Information and Software Technology 131 (2021) 106486

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A study of effectiveness of deep learning in locating real faults

Zhuo Zhang [a,c], Yan Lei [*,b], Xiaoguang Mao [c], Meng Yan [b], Ling Xu [b], Xiaohong Zhang [b]

[a] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541000, China
[b] School of Big Data & Software Engineering, Chongqing University, Chongqing 400044, China
[c] College of Computer, National University of Defense Technology, Changsha 410073, China

ARTICLE INFO

Keywords:
Fault localization
Debugging
Neural networks
Deep learning
Suspiciousness

ABSTRACT

Context: The recent progress of deep learning has shown its promising learning ability in making sense of data, and many fields have utilized this learning ability to learn an effective model, successfully solving their problems. Fault localization has explored and used deep learning to server an aid in debugging, showing the promising results on fault localization. However, as far as we know, there is no detailed studies on evaluating the benefits of using deep learning for locating real faults present in programs. **Objective:** To understand the benefits of deep learning in locating real faults, this paper explores more about deep learning by studying the effectiveness of fault localization using deep learning for a set of real bugs reported in the widely used programs. **Method:** We use three representative deep learning architectures (*i.e.* convolutional neural network, recurrent neural network and multi-layer perceptron) for fault localization, and conduct large-scale experiments on 8 real-world programs equipped with all real faults to evaluate their effectiveness on fault localization. **Results:** We observe that the localization effectiveness varies considerably among three neural networks in the context of real faults. Specifically, convolutional neural network performs the best in locating real faults, showing an average of 38.97% and 26.22% saving over multi-layer perceptron and recurrent neural network respectively; recurrent neural network and multi-layer perceptron yield comparable effectiveness even if the effectiveness of recurrent neural network is marginally higher than multi-layer perceptron. **Conclusion:** In context of real faults, convolutional neural network is the most effective for fault localization among the investigated architectures, and we suggest potential factors of deep learning for improving fault localization.

1. Introduction

In the process of software development and maintenance, debugging is to locate and fix faults within a program. It is a painstaking task because it usually requires much manual involvement of debugging engineers (*e.g.* inserting print statements, and setting break points). In fact, debugging has been identified as one of the most expensive and time-consuming processes for software developers [1,2]. To reduce the cost, many fault localization techniques have been proposed to provide automated assistance in locating the faults that cause executions to produce incorrect outputs (*i.e.* failures) [3–7].

In recent years, deep learning has progressed rapidly and became increasingly popular in various applications (*e.g.* image classification, object detection, and segmentation) due to its promising ability of providing tremendous improvement in robustness and accuracy [8]. It implies that deep learning may open a new perspective for fault localization by using its learning ability of data to construct a localization model, evaluating which statements being suspiciously faulty. In fact, some researchers have explored the use of deep learning to discuss and evaluate the potential of deep learning in fault localization [9,10]. They found that with the capability of estimating complicated functions by learning a deep nonlinear network structure and attaining distributed representation of input data, deep learning exhibits strong learning ability from sample data sets. This learning ability is beneficial for fault localization, showing better localization results over the state-of-the-art fault localization techniques (*e.g.* BP neural networks [11], PPDG [12], Tarantula [13], Dstar [14], Barinel [15], and Ochia [16]).

Although deep learning has shown its promising results on fault localization, the existing analysis still needs much further study. For example, the existing analysis [9,10] mostly utilizes those subject programs with seeded faults. The recent research [2] has revealed that artificial faults including seeded ones are not adequate for evaluating

* Corresponding author.
E-mail addresses: zz8477@126.com (Z. Zhang), yanlei@cqu.edu.cn (Y. Lei), xgmao@nudt.edu.cn (X. Mao), mengy@cqu.edu.cn (M. Yan), xuling@cqu.edu.cn (L. Xu), xhongz@cqu.edu.cn (X. Zhang).

https://doi.org/10.1016/j.infsof.2020.106486
Received 29 October 2019; Received in revised form 11 September 2020; Accepted 5 November 2020
Available online 15 November 2020

fault localization techniques, and recommends using real faults for evaluation. Furthermore, deep learning has different learning architectures, which may demonstrate different abilities in terms of fault localization effectiveness. It is necessary to identify the varying degree of the representative deep learning architectures on fault localization and recommend promising learning architectures for debugging engineers. It is also vital to investigate what factors are important for a deep learning architecture to take the advantage on improving fault localization.

Therefore, this paper explores more about deep learning in improving fault localization, *i.e.*, we aim at evaluating and understanding the benefits of using the representative deep learning architectures on fault localization in real faults. To achieve this goal, we face three fundamental problems. The first one is that what deep learning architectures are used as representative ones for the study. The solution to this problem is to use the basic and popular deep learning architectures as the representative ones. In deep learning, convolutional neural network (CNN) [17] recurrent neural network (RNN) [18,19] and multi-layer perceptron (MLP) [20] are the three basic and popular learning architectures, and more importantly, the three learning architectures have different distinct features probably benefiting fault localization. Among the three learning architectures, MLP has the simplest yet effective structure, and can be easily deployed by fault localization; CNN is capable of processing the rapid expansion of the number of parameters to be trained due to using local connections, parameter sharing, and down-sampling in pooling, and may be useful for enhancing the generation ability of the fault localization model; RNN is capable of learning long-term dependencies, and may be useful for analyzing how a fault depends on other statements to cause a program failure. Thus, we use the three representative and distinct learning architectures for the study to understand what factors of a learning architecture are important for effective fault localization. The second one is how to use the methodology of different deep learning architectures for fault localization. The solution is to utilize the methodology of deep learning from the problem domain of fault localization, that is, we should construct training samples and fix the labels for deep learning in the perspective of fault localization. In the literature of fault localization, there is an effective and widely used information model [14], using coverage information and test results, reflects the execution information of a statement (involved or not involved) and the test results (*i.e.* failed or not failed). This information model has a well-define structure, including coverage information and test results, which can be potentially used as training samples and the labels for deep learning in the perspective of fault localization. It means that we may utilize this effective information model to successfully apply the methodology of deep learning from the problem domain of fault localization to output a fault localization solution. The third one is how to identify a set of representative subject programs with all real faults for the study. The solution is to find those large-sized programs with all real faults widely used in the literature.

Based on the above analysis, we use convolutional neural network (CNN) [17], recurrent neural network (RNN) [18,19], and multi-layer perceptron (MLP) [20] for studying the benefits of using deep learning on fault localization. With the effective and widely used information model [14], we leverage its structure of coverage information and test results to construct training samples and fix the labels, for successfully applying the methodology of deep learning from problem domain of fault localization. Therefore, with the above solution, we study three fault localization approaches based on deep learning, namely CNN-FL, RNN-FL, and MLP-FL representing fault localization using convolutional neural network, recurrent neural network and multi-layer perceptron respectively. Among the three approaches, we propose CNN-FL and RNN-FL while Zheng et al. [9] proposes MLP-FL which complies with the structure of the other two approaches. The three approaches using deep learning train their networks with test cases, and finally evaluate the suspiciousness of a statement being faulty by testing the trained model with a virtual test suite.

Based on our earlier work [10], we collect 8 programs from the widely used benchmarks in the field of software debugging. Thus, we conduct a large-scale empirical study on the 8 real-life programs with all real faults to investigate the varying levels among different learning architectures, pinpoint the advantage of which architecture over other ones, and discuss the factors behind this advantage. The results show that CNN-FL performs best among the representative deep learning architectures, and reducing the number of parameters implies a key factor of a deep learning architecture in taking advantage over other architectures for improving fault localization.

The main contributions of this paper can be summarized as:

- We present a methodology for using deep learning to locate faults, and based on the methodology we propose CNN-FL and RNN-FL.
- We conduct a large-scale empirical study to evaluate the effectiveness of three deep learning approaches across various representative programs in the context of locating real faults, identifying the varying localization levels of the representative learning architectures and showing CNN-FL is the most effective one among the three deep learning approaches.
- We further demonstrate the potential of using deep learning as a new perspective and discuss the key factors of deep learning for effective fault localization.

