31]. In typical Learning-to-Rank fault localization techniques, various suspiciousness values computed by different fault localization techniques...
are used as learning features and whether the element is faulty is treated as the label information. E.g., MULTRIC [9] combines various SBFL techniques via Learning-to-Rank, while TraPT [29] combines SBFL and MBFL techniques via Learning-to-Rank. Although Learning-to-Rank has been demonstrated to be effective for fault localization, it may not fully utilize the training data information since it lacks the capability to automatically select existing powerful features and discover new advanced features. Therefore, in this work, we propose DeepFL, a general deep-learning approach to automatically identify or create the most effective features for precise fault localization. DeepFL takes suspiciousness-based features from the fault localization area (including both SBFL and MBFL), fault-proneness-based features (e.g., code-complexity metrics) from the defect prediction area [32] and textual-similarity-based features from the information retrieval area [33]. We have built various DeepFL techniques based on the TensorFlow framework [34], including the basic Multi-layer Perceptron (MLP) and Recurrent Neural Networks models, as well as our tailored model variants considering the hierarchical connections between different feature groups for deep fault localization (i.e., MLP_{DFL}). To evaluate DeepFL, we perform a study on 395 real software faults from the widely used Defects4J benchmark (V1.2.0) [35]. The experimental results show that DeepFL can significantly outperform state-of-the-art TraPT [29] and FLUCCS [31] (e.g., localizing 213 faults within Top-1, 50+ more than TraPT/FLUCCS). The study also investigates the impacts of different deep model configurations and features. The results reveal various interesting findings, including: (1) deep models considering hierarchical feature group connections can significantly outperform the traditional Learning-to-Rank algorithms, while simply increasing deep learning layers does not help; (2) the mutation-based features are the most important ones for fault localization; (3) the softmax loss function is more stable than the pairwise loss function for DeepFL. Finally, the study shows that DeepFL can even perform precise fault localization for cross-project prediction, indicating a promising future for deep-learning-based fault localization. This paper makes the following contributions:

- **DeepFL** A deep-learning-based approach to predict potential faulty locations via incorporating various dimensions of fault diagnosis information.
- **Techniques** A set of DeepFL techniques (implemented using TensorFlow) based on widely used neural networks (such as MLP, RNN), and our tailored MLP_{DFL}.
- **Study** An extensive study demonstrating the effectiveness of DeepFL techniques on localizing real-world bugs as well as investigating various configurations of DeepFL.
- **Dataset** An extensive fault localization dataset for all 395 Defects4J V1.2.0 bugs, including 200+ unique dynamic or static fault localization features for each program element within each buggy version.

2 BACKGROUND AND RELATED WORK

2.1 Traditional Fault Localization

**Spectrum-based Fault Localization.** SBFL [2, 5, 19, 20, 22, 36–39] has been intensively studied in the literature. Although various SBFL techniques have been proposed, they share the same basic insight, i.e., code elements mainly executed by failed tests are more suspicious. The input of SBFL is the coverage information of all tests and the output is a ranked list of code elements (e.g., statements or methods) according to their descending order of suspiciousness values calculated by specific formulae. To date, various SBFL techniques have been proposed, including Tarantula [40], SBI [37], Ochiai [20] and Jaccard [5]. The calculation of these SBFL techniques mainly rely on: (1) the set of all failed/passed tests, i.e., \( T_f/T_p \), (2) the set of failed/passed tests executing code element \( e \), i.e., \( T_f(e)/T_p(e) \), and (3) the set of failed/passed tests that do not execute code element \( e \), i.e., \( T_f(e)/T_p(e) \). For example, SBI formula can calculate the suspiciousness value of one code element \( e \) as \( \text{Susp}(e) = |T_f(e)/(T_f(e)+|T_p(e)|). \) The objective of SBFL is to rank the actual faulty elements as high as possible to save developers’ efforts in manually inspecting the ranked elements, or save the CPU time during automated program repair [13–15, 41–44].

**Mutation-based Fault Localization.** MBFL [23–27] aims to additionally consider impact information for fault localization. Since code elements covered by failed/passed tests may not have any impact on the corresponding test outcomes, typical MBFL techniques use mutation testing [28, 45, 46] to simulate the impact of each code element for more precise fault localization. Here we mainly discuss the general MBFL techniques since the regression MBFL techniques (e.g., FIFL [27]) are not closely related to this work. The first general MBFL technique, Metallaxis [24, 26] is based on the following intuition: if one mutant has impacts on failed tests (e.g., the tests outcomes change after mutation), its corresponding code element may have caused the test failures; similarly, if one mutant has impacts on passed tests, its corresponding code element may not be faulty (otherwise the passed tests would have failed). Metallaxis treats mutants that have impacts on tests as elements covered by the tests while the others as uncovered. Then it applies traditional SBFL formulae to calculate the suspiciousness of each mutant. Finally, the maximum suspiciousness value of mutants is returned as the suspiciousness of their corresponding code element. Assume formula SBI is used, the suspiciousness of mutant \( m \) is \( \text{Susp}(m) = |T_f(m)(e)/|T_f| - \alpha \cdot |T_p(m)(e)/|T_p| \), where \( T_f(m)(e)/|T_f| \) is the number of failed/pass tests impacted by \( m \) on element \( e \).

The more recent MUSE [23] technique has similar insights: (1) mutating faulty elements may cause more failed tests to pass than mutating correct elements; (2) mutating correct elements may cause more passed tests to fail than mutating faulty elements. In MUSE, the suspiciousness of mutant \( m \) is computed as \( \text{Susp}(m) = |T_f(m)(e)|/|T_f| - \alpha \cdot |T_p(m)(e)|/|T_p| \), where \( T_f/T_p \) denotes the set of originally passed/failed tests, and \( T_f(m)(e)/T_p(m)(e) \) denotes the set of originally failed/passed tests that have changed outcomes on mutant \( m \). \( \alpha \) is a balancing weight and can be calculated as \( f_{2p}/|T_f| \) or \( f_{2p}/|T_f| \), where \( f_{2p}/|T_f| \) denotes the total number of failed/passed tests that changed outcomes during mutation.

2.2 Learning-Based Fault Localization

Learning-to-Rank is an application of supervised machine learning for solving ranking problems in information retrieval. In the learning phase, Learning-to-Rank takes a group of training data as input, and learns a ranking model by taking specific attributes of documents and queries as different features, e.g., cosine similarity.
and proximity values. The ranking model is usually an optimal combination of weights for different features. Then, in the ranking phase, the ranking model is used to predict a ranked list of documents by accepting a set of test data including new queries and documents. Learning-to-Rank has three major categories of approaches: (1) pointwise approach, (2) pairwise approach, and (3) listwise approach. Pointwise indicates that each document in the training data has its own label. Pairwise indicates that each pair of two documents can be computed a label based on their ordering for the given query. Listwise indicates that the order of a list of documents is considered for prediction.

Recently, Learning-to-Rank has been applied to improve the effectiveness of spectrum-based fault localization [8, 9, 29, 31]. The basic idea of Learning-to-Rank fault localization is to learn the potential faulty locations via combining suspiciousness values computed by various fault localization techniques (i.e., features). MULTRIC [9] is the first Learning-to-Rank fault localization technique to combine different suspiciousness values from SBFL. Later on, researchers have also combined SBFL suspiciousness values with other information, e.g., program invariant [8] or source code complexity information [31], for more effective Learning-to-Rank fault localization. TraPT [29] has been proposed to combine suspiciousness values not only from SBFL, but also from MBFL. More specifically, TraPT further extends Metallaxis and MUSE to have MBFL at different test outcome levels. Recently, another independent work also proposed to combine various different existing techniques via Learning-to-Rank [47]. In all traditional Learning-to-Rank fault localization techniques, pairwise approach is selected because fault localization aims to rank faulty elements higher than correct ones, and other relationships within faulty or correct elements are not considered. Formally, assume \( x_e \) denotes the corresponding training feature vector for element \( e \), then the \( i \)th feature value of \( e \) is \( x_{ei} \) (\( i = 1, 2, \ldots, n \)). A Learning-to-Rank algorithm can learn the weight for each feature of \( e \) as \( W_{ei} \) (\( i = 1, 2, \ldots, n \)), and the new combined suspiciousness value of element \( e \) can be calculated as: \( \tilde{y}_e = W_{ei}^T x_e \).

The code elements can be ranked according to new suspiciousness values, and the loss function can be defined as the number of incorrectly rankings: \( L = \sum_{(e^+, e^-)} \| \tilde{y}_{e^+} - \tilde{y}_{e^-} \| \), where \( e^+ \) and \( e^- \) denote any pair of faulty and correct code elements.

Actually, various other machine learning techniques, and even neural networks have also been applied to fault localization [48–51]. However, those techniques mainly work on the test coverage information, which has clear limitations (e.g., it cannot distinguish elements accidentally executed by failed tests and the actual faulty elements) [29], and are usually studied on artificial faults or small programs. In contrast, DeepFL is the first deep-learning-based approach to incorporate various dimensions of fault diagnosis information, and has been studied on real faults of real-world projects.

3 APPROACH

In this section, we discuss the basic ideas of the used deep learning models, as well as how we apply them to DeepFL (Section 3.1). Then we introduce the features used in DeepFL (Section 3.2). Finally, we discuss about the DeepFL loss functions (Section 3.3).

### 3.1 Deep Learning for Fault Localization

Previous fault localization techniques (such as MULTRIC [9], Savant [8], FLUCCS [31], and TraPT [29]) all utilized the Learning-to-Rank algorithms to predict potential fault locations based on various fault diagnosis features (shown in Section 3.2). However, traditional machine learning techniques (including Learning-to-Rank) largely rely on existing features, and cannot discover new latent features for more effective learning. Furthermore, it is also important to identify the most important features for effective fault localization with the increasing number of features. Recently, deep learning has demonstrated its superiority in feature engineering (i.e., including selecting useful existing features and learning latent features), and has been widely used to solve software engineering problems, such as defect prediction [32] and requirements traceability [32]. Deep learning [53] is the application of artificial neural network (ANN) with hidden layers to solve machine learning tasks. Backpropagation [54] is widely used for training and adjusting ANN internal weights to better compute the representation in each layer. In this section, we will introduce how we adapt the traditional MLP [55] and RNN networks [56] for fault localization; lastly, we will also present our tailored MLP models for deep fault localization.