The structure of the rest paper is organized as follows. Section 2 provides the necessary background. Section 3 describes the three fault localization approach based on deep learning. Section 4 presents results of our empirical study including experiment subjects, experiment design, data analysis, discussion, and threats to validity. Section 5 introduces related work and Section 6 concludes.

2. Background

In this section, we will briefly introduce the three basic and popular deep learning architectures, namely multi-Layer perceptron (MLP), convolutional neural network (CNN), and recurrent neural network (RNN). Hinton et al. [20] proposed multi-layer perceptron (MLP), which is an artificial deep neural network with multiple hidden layers, and each node at the same hidden layer uses the same nonlinear function to map the feature input from the layer below. The structure of an MLP is flexible and demonstrates promising capacity to fit the highly complex nonlinear relationship between inputs and outputs. It is one of the deep learning models and has been successfully applied in many areas of software engineering [8]. And also it has already been rapidly evolved into making sense of data such as images, text, and sound [8,21]. MLPs have the capability of estimating complicated functions by learning a deep nonlinear network structure and further obtaining distributed representation of input data. It exhibits strong capability in learning representation from sample data.

A convolutional neural network (CNN) [17] is a specialized kind of neural network for processing data that has a known, grid-like technology. It could learn rich highly abstract data features to represent complex objects effectively [17,22–26]. CNNs have continued to grow in popularity and are originally designed to solve problems such as computer vision related tasks, but are not limited to images. They can also be used for audio data, and text data. A CNN consists of five main components: input layer, convolution layers, pooling layers, fully connected layers, and output layer. These five components help match complex patterns by highlighting important information while ignoring noise. In practice, the first layer is the input layer with input data sets and the second layer is the convolution layer. Technically, A CNN has at least one layer that does a convolution with its configurable kernels, meaning that the goal of a convolution operation is to utilize a filter (also known as convolution kernel), and slide the filter over an input tensor's each area that has the same size as the filter. Activation functions such as

rectified linear unit are utilized to support convolution layer and are the source of non-linearity of the mapping function represented by the CNN. Usually, followed by a convolution layer, it is a pooling layer. The pooling layer reduces the size of the input, and thus boosts training speed and reduces the possibility of over-fitting. It keeps important information for the next layer while scaling down input. Pooling layers execute much faster than convolution layers, though the latter could also reduce the size of input. After several groups of convolution layer and pooling layer, there are several fully connected layers. These layers are often in the format of multiplying layer input with weight, and then adding bias where layer input, weight and bias are all tensors. These short-hand layers will do the same thing while taking care of the intricacies involved in managing the weight and bias tensors. With their flexible structure and multiple hidden layers, CNNs have the capacity to fit highly complex nonlinear relationship between inputs and outputs. The last layer is the output layer that outputs the results. By utilizing the difference between the output and the actual target output to construct a cost function, we can then train the CNN by back propagation (BP) algorithm. The Back propagation algorithm is for updating the model parameters, by minimizing the loss between network's output and target output through a number of training steps. With that goal, applying gradient descent to find the minimum of the loss function will result in the model learning from the input data set. Generally speaking, CNNs are useful network architectures in multiple industries from audio to media and so on.

A recurrent neural network (RNN), another important branch of deep neural networks family, is a family of networks that are good at solving sequential tasks in many domains such as speech recognition, speech synthesis, connected handwriting recognition, time-series forecasting, image caption generation, and end-to-end machine translation. RNNs build on the same neurons summing up weighted inputs from other neurons. The neurons of an RNN are allowed to connect both forward to higher layers and backward to lower layers. Various variants of RNNs were not widely used until recently since insufficient computation power and difficulties in training. We have seen powerful applications of RNNs since the invention of architectures like LSTM [18,19]. LSTM, proposed by Hochreiter and Schmidhuber [18], is a special form of RNNs which is designed to overcome vanishing and exploding gradients problem. It performs significantly better for long-term dependencies problems and becomes a representative for RNNs. In LSTM architecture, normal neurons in an RNN are replaced by so-called LSTM cells that have a little memory inside to cope with the problem of vanishing and exploding gradients. These cells are wired together as they are in a usual RNN but they own internal states that help to remember errors over many time steps. The trick of these internal states lies in the fact that they have a self-connection with a fixed weight of one and a linear activation function, so that its local derivatives is always one. During back propagation, this so called error carousel can carry errors over many time steps without having the gradient vanish or explode. LSTM is directional, and only uses past contexts. However, in many tasks, information from both directions are useful and complementary to each other. Therefore, bidirectional LSTM [27] was proposed by combining two LSTMs, one forward and one backward. This architecture allows higher level of abstractions and has achieved significant performance improvements in the task of speech recognition. Considering the popularity and the performance, we use bidirectional LSTM in the study.

3. Deep-learning-based fault localization

This section will present the methodology of deep-learning-based fault localization using three deep learning architectures in detail. Furthermore, we will provide a concrete example for a demonstration of how to apply deep learning for localizing faults.

3.1. Methodology

We will depict the overview methodology for using different neural networks to locate faults. Deep learning learns rich highly abstract data features from the input data set and its learning power should be useful to evaluate which statements exhibiting more relevance with failures. Thus, the basic idea of deep-learning-based fault localization is to adopt a specific neural network to build a model for learning and estimating the association of a statement with failures. We use the estimation of the association as the suspiciousness of the statements of being faulty. For leveraging the learning power of a specific neural network, we should first initiate the neural network for fault localization, *i.e.* , we should first construct training samples and fix their labels in context of fault localization. The process of fault localization starts from a failure, and focuses on which statement to be the fault causing a failure. It means that we should initiate the neural network with training samples and their labels, which can reflect the association of the execution information of a statement with failures.

For realizing such initiation, we adopt an effective and widely-used information model [14] in the fault localization literature by using coverage information and test results. Specifically, given a program P with N statements, it is executed by a test suite T with M test cases, which contain at least one failed test case (see Fig. 1). The element x_{ij}=1 means that the statement j is executed by the test case i, and x_{ij}=0 otherwise. The $M\times N$ matrix records the execution information of each statement in the test cases T. The error vector e represents the test results. The element e_i equals to 1 if the test case i failed, and 0 otherwise. The error vector shows the test results of each test case (*i.e.* failures or non-failures). Since they can associate the execution information of a statement (*i.e.* the matrix) with failures or non-failures (*i.e.* the error vector), the matrix and the error vector are the training samples and their corresponding labels respectively, where each training sample is an N-dimensional vector. Based on the training samples and their labels, we use mini-batch stochastic gradient descent to update network parameters with the batch size settled to h, namely, each time we feed a $h\times N$ matrix as input to the network and use its corresponding error vector as labels. The network is trained iteratively.

The training process will learn a trained model, reflecting the complex nonlinear relationship between the statement coverage and test results. Finally, we construct a set of virtual test cases (see Fig. 2) as the testing input to measure the association of each statement with test results. Concretely, each time we choose one virtual test case and input it to the network, the output is the estimation of the probability of causing a failure by executing the virtual test case. Furthermore, suppose that if the virtual test case only covers one statement, the output is also the estimation of the probability of causing a failure by executing the statement. The estimation can show the suspiciousness of a statement of being faulty.