#### 3.1.1 Multi-Layer Perceptron

Multi-layer Perceptron (MLP) is a basic class of feedforward ANNs which indicates that the network does not have any loop and the output of each node does not affect the node itself [57]. MLP is a supervised learning algorithm that learns a function \( f \) mapping from \( \mathbb{R}^n \) to \( \mathbb{R}^l \) by training a dataset which includes \( n \) features and \( l \) labels. It can learn a non-linear function approximator for either classification or regression problem. Assume that there is a set of training data \( D:\{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\} \), where \( x_i \in \mathbb{R}^n \) and \( y_i \in \mathbb{R}^l \), and one hidden layer with \( k \) nodes, the function that MLP learns is as following:

\[
f(x) = \sigma_h(W_{ih}^T \cdot x + b_h) + b_o
\]

where \( W_{ih} \in \mathbb{R}^{n \times k} \) represents the weights between input layer and hidden layer and \( W_{ho} \in \mathbb{R}^{k \times l} \) represents the weights between hidden layer and output layer. \( b_h \in \mathbb{R}^k \) and \( b_o \in \mathbb{R}^l \) represent the bias of hidden layer and output layer, respectively. \( \sigma_h \) and \( \sigma_o \) represents the activation functions (such as \( \tanh, ReLU, sigmoid, \) and \( \text{softmax} \)) for the hidden layer and output layer, respectively. Figure 1 (where \( z^{(i)} \) presents the \( i \)th dimension of vector \( z \)) presents the definitions for four widely used activation functions in neural networks. Usually, \( \tanh, ReLU, \) and \( \text{sigmoid} \) can be used as hidden layer activation function, while \( \text{softmax} \) can be used as the output layer activation function for multi-class classification problems.

To illustrate, Figure 2 shows a classic MLP with one input layer, one hidden layer and one output layer.

For our study of fault localization using MLP, we directly feed our training data to MLP and use the MLP output information to
3.1.2 Recurrent Neural Network. Recurrent Neural Network (RNN) is widely used in Natural Language Processing [56]. The primary difference between RNN and other deep learning models is that RNN considers the time-evolving state. Figure 3 shows a classic structure of RNN and a fully unfolded network. "Unfolded" means that the network is represented as the complete sequence. For example, if the input is a sequence of 5 words, the network would be unfolded into 5-layer. In this figure, $x_t$ is the input and $h_t$ is the output at time state $t$. Assume the input layer has $n$ nodes and the hidden layer has $k$ nodes, $h_t$ can be calculated based on the previous hidden output $h_{t−1}$ and the input at the current state:

$$h_t = \tanh(W^x_t x_t + U h_{t−1} + b)$$

where $W \in \mathbb{R}^{n \times k}$, $U \in \mathbb{R}^{k \times k}$ and $b \in \mathbb{R}^k$ represent the weight matrices and bias vector.

However, standard RNN has a primary disadvantage that the ability of network may downgrade due to vanishing gradient [58]. To overcome this disadvantage, a variant of RNN called Long Short Term Memory (LSTM) was introduced [59] to preserve long-term dependencies. The basic idea of LSTM is that a memory cell vector is introduced to preserve its state over time. The memory cell consists of an explicit memory and gating units which can control the information flow into and out of the memory. LSTM uses input gates to control what new information is added to cell state from current input, forget gates to control what information to throw away from memory, and output gates to decide what information to output from the memory. The state of each gate is decided by $x_t$ and $h_{t−1}$. The state of forget and input gates can be calculated as:

$$f_t = \text{sigmoid}(W^f_t x_t + U_f h_{t−1} + b_f)$$

$$i_t = \text{sigmoid}(W^i_t x_t + U_i h_{t−1} + b_i)$$

To update the information in the memory cell, a memory candidate vector $c_t$ is firstly calculated. Then, the cell state vector aggregates old memory via the forget gate and new memory via the input gate, to update its information (Hadamard product $\odot$ is used to control the information passing the gates):

$$c_t = f_t \odot c_{t−1} + i_t \odot \tilde{c}_t$$

where $W \in \mathbb{R}^{k \times k}$, $b \in \mathbb{R}^k$ and $\tilde{c}_t$ is the Hadamard product of $\tanh(W^c_t x_t + U_c h_{t−1} + b_c)$ and $i_t$.

Finally, LSTM calculates its output $h_t$ based on output gate state:

$$o_t = \text{sigmoid}(W^o_t x_t + U_o h_{t−1} + b_o)$$

$$h_t = o_t \odot \tanh(c_t)$$

When performing learning-based fault localization, we found that similar features may provide fault diagnosis information from the same dimension or information source. For example, all the suspicious values computed by spectrum-based fault localization techniques try to provide fault diagnosis information from the execution dimension, while all the code-based metric values try to provide fault diagnosis information from the code complexity dimension. Therefore, we can group all the available features into different groups based on their information sources. Then, the features in the same group can be processed together first to provide the most useful fault diagnosis information from that information source. Finally, the fault diagnosis information from different information sources can be further linked together to provide the final fault diagnosis supports. Actually, RNN with LSTM is a natural choice – different feature groups can be treated as inputs for different time steps in RNN; different feature groups can also be linked together via the shared time state to provide the final output. Figure 4 presents our network architecture of the deep fault localization technique via RNN. Shown in the figure, the inputs are the original features of each instance program element. Shown in Section 3.2, the 7 different feature groups of DeepFL can have different sizes, while RNN requires inputs for different time steps to have the same shape. Therefore, each feature group is normalized to have the same length as the largest feature group via zero padding. Then, the normalized feature groups can be represented as a set of vectors with uniform length, which can be directly fed into RNN for deep fault localization. By design, each RNN unit will compute the current feature group information while also considering the earlier feature groups. Then, the output of the last RNN unit actually integrates all the information learnt from all the feature groups. Finally, the output of the last RNN unit will go through the last integration layer to compute the final suspiciousness value for the program element. Assume each RNN unit has $r$ hidden nodes, then the output of the last RNN unit for input $x_t$ can be represented as $z_t \in \mathbb{R}^r$, and the final prediction results can be computed as:

$$\hat{y}_t = \text{softmax}(W^r x_t + b)$$

where $W \in \mathbb{R}^{r \times 2}$, $b \in \mathbb{R}^r$ since there are only two final output nodes (i.e., denoting buggy or correct).

3.1.3 Tailored MLP Based Neural Network. The classic MLP model introduced in Section 3.1.1 treats all the feature groups uniformly;
the hidden and output layers then compute the final prediction results via connecting all the feature groups. Such models may incur a huge number of weights (for connecting all the feature groups via fully-connected layers) to be trained, making the model training expensive and inferior. In fact, as discussed in Section 3.1.2, different feature groups contain different dimensions of debugging information, and can be processed separately first and then connected later to save the number of weights. Indeed, the RNN model discussed in Section 3.1.2 actively considered feature group information and can greatly reduce the number of weights. However, the RNN model has two clear drawbacks: (1) all the feature groups have to share the same weights based on the native design of RNN, while different feature groups contain different debugging information and should not be processed in the same way; (2) the RNN model is quite inflexible and can only take a set of plain feature groups and treat them uniformly, while different feature groups may share some hierarchical relationships (explained in more details in the next paragraph). Therefore, besides the commonly used deep learning structures, in this section we also design variants of the classic MLP model tailored for DeepFL, which we called MLP_{DFL}. MLP_{DFL}(1).

Figure 5 shows the network architecture of the basic variant, called MLP_{DFL}(1). As shown in the figure, each feature group is first connected into its own fully-connected layer to compute and extract the useful debugging information within this particular group. Then, the extracted debugging information for each group are concatenated together to construct a complete layer. The new complete layer can be viewed as the new hidden layer. Note that each node within this new hidden layer is only connected to the nodes within its corresponding feature group, while each node within the hidden layer of the traditional MLP is connected with nodes from all feature groups. In this way, a lot of unnecessary edges (and thus weights) can be reduced. Finally, the fully-connected layer is connected to the output layer to perform the prediction. Formally, the computation of MLP_{DFL}(1) is:

\[ h_t = \sigma_h(W_{th}^t x_t + b_t) \]
\[ g = [h_1, h_2, ..., h_n] \]
\[ f((x_1, x_2, ..., x_t)) = \sigma_o(W_{ho}^t g + b_o) \]

where \( x_t \in \mathbb{R}^{m_t} \) represents the tth feature group (\( m_t \) is the number of features within this group), while \( h_t \in \mathbb{R}^{k_t} \) represents the single fully-connected layer connected with the tth feature group (\( k_t \) is the number of nodes of the fully-connected layer). \( n \) represents the number of the feature groups (e.g., 7 for Figure 5). \( g \) represents the complete fully-connected layer and \( f \) represents the final output. \( W_{th}^t \in \mathbb{R}^{m_t \times k_t} \) represents the weights between input layer and fully-connected layer of tth feature group. \( W_{ho}^t \in \mathbb{R}^{(\sum_{\ell=1}^{n}k_{\ell}) \times k_{t}} \) represents the weights between complete fully-connected layer and output layer (\( l \) is the number of labels), \( b_{th} \) represents the bias of the fully-connected layer of tth feature group and \( b_o \) represents the bias of the output layer. \( \sigma_h \) and \( \sigma_o \) represents the activation functions for the fully-connected layer of tth feature group and output layer, respectively.

MLP_{DFL}(2). Note that some feature groups may share similar information and can also be connected earlier before connecting with all other feature groups. E.g., the four groups of mutation-based debugging information (shown in Section 3.2) are close to each other and should be merged first; also, the merged mutation information is more similar to the spectrum-based feature group (since they are both dynamic execution-based debugging information), and should also be merged before merging with other groups. Therefore, we further design another advanced variant (called MLP_{DFL}(2)) by concatenating the single fully-connected layers of similar feature groups first. Figure 6 shows the architecture of MLP_{DFL}(2). For example, instead of connecting fully-connected layers of all feature groups, this mode first concatenates the fully-connected layers of feature groups \( x_4 \) to \( x_7 \) (assuming they correspond to the 4 mutation-based groups). Then, the extracted mutation-based information is further combined with the spectrum-based feature group (i.e., \( x_8 \)) since they both belong to dynamic-execution-based information. Lastly, the useful extracted dynamic-execution-based information is further concatenated with the rest two static feature groups. The motivation of the advanced MLP_{DFL}(2) variant is that it can dig more helpful and hidden debugging information in a hierarchical way before constructing the complete fully-connected layer, similar to the idea of the traditional Convolutional Neural Network (CNN) [60]. We omit the equations for the advanced variant since it can be easily inferred from Equation (8).