Thus, as shown in Fig. 2, we construct N virtual test cases, equaling to the number of the statements, where each virtual test case only covers one statement. Specifically, the element x_i =1 means that the statement i is only covered by the virtual test case t_i, and x_i =0 in the other virtual test cases. When the coverage vector of a virtual test case is inputted to the trained neural network, the output of the network is the estimation of the virtual test case's execution result of being a failure by covering only one statement. The value of the result is between 0 and 1. The larger the value is, the more likely it is that the statement only covered by the

$$\begin{array}{c} \\ M\ test\ cases \end{array} \begin{array}{c} \begin{array}{cccc} N\ statements & & & \end{array} \\ \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} \end{array} \begin{array}{c} errors \\ \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \end{array}$$

Fig. 1. The coverage and results of M executions.

$$
\begin{array}{c} \\ t_1 \\ t_2 \\ \vdots \\ t_N \end{array}
\begin{array}{c}
\overset{N\ dimensional}{\overbrace{\begin{array}{cccc} x_1 & x_2 & \cdots & x_N \end{array}}} \\
\begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}
\end{array}
$$

Fig. 2. Virtual test cases.

coverage vector is the buggy statement. For example, the statement i is likely to be the buggy statement. Then we input t_i to the trained neural network, and the output of virtual test case t_i represents the probability of test case execution result of being a failure by only covering the statement i. The value of the result is the suspiciousness of the statement i. In this way, we can evaluate the suspiciousness of each statement being faulty.

Although different neural networks use the same overview methodology, their specific learning processes are different due to their different structures. The following subsections will depict how each of the three neural networks apply the overview methodology for fault localization.

3.2. CNN-FL

Architecture Based on the overview methodology at Section 3.1, CNN-FL uses CNN to learn a fault localization model evaluating the suspiciousness of each statement being faulty. As shown in Fig. 3, the architecture of CNN-FL consists of one input layer, two convolution layers, two rectified linear units, two pooling layers, several fully connected layers, and one output layer.

In the input layer, based on the matrix $M\times N$ and the error vector (see Fig. 1), h rows of the $M\times N$ matrix and the error vector are used as an input, which are the coverage information of h test cases and their execution results starting from the ith row, where $i \in \{1,1+h,1+2h,...,1+(\lfloor M/h \rfloor\text{-}1)\times h\}$. As a reminder, if M/h is not an integer, we will also input the execution information of the remainder $(M - \lfloor M/h \rfloor .*h)$ test cases and their execution results starting from the $(\lfloor M/h \rfloor .*h + 1)$th row. When the size of the program is large, the elements of each row are numerous, leading to a large number of parameters related to the input data vector. Thus, it is dispensable to reduce the number of parameters while preserving the significant features. That is the function of convolution layers.

The convolution layer often has several convolution kernels (filters), leveraging certain patterns to highlight the features in the input data, and thus recognizing the important attributes of the input. The convolution operations enable CNNs to accurately match diverse patterns. When an operation is sliding the kernel over each point in the input, it is utilizing the strides parameter to change how it walks over the input. The strides parameter configures the operation to skip less important elements and obviously reduces the dimensionality of the output, while the kernel allows the operation to use all the input values. The input becomes a smaller tensor after the convolution operation and consequently the dimensionality is reduced. This makes less processing power be required and will keep from creating receptive fields that completely overlap. CNNs generally consist of multiple convolution layers, where each convolution layer can have multiple different convolution kernels, and furthermore each convolution kernel corresponds to new values after filtering. In our model, there are two convolution layers, where each one has several convolution kernels, *i.e.* , convolution layer 1 has k_1 kernels while convolution layer 2 has k_2 kernels. In our model, we set k_1 and k_2 to be 32 and 64 respectively. Every kernel W_k is a vector, where$W^k \in R^{10}$ (*i.e.* the size of kernel W_k is 10 in our experiment.), which is used to slide over each element of the input coverage data vector and recognize the important attributes. Our model uses strides parameters to reduce the dimensionality of the output and skip less important elements. For example, we reduce those elements are all 0 in all the coverage data vectors, which means that those statements are not executed in all the test cases.

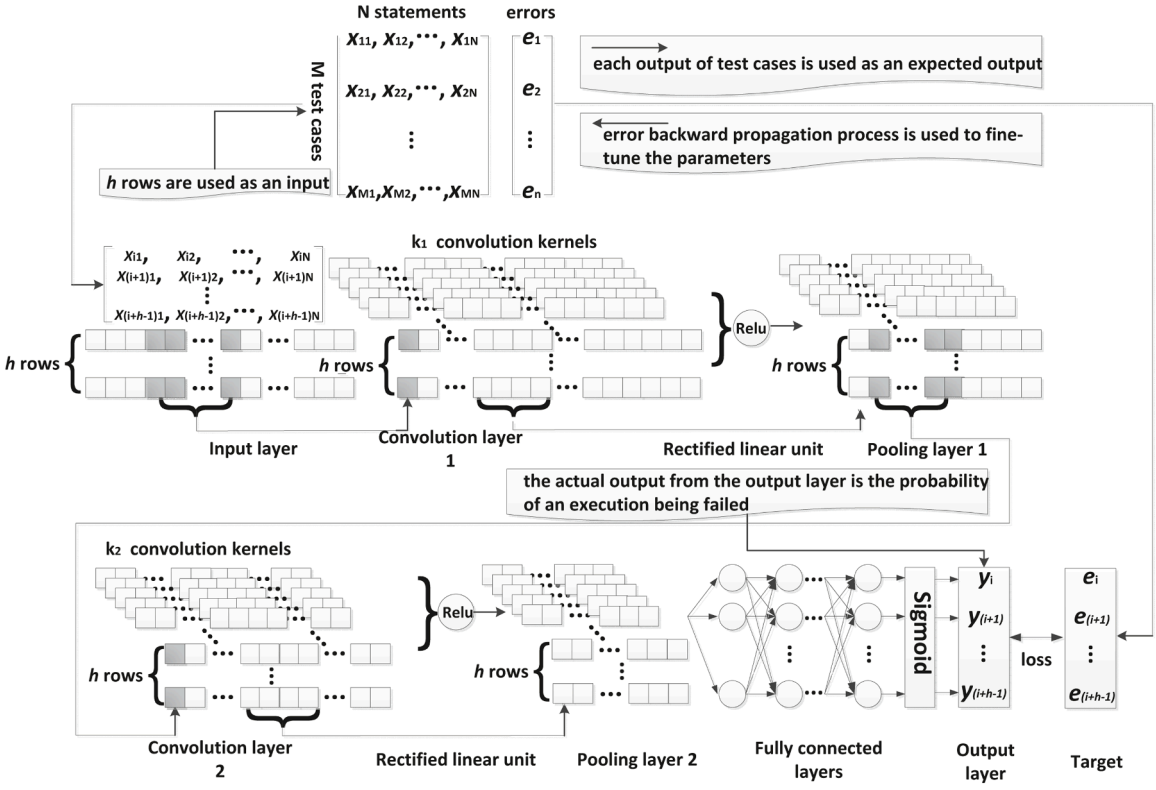


Fig. 3. The workflow of CNN-FL.

After each convolution layer, there is a rectified linear unit (*ReLU*) which is an activation function for supporting convolution layers. There are three types of activation functions often used [8]: *ReLU* function, *sigmoid* function, and hyperbolic tangent function (*tanh*). *ReLU* function empirically works well although it sacrifices information. It keeps the same input values for any positive numbers while setting all negative numbers to be 0. As shown in Fig. 4 and Eq. (1), x is a variable, *ReLU* has a range of [0, $+\infty$], and it has a benefit that it does not suffer from the gradient vanishing problem. The *sigmoid* function keeps a value between 0 and 1, which is useful in networks that train on probabilities in the range of [0, 1]. Larger values sent into a sigmoid function will trend closer to 1 just as illustrated in Fig. 4 and Eq. (2). The drawback of *sigmoid* is that when there is an input becoming saturated and the changes in the input become exaggerated, the reduced range of output values could cause gradient vanishing. As shown in Fig. 4 and Eq. (3), *tanh* function, also called hyperbolic tangent function, is a close relative to sigmoid function with the same benefits and drawbacks. *Tanh* function keeps a value between -1 and 1. It is used in the range of $[-1, 1]$, which is the main difference between *sigmoid* and *tanh*. And *tanh* can output negative values useful in certain networks. In our model, we use *ReLU* function as an activation function because it works well and does not suffer from gradient vanishing problem.

$$\mathrm{Re}LU(x) = \max(0, x) \tag{1}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3}$$

After rectified linear unit, it is the pooling layer. There are two main pooling methods. One is *Max_pool* and the other is *Avg_pool*. *Max_pool* picks the maximum value within a certain window size (kernel size) with striding over a tensor. The goal of *Max_pool* is to keep the largest value in the tensor. Just as shown in Fig. 5, the largest value of the tensor is 2.0. Hence, *Max_pool* will choose the largest value in each kernel size as it strides over the input from the last layer. Different from *Max_pool*, *Avg_pool* strides over the tensor and averages all the values found in a window size. The advantage of *Avg_pool* is that it could be useful when reducing values where the entire kernel is important such as an input tensor with a large width and height but small depth. In Fig. 5, *Avg_pool* returns the value of 0.67, which is the average of all the values within the tensor ((1.0+0.2+2.0+0.3+0.5+0.7+1.0+0.1+0.2)/9). In our model, we choose *Max_pool* because the empirical results show that the largest value in the tensor is beneficial for fault localization effectiveness.