3.2 DeepFL Features

Existing learning-based fault localization studies [8, 9, 29, 31] have demonstrated that suspiciousness values computed from various traditional fault localization techniques can provide useful guidelines for localizing the buggy program elements. Recently, Li et al. [29] showed that combining spectrum-based and mutation-based suspiciousness values can achieve the best fault localization results. Although suspiciousness-based features are effective, they only consider the runtime correlation between program elements and the failed/passed tests. Actually, the fault-proneness of the program elements can also be useful for fault localization [31]. E.g., despite two program elements have equivalent suspiciousness values, one may still have higher probability to be faulty since it is simply more fault-prone (e.g., more complex). In addition, textual
similarity information from the information retrieval area can be used as another feature dimension because it can reflect the textual relevance between source code and failed tests. Thus, we include all above feature dimensions in our DeepFL general framework.

**Spectrum-based Suspiciousness.** Xuan and Monperrus firstly proposed MULTRIC [9] to learn faulty locations based on suspiciousness values computed by 25 traditional spectrum-based fault localization techniques, such as Jaccard [5], Ochiai [20], Ochiaiz [22], and Kulczynskiz [22]. Recently, Xie et al. [61] found that some manually-created spectrum-based formulae (such as ER1a and ER5c) can perform extremely well on single-fault programs in theory. In addition, Xie et al. [62] also generate a set of optimal spectrum-based formulae via genetic programming (GP). Following recent learning-based fault localization work [8, 29, 31], we use all the above mentioned formulae to collect the suspiciousness-based features. In total, we have the same 34 SBFL features as TraPT [29] (the duplicated formulae among prior work [9, 61, 62] were removed).

**Mutation-based Suspiciousness.** The recent TraPT work [29] by Li et al. firstly use suspiciousness values computed by MBFL techniques to perform learning-based fault localization, and demonstrate that they can significantly boost fault localization results. In that work, both existing general MBFL techniques (i.e., Metallaxis [26] and MUSE [23]) are used. Since Metallaxis uses traditional spectrum-based formulae, 34 Metallaxis variants are used based on the 34 used spectrum-based formulae. Furthermore, the TraPT work firstly shows that MBFL techniques can perform differently using different types of failure outputs/messages on different faulty programs. The reason is that different number of tests are computed as impacted based on different levels of test outcomes. For example, when a test $t$ changes its exception type on a mutant $m$, $t$ can be treated as impacted by $m$ when using exception type information as test outcomes, while not impacted by $m$ when using only pass/fail information as test outcomes (since $t$ fails both before and after $m$). Therefore, TraPT further considers MBFL results with the following four different types of test outcomes: (1) Type1: pass/fail information, (2) Type2: exception type information, (3) Type3: exception type and message, and (4) Type4: exception type, message, and the full stack trace information. Note that we do not use the assertion-level information since it is not applicable to all programs, but DeepFL is general and can easily include it in the future. In total, we implement 34 Metallaxis variants and MUSE at each of the four test outcome types studied in TraPT, yielding $(34 + 1) \times 4 = 140$ suspiciousness values.

**Complexity-based Fault Proneness.** In the literature, code complexity metrics have been widely used to estimate the fault-proneness of code elements in the field of defect prediction [32, 63–65]. Code metrics can measure various software characteristics to reveal quality information. For example, Cyclomatic Complexity measures the number of linearly independent paths within the underlying program elements, and a program element with more independent paths may have high probability to be faulty; Halstead Difficulty [66] measures the difficulty to write or understand (e.g. when doing code review) the underlying code elements, and it can be harder to test/verify a program element that is more difficult to write or review. Recently, code metrics have been utilized to learning-based fault localization [31]. However, only three types of code complexity metrics (number of formal arguments, local variables, and statements/instructions) were used. In addition, the code complexity metrics were always applied together with change history information, making prior work only applicable to a subset of projects. To fully evaluate the effectiveness of code complexity metrics for fault localization, we collect all the 21 widely-used code complexity metrics (shown in Table 1) for the method level (since DeepFL works at the method level). In addition, we also collect the statistics for each type of Java ASM bytecode instructions shown Table 1. In total, we have 37 complexity-based features.

**Textual Similarity Information.** Besides various suspiciousness and fault-proneness information, information retrieval (i.e., IR) techniques have also been applied for bug localization [33, 68–71]. Such IR-based techniques investigate the textual similarity between source files and bug reports, treating each bug report as a query and the source files to be searched as a document collection. Then such techniques can return a ranked list of candidate source files based their predicted relevance with bug reports. However, in practice, bug reports are not always available, thus limiting the applications of the IR-based techniques. In this work, inspired by the idea of IR-based techniques, we investigate the textual similarity between source code methods and failed test information. Similar with the structured information retrieval work [69], we also collect different fields for both queries and documents. Query fields come from failed tests, including the name of failed tests, the source code of failed tests and the complete failure message (including exception type, message, and stacktrace). Document fields come from source code methods, including: the full qualified name of the method, accessed classes, method invocations, used variables, and comments. Each field of query can be searched on each field of document, yielding 15 combinations. For each combination, we calculate the similarity score between the query field and the document field by using the popular TF.IDF model [72, 73]. Then, we treat the similarity scores of the 15 combinations as 15 features for our DeepFL. For example, Figure 7 shows the buggy method, failed test and failure message of our subject Lang-3. From this simple example, we can find that multiple occurrences of words “create”, “number” and “createnumber” can be extracted from both the failed test (including failure message) and the buggy method. In this case, the similarity score between them is high based on the IR techniques, which can further help improve the effectiveness of fault localization.

### 3.3 DeepFL Loss Function

Loss functions guide the learning process towards certain goals and are essential for DeepFL. The *pairwise* function has been widely used for the traditional Learning-to-Rank, thus we also adopt it for our DeepFL. We also use the *cross-entropy* function [74] since it has been widely used in deep learning based classification problems.
DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization

ISSTA ’19, July 15–19, 2019, Beijing, China

which has been widely used to penalize complex models to reduce overfitting.

4 EXPERIMENTAL SETUP

According to prior Learning-to-Rank work [75], the pairwise loss function which can be defined as:

$$L(\theta) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \phi(z) + \frac{\lambda}{2} ||\theta||^2_2$$

where $z = \hat{y}_i - \hat{y}_j$ and the function $\phi$ can be (1 - z), (hinge function), $e^{-z}$ (exponential function) or $\log(1 + e^{-z})$ (logistic function). $y_s$ or $\hat{y}_i$ denotes the label vector, and $\hat{y}_j$ or $\hat{y}_j$ denotes the prediction result for the sth and jth instance.

Cross-entropy. Since we transfer the fault localization problem into classification problem using MLP and RNN, we also use the widely used cross-entropy loss function for our approach. The cross-entropy loss function for m instances can be computed as:

$$L(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y_i^{(1)} \log \hat{y}_i^{(1)} + y_i^{(2)} \log \hat{y}_i^{(2)} + \frac{\lambda}{2} ||\theta||^2_2$$

where $y_i^{(1)}$ denotes the first element in vector $y$.

In the above loss functions, $\theta$ is the parameters involved in training the neural networks. $\frac{\lambda}{2} ||\theta||^2_2$ denotes the L2-Regularization, which has been widely used to penalize complex models to reduce overfitting. $\lambda$ is a hyperparameter that controls the magnitude of the penalty. Based on a certain loss function, various optimizer methods can be used to update the network parameters to minimize the loss, such as Stochastic Gradient Descent (SGD) [76], Nesterov Accelerated Gradient (NAG) [77], and Adaptive Moment Estimation (Adam) [78]. We use the Adam optimizer since it is a recent advanced optimizer handling both sparse gradients and non-stationary objectives well. Furthermore, Adam has been shown to work well in practice and converge faster than others [78].

4 EXPERIMENTAL SETUP

This work investigates the following research questions:

- **RQ1**: How does DeepFL perform in localizing real faults compared with state-of-the-art techniques?
- **RQ2**: How do different DeepFL architectures and different feature dimensions impact the DeepFL effectiveness?
- **RQ3**: How do different epochs and loss functions impact DeepFL results?
- **RQ4**: How does DeepFL perform in cross-project prediction?

To investigate the RQs, we use the Defects4J benchmark [35] that has been widely used in software testing research [8, 12, 13, 29, 31, 79]. We use all the 6 Defects4J V1.2.0 subjects, totaling 395 real faults detected during actual software development.

4.1 Implementation Details

Spectrum-based Features. We implement all the 34 spectrum-based fault localization formulae (Section 3.2) in Java. To collect coverage information required by SBFL, we perform on-the-fly bytecode instrumentation using ASM [67] and Java Agent [80].

Mutation-based Features. Following TraPT, we modify the widely used PIT [81] mutation testing tool (Version 1.1.5) to implement Metallaxis and MUSE: (1) we enable PIT on programs with failed tests by disabling the PIT is green check, (2) we force PIT to execute each mutant against the remaining tests after the mutant is killed by earlier tests, (3) we enable PIT to further capture detailed test outputs/messages (i.e., exception types, exception messages, and stack traces) via overriding onTestFailure(). Same with TraPT, we used all the 16 mutation operators available in PIT-1.1.5.

Complexity-based Features. For complexity-based features, we use JHawk [82] to collect all the 21 source code metrics at the method level by parsing generated XML files. We also use the ASM bytecode manipulation framework to collect 16 bytecode metrics by overriding the method instruction visitor methods, such as visitTypeInsn, and visitFieldInsn.

Textual-based Features. For textual-similarity-based features, we use Indri [83] (version 5.11), a search engine that provides state-of-the-art text search and a structured query language for text collections, to generate the 15 types of similarity scores between source code methods and failed tests. We follow similar configuration with prior work [69], e.g., keeping both full and split tokens during camel case splitting, setting the baseline method as tfidf, and setting parameters k1 and b as 1 and 0.3, respectively.

Learning Techniques. We build DeepFL models based on TensorFlow [34] (Version 1.5.0), one of the most widely used deep learning frameworks. For MLP, we implement two variants, one with one hidden layer and one with two hidden layers. For RNN, we implement Bidirectional RNN (BiRNN) with LSTM. While standard RNNs have restrictions as the future input information cannot be reached from the current state, BiRNNs allow future input information to be reachable from the current state. Lastly, we implement two variants of MLPDEFL - (1) the basic MLPDEFL(1) that treats all 7 groups of features uniformly, and (2) the hierarchical MLPDEFL(2). Note that MLPDEFL(2) is the default DeepFL model for this work.