The next are fully connected layers. CNN architecture arranges fully connected layers in a chain structure, with each layer being a function of the layer that preceded it. The relationship between the fully connected layers in a CNN model can be interpreted as follows:

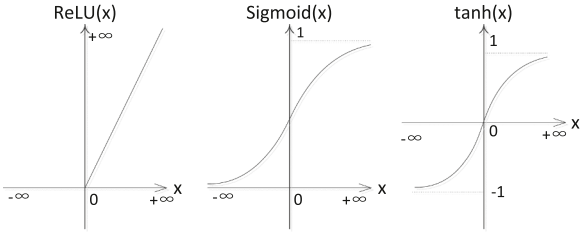


Fig. 4. *ReLU, sigmoid* and *tanh* function.

$$\begin{aligned} x &= output\ of\ pooling\ layer\ 2 \\ h_1 &= f(w_1 x + b_1) \\ h_2 &= f(w_2 h_1 + b_2) \\ &\vdots \\ h_n &= f(w_n h_{n-1} + b_n) \end{aligned} \tag{4}$$

Where, x is the output of pooling layer 2, and w_i, (b_i) and h_i denote the weight parameter, bias parameter and the output of each layer, respectively. According to the formulas in Eq. (4), we can see that the architecture of a CNN arranges these layers in a chain structure, with each layer being a function of the layer that preceded it. Though one hidden layer could also be sufficient to fit the training set, deeper network architectures with more layers are required to get better results. Nevertheless, those deeper network architectures have a drawback that they are often harder to optimize. Experimentation guided by monitoring the validation set error ought to be found to get an ideal network architecture. By utilizing the difference between the output by the network (y_i) and the actual target output (e_i) to construct a cost function, we can then train the layers by using back propagation (BP) algorithm. During the training process, the weight (w_i) and bias (b_i) parameter are determined to choose the width of the hidden layers and the depth of the network. In our model, there are 3 fully connected layers with the number of nodes in each one to be 1024.

The last layer is output layer. The output function in the output layer is *sigmoid* function (see Eq. (3)) We use *sigmoid* function to output the result in our model because values sent into a *sigmoid* function will be 0 to 1 as illustrated in Fig. 4. Each element in the result vector of the *sigmoid* function has difference with the corresponding element of the target vector. The difference is the loss between the output layer and target.

We use back propagation algorithm [8] to fine-tune the parameters of the model because it is commonly used to adjust the weight of neurons by calculating the gradient of the loss function in the context of deep learning. The goal is to minimize the loss between training result y and errors-vector e. The algorithm goes forward from the input layer calculating the outputs of each layer up to the output layer. Then it starts calculating derivatives going backwards through the layers and propagating the results in order to do less calculation by reusing all of the elements already calculated. The learning rate impacts the speed of convergence. Our model adopts dynamic adjusting learning rate for its two merits. One is that it can make large changes at the beginning of the training procedure when larger learning rate values are used. The other one is that it can decrease the learning rate with a smaller training updates for computing weights later, resulting in accurate weights more quickly. The experiment results conform to the effectiveness of dynamic adjusting learning rate. In the following equation (Eq. (5)), one *Epoch* means completing all training data once, *LR* represents learning rate, *DropRate* is the amount that learning rate is modified each time while *EpochDrop* is how often to change the learning rate. We set the initial learning rate to be 0.01 and *DropRate* to be 0.98. *EpochDrop* is set according to the number of the test cases.

$$LR = LR{*}DropRate^{(Epoch+1)/EpochDrop} \tag{5}$$

Summary of steps Finally, we summarize the steps of CNN-FL as follows:

(1) Construct a CNN that consists of one input layer according to the number of statements, two convolution layers, two pooling layers, two rectified linear units, several fully connected layers with the number of nodes in each one estimated by the program size and one output layer with the number of nodes to be 1. We choose *ReLU* function to be the activation function due to its good performance and the advantage of not suffering from gradient vanishing.

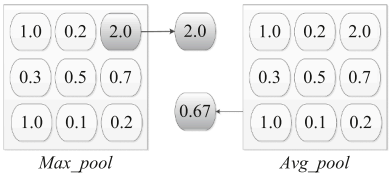


Fig. 5. Example of *Max_pool* and *Avg_pool*.

(2) Train the network using the coverage data and execution result of the test case according to Fig. 1as training sample set. CNN-FL inputs each coverage data matrix one by one into the CNN model to obtain the complex nonlinear mapping relationship between the coverage data and the execution result. In each coverage data matrix, there are h rows and N columns, we set batch size h to be 10 and N is the number of statements.
(3) Input each coverage data vector of virtual test cases according to Fig. 2 into the trained CNN and obtain the outputs. The output of t_i reflects the probability that x_i contains a bug, showing the suspiciousness of the statement i of being faulty. The larger the value of output of t_i is, the more likely it is that the statement i contains the bug.
(4) Rank the statements in descending order of the suspiciousness calculated by the previous step. This step outputs localization results in a text form. The text form contains the statements in descending order of suspiciousness given by our model. This form provides potentially suspicious statements (*e.g.* suspiciousness, and line number), which can assist developers or automated program repair tools in locating and fixing faults.

3.3. RNN-FL

Architecture RNN-FL utilizes RNN to conduct the training and learn a localization model via the overview methodology. Fig. 6 shows the architecture of RNN-FL. As shown in Fig. 6, the architecture of RNN-FL consists of one input layer, two recurrent layers, and one output layer. Each recurrent layer contains one forward (left to right) LSTM and one backward (right to left) LSTM, Specifically, given a program P with N statements, it is executed by M test cases which contains at least one failed test case (see Fig. 1). Based on the matrix $M \times N$and its errors-vector, we use mini-batch stochastic gradient descent to update

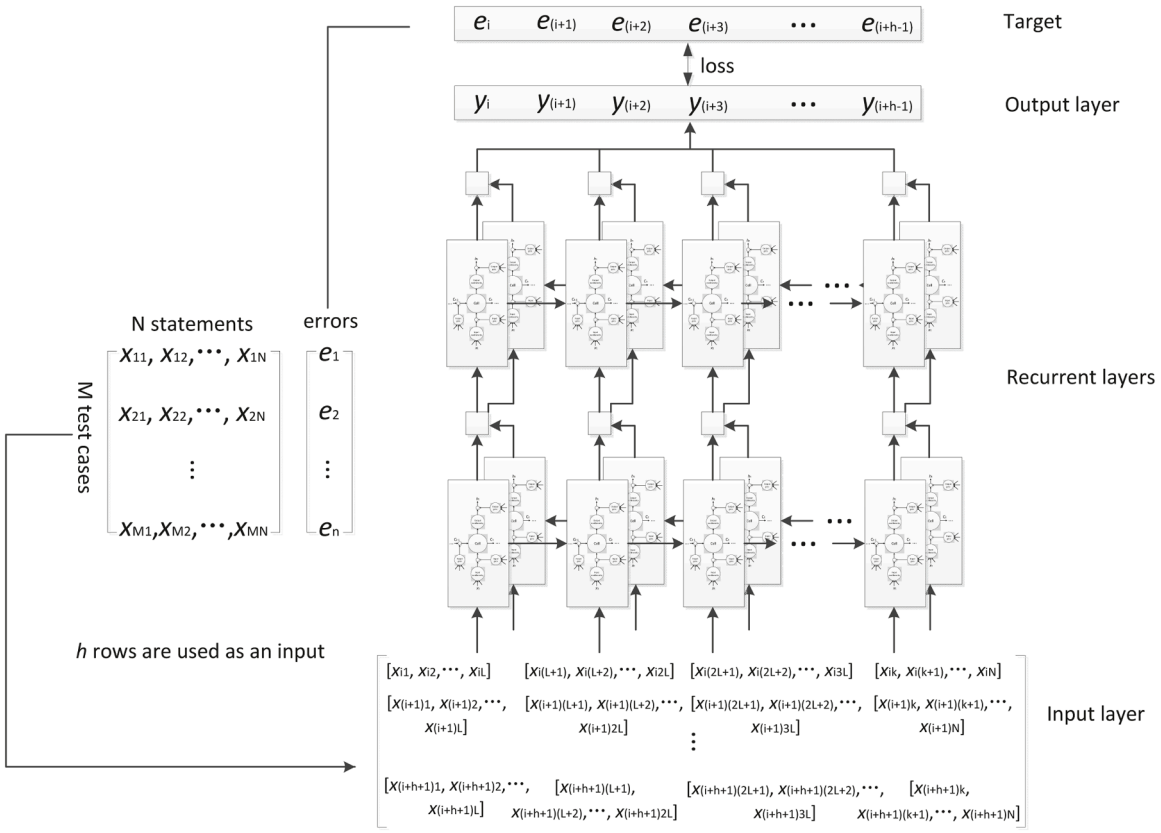


Fig. 6. The architecture of RNN-based fault localization.

network parameters with the batch size settled to h. Thus, similar to CNN-FL, h rows of the matrix $M \times N$and its errors-vector are used as an input, which are the coverage information of h test cases and their execution results starting from the ith row, (i $\in$ {1,1+h,1+2h,...,1+($\lfloor M/h \rfloor$-1) $\times$ h}). In each row, there are N statements. For the specific feature of RNN in capturing long-range dependencies, RNN requires cut the N statements into different groups. Specifically, if N/L is an integer, these N statements are cut into N/L groups, where each group has L statements; otherwise, the N statements are categorized into $\lfloor N/L \rfloor+1$ groups, where each group in the first $\lfloor N/L \rfloor$ groups has L statements while the last group has N-$\lfloor N/L \rfloor \times L$ statements. As a reminder, $\lfloor N/L \rfloor$ means the integer part of N/L.

The next are recurrent layers, which consist of several LSTM units. Fig. 7 illustrates the architecture of an LSTM Unit. An LSTM unit consists of a memory cell and three multiplicative gates, namely input gate, forget gate and output gate. Conceptually, the memory cell stores the past information. The input gate and the output gate allow the cell to store information for a long period of time. At the same time, the memory in the cell could be cleared by the forget gate. This architecture of LSTM allows it to capture long-range dependencies, which often occur in fault localization tasks. Each time one unit receives the information of one group that has L statements or N-$\lfloor N/L \rfloor \times L$ statements as input, and it updates its internal state with both input and past state, which could capture past contexts and make prediction. LSTM is directional, it only uses past information. However, in the statements sequences of a program, information from both directions are useful. Therefore, we combine two LSTMs, one forward and one backward, into a bidirectional LSTM. Furthermore, multiple directional LSTMs can be stacked, resulting in a deep structure as illustrated in Fig. 6. We utilize bidirectional LSTM for its ability of higher level of abstractions and the achievement of significant performance improvement. The complex nonlinear relationship between the coverage information of the statements and the results of test cases can be reflected after training the network

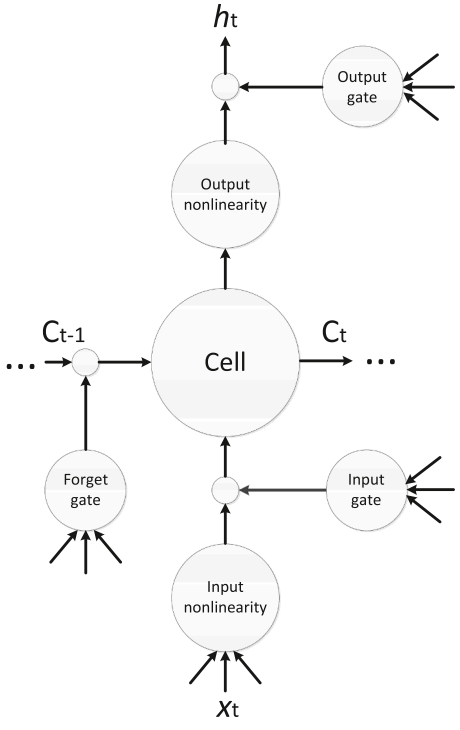


Fig. 7. The architecture of an LSTM unit.

iteratively. At last, a set of virtual test cases (see Fig. 2) that is an N-dimensional unit matrix is constructed. Each virtual test case is used as a testing input, and their outputs are the suspiciousness of the corresponding statements. Our RNN model adopts dynamic adjusting

learning rate as depicted in Eq. 5.***Summary of steps*** Finally, we summarize the steps of RNN-FL as follows:

(1) Construct a RNN consists of one input layer according to the number of statements and cut the statements into N/Lor $\lfloor N/L \rfloor+1$ groups, two recurrent layers and one output layer with the number of nodes according to the number of test cases. In recurrent layers, we combine two LSTMs, one forward and one backward, into a bidirectional LSTM. We stack two bidirectional LSTMs, resulting in a deep structure to get higher level of abstractions and achieve significant performance.
(2) Train the network using the coverage data and execution result of the test cases according to Fig. 1as the training sample set. The model inputs each coverage data matrix one by one into the RNN network to obtain the complex nonlinear mapping relationship between the coverage data and the execution result.
(3) Input each coverage data vector of virtual test cases according to Fig. 2 into the trained RNN and obtain the outputs. The output of t_i reflects the probability that x_i contains a bug, showing the suspiciousness of the statement i being faulty. The larger the value of output of t_i is, the more likely it is that the statement i contains the bug.
(4) Rank the statements in descending order of the suspiciousness calculated by the previous step. This step outputs localization results in a text form. The text form contains the statements in descending order of suspiciousness evaluated by the RNN model.

3.4. MLP-FL

MLP-FL, proposed by Zheng et al. [9], adopts MLP to evaluate the suspiciousness of a statement being faulty, and complies with the overview methodology. MLP-FL first constructs a MLP with three parts: an input layer, hidden layers and an output layer. Then, it trains this model by utilizing statement coverage and test results as the input. Finally, MLP-FL tests the trained model using virtual test suite to evaluate the suspiciousness of each statement being faulty.

Specifically, based on the matrix $M\times N$ and its errors-vector as input (see Fig. 1), MLP-FL uses mini-batch stochastic gradient descent to update network parameters with the batch size settled to h, the network is trained by feeding a $h \times N$ matrix as input iteratively. Then, a set of virtual test cases (see Fig. 2), an N-dimensional unit matrix, is constructed and each virtual test case is used as a testing input. Finally, the output of each testing input is the suspiciousness of their corresponding statements.

Fig. 8 shows the architecture of MLP-FL. In MLP-FL, there are one input layer with the number of nodes according to the number of the statements, one output layer with the number of nodes according to the size of the test cases, appropriate number of hidden layers with the number of nodes in each one estimating by the following formula:

$$number = round\left(\frac{n}{30}+1\right)*10 \tag{6}$$

Where, n represents the number of executable statements. The hidden layers extract features from the input layer. Transfer function reflects the complex relationship between input and output. MLP-FL uses *ReLU* function (see Fig. 4 and Eq. (1)) in the hidden layer as transfer function and *sigmoid* function (see Fig. 4 and Eq. (2)) to output the result. MLP-FL adopts dynamic adjusting learning rate as illustrated in Eq. (5), and adjusts impulse factor according to the size of the sample. It further uses back propagation algorithm to fine-tune the parameters (weight and bias) of the model, and the goal is to minimize the difference between the error vector e (see Fig. 1) and the training result y.