For all studied techniques, we set the number of nodes in hidden layer(s) to be the same as the number of input features. We globally use the widely used learning rate of 0.001, batch size of 500, and the default training epoch of 55. We use the default softmax loss function (from tf.nn.softmax_cross_entropy_with_logits), and implement the pairwise loss function from scratch due to its absence in TensorFlow. Note that we tried all three $\phi$ functions for the pairwise loss function, and only kept the exponential one due to its effectiveness. Furthermore, we apply the L2-regularization with the widely used $\lambda$ value of 0.0001, and Dropout Regularization with the rate of 0.30 to reduce overfitting. For the baseline Learning-to-Rank technique, we use RankSVM with linear kernel (version 1.95) from the widely used LIBSVM [84]. We use the same LIBSVM settings as existing Learning-to-Rank fault localization techniques [8, 29, 31].
### 4.2 Measurements and Experimental Setup

Researchers have shown that statement-level fault localization may be too fine-grained and miss useful context information [85] and class-level fault localization is too coarse-grained to help understand and fix the bug within a class [6]. Furthermore, recent advanced program repair techniques also require precise localization of buggy methods [86, 87]. Therefore, following recent work on fault localization [8, 33, 79, 88], we perform fault localization techniques on method-level, i.e., localizing the faulty methods among all source code methods. The following measurements are used.

**Recall at Top-N:** In practice, most developers usually only inspect the top-ranked code elements during fault localization, e.g., 73.58% developers only check Top-5 localized elements according to a recent study [6]. Therefore, following prior work, we use Top-N (N=1, 3, 5) to denote the number of faults with at least one faulty element located within the first N positions, emphasizing earlier fault detection [69, 70, 88]. Note that even when multiple faulty elements of a fault is localized within Top-N, it is only counted once.

**Mean Average Rank (MAR):** For precise localization of all faulty elements of each fault, we compute the average ranking of all the faulty elements for each fault. MAR of each project is simply the mean of the average ranking of all its faults.

**Mean First Rank (MFR):** For a fault with multiple faulty elements, the localization of the first faulty element is critical since the rest faulty elements may be directly localized after that. Therefore, for each project, we use MFR to compute the mean of the first faulty element’s rank for each fault.

Following prior fault localization work on Defects4J [8, 29], we perform leave-one-out cross validation on the faults for each project. To illustrate, for a project with k faulty versions in total, we separate them into two groups: one faulty version as test data to predict its rank and other k-1 faulty versions (of the same project) as training data to build the ranking model. Please note each buggy version includes a large number of training instances (i.e., methods executed by failed tests), e.g., Closure itself already has 100,000+ training instances, much larger than the famous MNIST deep-learning dataset [89]. We also applied cross-validation/L2-Regularization/Dropout to prevent overfitting. Besides the above within-project scenario, we also perform cross-project learning-based fault localization in RQ4, i.e., for localizing one faulty version of a project, all faulty versions from other five projects are used as training data. Our experiments were conducted on a Dell machine with Intel Xeon CPU E5-2660 v4@2.00GHz and 220G RAM, running Ubuntu 14.04. Our data/script are publicly available [90].

### 5 RESULT ANALYSIS

**RQ1: DeepFL Effectiveness.** To answer this RQ, we compare the effectiveness of DeepFL (with its default MLP_{DEFL} (2) model and default setting) with state-of-the-art fault localization techniques, including MULTRIC [9], FLUCCS [31], and TraPT [29]. We also compare DeepFL with the most effective traditional SBFL and MBFL techniques, i.e., Ochiai [20] and Metallaxis [24, 26] with the Ochiai formula. Note that the original FLUCCS uses software age and change information which is not always available and cannot be applied to all Defects4J subjects, so we simplify it to integrate only its complexity metrics and spectrum-based suspiciousness values after method-level aggregation via LIBSVM. Table 2 presents the detailed experimental results. In the table, Column 1 lists all the studied Defects4J subjects; Column 2 lists all the compared techniques; the remaining columns present the main measurements used in this work. Surprisingly, DeepFL is able to localize 213 faults within Top-1, i.e., 57/53 more Top-1 faults than TraPT and FLUCCS. Also, the MFR and MAR values of DeepFL are the best among all studied techniques. We think the reason to be that MLP_{DEFL} (2) considers the hierarchical connections between different feature groups, providing more effective fault-diagnosis information analysis.

To investigate whether the differences between DeepFL with other state-of-the-art techniques are statistically significant, we further perform Wilcoxon signed-rank test [91] with Bonferroni corrections [92]. The results show that DeepFL is significantly better than all existing techniques in terms of buggy-method rankings at significance level of 0.05 (with p-values from 1.74e-28 to 4.63e-7 after corrections and effect-size from 0.14 to 0.33). The reason for small effect-size is that bug-ranking has huge standard-deviations across all bugs for all fault localization techniques (e.g., from 1st
Table 3: Effectiveness of different deep learning models

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Top-1</th>
<th>Top-3</th>
<th>Top-5</th>
<th>MFR</th>
<th>MAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBSVM</td>
<td>183</td>
<td>268</td>
<td>299</td>
<td>9.28</td>
<td>11.83</td>
</tr>
<tr>
<td>MLP</td>
<td>174</td>
<td>244</td>
<td>269</td>
<td>23.94</td>
<td>28.13</td>
</tr>
<tr>
<td>MLP2</td>
<td>169</td>
<td>262</td>
<td>290</td>
<td>8.40</td>
<td>10.31</td>
</tr>
<tr>
<td>BiRNN</td>
<td>192</td>
<td>277</td>
<td>310</td>
<td>7.23</td>
<td>9.52</td>
</tr>
<tr>
<td>MLP$_{DFL}(1)$</td>
<td>202</td>
<td>280</td>
<td>309</td>
<td>6.36</td>
<td>8.09</td>
</tr>
<tr>
<td>MLP$_{DFL}(2)$</td>
<td>213</td>
<td>282</td>
<td>305</td>
<td>6.63</td>
<td>8.27</td>
</tr>
</tbody>
</table>