3.5. An illustrative example

Fig. 9 illustrates a faulty program P with 16 statements that contains a faulty statement s_3 to show just how the three fault localization approaches are to be applied. In Fig. 9, the table is the input matrix of fault localization (see Fig. 1). Specifically, for the table,the cells below each statement indicate whether the statement is covered by the execution of a test case or not (1 for executed and 0 for not executed) and the rightmost cells represent whether the execution of a test case is failed or not (0 for pass and 1 for fail). There are 6 test cases, in which two of them failed (*i.e.* t_1 and t_6).

The first step of the three fault localization approaches is to construct a neural network. CNN-FL constructs a CNN model with one input layer with the number of nodes being 16 (*i.e.* the number of the statements), two convolution layers, two pooling layers with rectified linear units, two fully connected layers with the number of nodes in each layer simply set to be 8, and one output layer with the number of nodes being 1, outputting the result with *sigmoid* function. RNN-FL builds up an RNN model with one input layer with the 16 statements cut into 4 groups (*i.e.* each group with 4 statements), two recurrent layers and one output layer with the number of nodes being 1. MLP-FL constructs a MLP model with the number of input layer nodes being 16, 3 hidden layers with the number of nodes in each layer being 10 according to Eq. (6) and the number of output layer nodes being 1.

The second step is to train the networks with the coverage data, and input the error vector into the target vector. The batch size h is 3 in this example. Therefore, in CNN-FL, the first input matrix with a size of 3 is ((1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0), (1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0), (1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1)) and target output vector is (1, 0, 0); we secondly input the matrix ((1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1), (1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0), (1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0)) and its execution result vector (0, 0, 1). In MLP-FL, the first input vector is (1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0) and target output is 1. We secondly input the vector (1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0) and its execution result 0, and repeat this process. In RNN-FL, the first input vector is cut into 4 groups, they are (1, 1, 1, 0), (0, 0, 1, 1), (1, 1, 0, 1), (1, 1, 0, 0) and target output is 1. The second input is (1, 1, 1, 1), (1, 1, 0, 0), (0, 0, 0, 0), (1, 1, 0, 0) and its execution result 0, and repeat this process. These approaches will repeat training the networks using these data until the loss is small enough to reach the

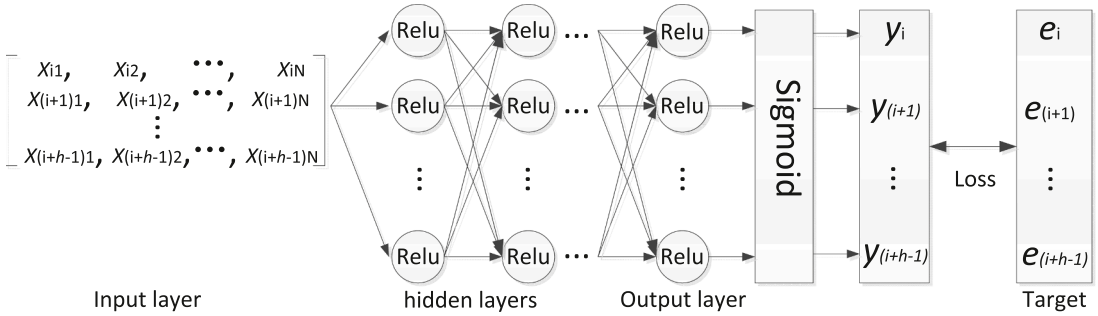
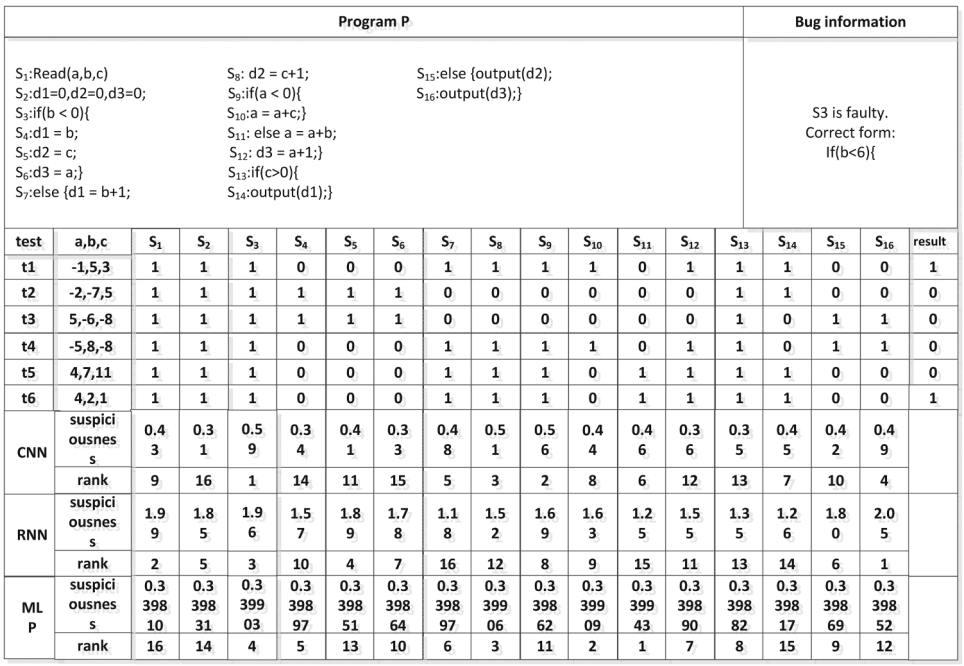


Fig. 8. The architecture of MLP-based fault localization.



Program P				Bug information

S_1:Read(a,b,c)
S_2:d1=0,d2=0,d3=0;
S_3:if(b < 0){
S_4:d1 = b;
S_5:d2 = c;
S_6:d3 = a;}
S_7:else {d1 = b+1;
	S_8: d2 = c+1;
	S_9:if(a < 0){
	S_{10}:a = a+c;}
	S_{11}: else a = a+b;
	S_{12}: d3 = a+1;}
	S_{13}:if(c>0){
	S_{14}:output(d1);}
		S_{15}:else {output(d2);
		S_{16}:output(d3);}
				S3 is faulty.
				Correct form:
				If(b<6){

test	a,b,c	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	result
t1	-1,5,3	1	1	1	0	0	0	1	1	1	1	0	1	1	1	0	0	1
t2	-2,-7,5	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0
t3	5,-6,-8	1	1	1	1	1	1	0	0	0	0	0	0	1	0	1	1	0
t4	-5,8,-8	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0
t5	4,7,11	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0
t6	4,2,1	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1
CNN	suspiciousness	0.43	0.31	0.59	0.34	0.41	0.33	0.48	0.51	0.56	0.44	0.46	0.36	0.35	0.45	0.42	0.49	
	rank	9	16	1	14	11	15	5	3	2	8	6	12	13	7	10	4	
RNN	suspiciousness	1.99	1.85	1.96	1.57	1.89	1.78	1.18	1.52	1.69	1.63	1.25	1.55	1.35	1.26	1.80	2.05	
	rank	2	5	3	10	4	7	16	12	8	9	15	11	13	14	6	1	
MLP	suspiciousness	0.39810	0.39831	0.39903	0.39897	0.39851	0.39864	0.39897	0.39906	0.39862	0.39909	0.39943	0.39890	0.39882	0.39817	0.39869	0.39852	
	rank	16	14	4	5	13	10	6	3	11	2	1	7	8	15	9	12	

Fig. 9. The Example illustrating three deep learning-based fault localization approaches.

condition of convergence. After training, the three models reveal the complex nonlinear relationship between the statement coverage and test results.

The third step is to construct a virtual test set defined in Fig. 2, whose size equals to the number of the statements. Thus, the virtual test set contains 16 test cases, where each test case covering only one statement out of 16 statements. We input the virtual test set into the trained networks and output the suspiciousness vector, where an element of the vector denotes the suspiciousness of a statement being faulty. This step finally organizes the localization result as a ranking list of all statements in descending order of suspiciousness. As shown in Fig. 9, the ranking list of CNN-FL is $\{s_3, s_9, s_8, s_{16}, s_7, s_{11}, s_{14}, s_{10}, s_1, s_{15}, s_5, s_{12}, s_{13}, s_4, s_6, s_2\}$; the ranking list of RNN-FL is $\{s_{16}, s_1, s_3, s_5, s_2, s_{15}, s_6, s_9, s_{10}, s_4, s_{12}, s_8, s_{13}, s_{14}, s_{11}, s_7\}$; the ranking list of MLP-FL is $\{s_{11}, s_{10}, s_8, s_3, s_4, s_7, s_{12}, s_{13}, s_{15}, s_6, s_9, s_{16}, s_5, s_2, s_{14}, s_1\}$. We can see that the faulty statement s_3 is ranked 1st by CNN-FL, ranked 3rd by RNN-FL and ranked 4th by MLP-FL.

4. Experimental study

4.1. Experimental setup

To evaluate the effectiveness of deep learning in locating real faults, we conduct a large-scale empirical study. Specifically, we adopt three deep-learning-based fault localization approaches (*i.e.* CNN-FL, RNN-FL, and MLP-FL), where three representative deep learning architectures (*i.e.* CNN, RNN, and MLP) are used. To obtain reliable experimental results, we collect 8 real-world programs from the widely used benchmarks in the field of software debugging (*i.e.* Defects4J[1], ManyBugs[2], and SIR[3]). These subject programs are equipped with all real faults from the development of large-sized programs varying from 6.1 KLOC to 491 KLOC.

Table 1 summarizes the characteristics of these subject programs. For each program, it provides a brief functional description (column 'Description'), the number of faulty versions (column 'Versions'), the number of thousand lines of statements (column 'KLOC'), the number of test cases (column 'Test'), and the type of the faults (column 'Type'). The first four programs are from Defect4J, which is a database and extensible framework providing real bugs to enable reproducible studies in software testing and debugging research [28]. The other four programs are also real-world subjects. Specifically, *python, gzip* and *libtiff* are collected from ManyBugs, and *space* is acquired from the SIR.

Table 1
The summary of subject programs.

Program	Description	Versions	KLOC	Test	Type
chart	JFreeChart	26	96	2205	Real
math	Apache Commons Math	106	85	3602	Real
mockito	Framework for unit tests	38	6	1075	Real
time	Joda-Time	27	53	4130	Real
python	General-purpose language	8	407	355	Real
gzip	Data compression	5	491	12	Real
libtiff	Image processing	12	77	78	Real
space	ADL interpreter	38	6.1	13585	Real

The physical environment on which our experiments were carried out was on a computer containing a CPU of Intel I5-2640 with 128G physical memory and two 12G GPUs of NVIDIA TITAN X Pascal. The operating systems were Ubantu 16.04.3. We conducted experiments on the MATLAB R2016b.

4.2. Evaluation metrics

To evaluate the effectiveness of deep-learning-based fault localization, we adopt two widely used metrics, namely *Exam* [29], and relative improvement (referred as *RImp*) [30]. *Exam* is defined as the percentage of executable statements to be examined before finding the actual faulty statement. A lower value of *Exam* indicates better performance. *RImp* is a metric of comparing two fault localization approaches to see the improvement of one approach over the other one. Given two approaches FL1 and FL2, FL2 is the baseline approach. *RImp* is to compare the total number of statements that need to be examined to find all faults using FL1 versus the number that need to be examined by using FL2. A lower

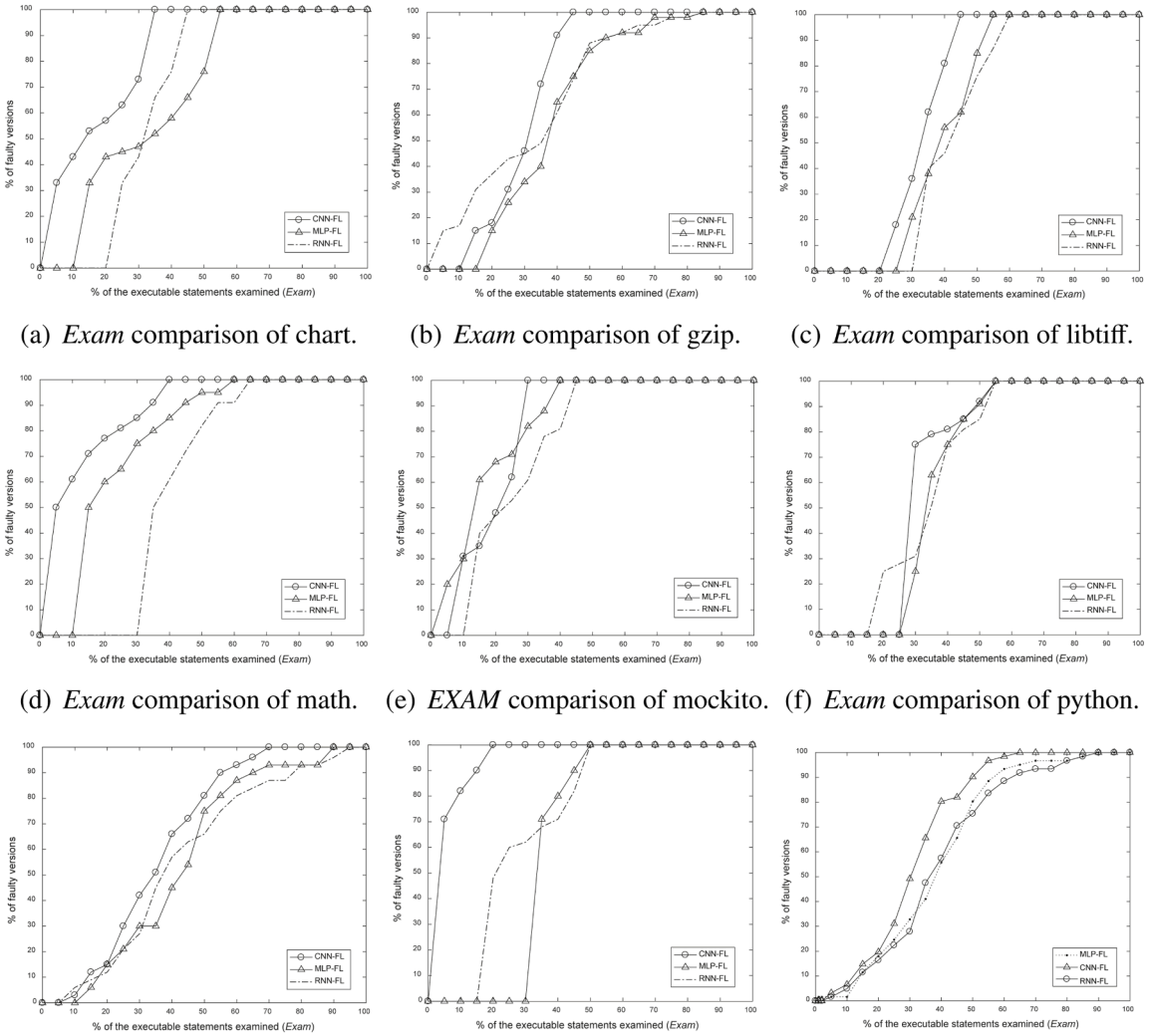
[1] Defects4J, https://github.com/rjust/defects4j
[2] ManyBugs, https://repairbenchmarks.cs.umass.edu/
[3] SIR, https://sir.csc.ncsu.edu/portal/index.php

value of *RImp* shows better improvement of FL1 over FL2.