Table 4: Impacts of different dimensions of features

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Top-1</th>
<th>Top-3</th>
<th>Top-5</th>
<th>MFR</th>
<th>MAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP$_{DFL}(1)$</td>
<td>202</td>
<td>280</td>
<td>309</td>
<td>6.36</td>
<td>8.09</td>
</tr>
<tr>
<td>MLP$_{DFL}(1)$-SpectrumInfor</td>
<td>197</td>
<td>269</td>
<td>304</td>
<td>7.06</td>
<td>9.12</td>
</tr>
<tr>
<td>MLP$_{DFL}(1)$-MutationInfor</td>
<td>166</td>
<td>245</td>
<td>276</td>
<td>14.91</td>
<td>17.89</td>
</tr>
<tr>
<td>MLP$_{DFL}(1)$-MetricsInfor</td>
<td>177</td>
<td>275</td>
<td>305</td>
<td>6.98</td>
<td>9.35</td>
</tr>
<tr>
<td>MLP$_{DFL}(1)$-TextualInfo</td>
<td>198</td>
<td>275</td>
<td>308</td>
<td>7.02</td>
<td>9.09</td>
</tr>
</tbody>
</table>

DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization

RQ2: Impacts of Epochs and Loss Functions. Shown in Section 4.1, our neural network models directly use widely used hyperparameters. However, the best training epoch number can be different for different problems/data. In addition, different loss functions can also affect the final results. Therefore, we further study the impact of training epoch number (i.e., less than 60) and two different loss functions on DeepFL in this RQ. Note that we do not show the impacts of learning rates since our results show they only affect convergence rate but not effectiveness. In Figure 8, each sub-plot presents the trend of one specific measurement value when using different number of epochs and loss functions; lines with different colors and point types present two different loss functions, e.g., softmax and pairwise. From the figure, we can clearly find that softmax is much better than pairwise after around 10 epochs in terms of all measurement values. We also find that the loss functions softmax and pairwise show very different trends. With the increasing epoch number, DeepFL with softmax loss function achieves better results and it has a slightly slow learning curve, e.g., the results are very close after around 10 epochs in terms of Top-3/5. However, the results of DeepFL with pairwise loss function are almost opposite compared with softmax. The best result appears in around 5 epochs and other results become much worse with increasing epoch numbers. These findings actually show that pairwise loss function can converge to a maximum within a small number of epochs, and may incur overfitting problem with increasing training epochs. In this RQ, softmax loss function needs more training time to achieve better results and its stable learning curve shows that it does not suffer from the overfitting problem. One potential reason could be...
We also find that the within-project prediction performs better. We also observe that the best Top-N result (i.e., 207 Top-1 faults) also significantly outperform the existing techniques, e.g., localizing 50+ more faults within Top-1 than the most effective existing technique. The experimental results also show that the default DeepFL with tailored MLP (DFL) is more effective than the traditional Learning-to-Rank algorithm using the same features.

### Threats to Validity

The main threat to *internal* validity is the potential mistake in our feature collection and technique implementation. To reduce this threat, we collect features and implement our techniques by utilizing state-of-the-art tools and frameworks, such as ASM, PIT, Jhaw, Indri and TensorFlow. The main threat to *external* validity mainly lies in the selection of the studied subjects. To reduce this threat, we plan to evaluate on more real-world projects. Also, the possible flaky tests in Defects4J may impact our results. Furthermore, due to the randomness of neural network models, the result might be different in different runs. To reduce this potential threat, we rerun DeepFL 10 times and observe a range from 209 to 216 in term of Top-1 value, showing the stability of our model/implementation. The main threat to *construct* validity is that the measurements used may not fully reflect real-world situations. To reduce this threat, we use Top-N, MAR and MFR metrics, which have been widely used in previous studies [9, 29, 31, 79].

### 6 CONCLUSION

In this paper, we propose DeepFL, a deep learning approach for localizing faults based on various feature dimensions. The experimental study on 395 real bugs from the widely used Defects4J benchmark shows that DeepFL can significantly outperform existing state-of-the-art learning-based fault localization techniques, e.g., localizing 50+ more faults within Top-1 than the most effective existing technique. The experimental results also show that the default DeepFL with tailored MLP (DFL) is more effective than the traditional Learning-to-Rank algorithm using the same features, and can also be much faster (e.g., up to 1200X faster) for prediction. In addition, further studies on the impacts of different feature dimensions reveal that all the feature dimensions studied in our paper are useful. Furthermore, the promising result in the cross-project scenario provides a practical guide for real-world debugging.

### ACKNOWLEDGMENTS

This work was partially supported by National Science Foundation under Grant Nos. CCF-1566589 and CCF-1763906, NVIDIA, Amazon, Shenzhen Peacock Plan (Grant No. KQTD2016112514355531), and Science and Technology Innovation Committee Foundation of Shenzhen (Grant No.JCYJ2017081710848086). The authors also thank Zehuai Wang for the help with the artifact preparation.
REFERENCES


[34] DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization ISSTA ’19, July 15–19, 2019, Beijing, China.