4.3. Data analysis

In this subsection, we will present the results using *Exam*, and *RImp*, and the statistical results among the three localization approaches using deep learning. Furthermore, we compare deep-learning-based fault localization with three state-of-the-art localization approaches.***Exam distribution.*** Fig. 10 illustrates the *Exam* distribution of three deep learning approaches in each subject program. For each curve, x-axis represents the percentage of executable statements examined while y-axis denotes the percentage of faults already located in all faulty versions. A point in Fig. 10 means when a percentage of executable statements is examined in each faulty version, the percentage of faulty versions has located their faults. From the three curves of the nine figures, we can observe that the curve of CNN-FL is always above the other two curves (*i.e.* RNN-FL and MLP-FL). It means that CNN-FL performs the best among the three localization approaches. We can further observe that the curves of RNN-FL and MLP-FL interweave, *i.e.* , RNN-FL locates more faults than MLP-FL in some percentages of executable exammined while MLP-FL locates more faults than RNN-FL in other percentages of executable statements examined. Thus, we cannot obtain a conclusive result on the effectiveness relationship between RNN-FL and MLP-FL.

Table 2 illustrates different types of *Exam* scores in MLP-FL, CNN-FL and RNN-FL. For each approach, it provides three types of *Exam* scores: *best, average*, and *variance*. The *best* score means the minimum *Exam* score of all faulty versions of a program, showing the best effectiveness of an approach can reach. The *average* score means the average *Exam* score of all faulty versions of a program, showing the average effectiveness of a approach can achieve. and the *variance* score means the variance of *Exam* scores of all faulty versions of a program, showing whether the effectiveness of an approach is stable or not. From Table 2, we observe that among the 8 programs, CNN-FL mostly obtains the best cases (marked in bold font) in the three types of *Exam* scores, *i.e.* CNN-FL obtains 5 firsts in *best Exam* scores, 7 firsts in *average Exam* scores and 4 firsts in *variance Exam* scores. The results show that CNN-FL is more effective and more stable than MLP-FL and RNN-FL. However, for MLP-FL and RNN-FL, we can still see that their three types of *Exam* scores do not show an apparent advantage over each other.***RImp distribution.*** For a detailed improvement, we use *RImp* to evaluate the three deep-learning-based localization approaches. Specifically, Figs. 11–13show the RImp in three cases: CNN-FL versus MLP-FL, CNN-FL versus RNN-FL, and RNN-FL versus MLP-FL, where x-axis represents the program and y-axis denotes the *RImp* obtained in the program. Fig. 11 shows the *RImp* score of CNN-FL approach versus MLP-FL approach in each program.



(a) *Exam* comparison of chart. (b) *Exam* comparison of gzip. (c) *Exam* comparison of libtiff.

(d) *Exam* comparison of math. (e) *EXAM* comparison of mockito. (f) *Exam* comparison of python.

(g) *Exam* comparison of space. (h) *Exam* comparison of time. (i) *Exam* comparison of all the faulty versions.

Fig. 10. *Exam* distribution of CNN-FL, RNN-FL and MLP-FL in each program.

Table 2
Three types of *Exam* scores among CNN-FL, MLP-FL and RNN-FL.

		python	gzip	libtiff	space	chart	math	mockito	time
MLP-FL	best	0.133016	**0.006329**	0.195170	0.111660	0.145078	0.119571	**0.008208**	0.34484
	average	0.322419	0.244269	0.412189	0.402564	0.3565063	0.3493785	**0.1856458**	0.3416445
	variance	0.159671	0.174213	0.214892	0.167041	0.186936	0.324996	0.14786809	0.45191
CNN-FL	best	**0.033764**	**0.006329**	0.110802	0.100686	**0.041451**	**0.031021**	0.167024	**0.054494**
	average	**0.276977**	**0.104460**	**0.302835**	**0.339422**	**0.2208333**	**0.2146055**	0.2329684	**0.0721665**
	variance	0.208231	**0.088525**	**0.107424**	**0.139387**	0.155409	0.259628	**0.053328**	0.24993
RNN-FL	best	0.173271	0.118297	**0.046811**	**0.0738**	0.236184	0.168077	0.110992	0.159963
	average	0.357227	0.2675652	0.356843571	0.4122877	0.3172153	0.2424095	0.2617824	0.3071585
	variance	**0.145296**	0.137333	0.25899	0.199216	**0.087361**	**0.105122**	0.136977	**0.208166**

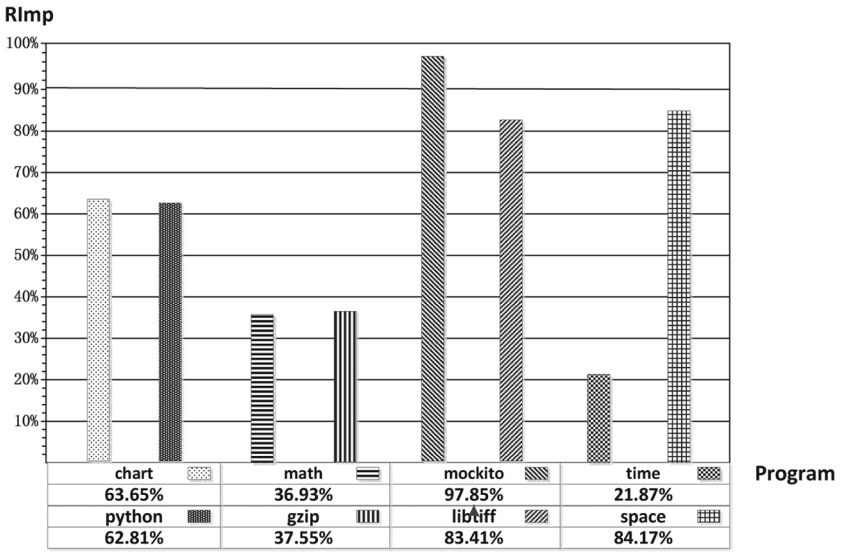


Fig. 11. *RImp* of CNN-FL versus MLP-FL.

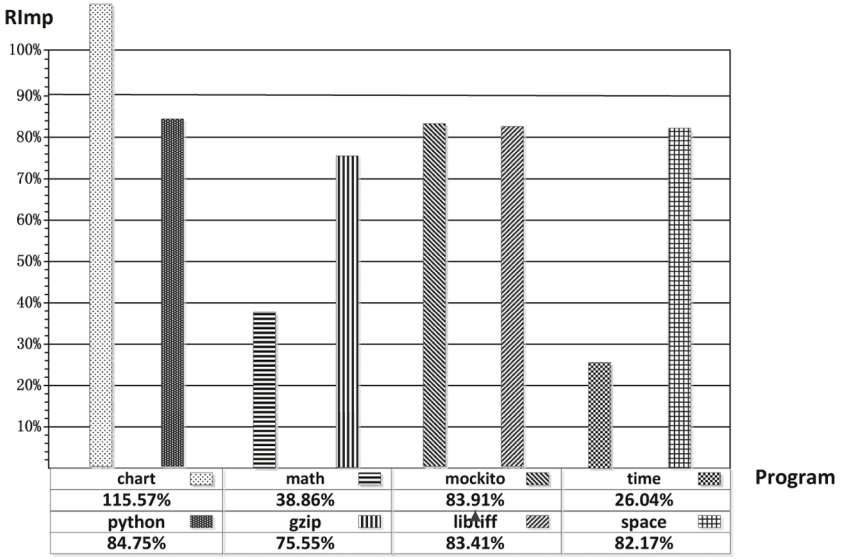


Fig. 12. *RImp* of CNN-FL versus RNN-FL.

With CNN architecture, the statements that need to be examined are cut down ranging from 21.87% such as *time* to 97.85% such as *mockito*. This means that CNN-FL needs to examine 21.87% to 97.85% of executed statements that MLP-FL needs to examine of. In other words, the maximum saving is 78.13% (100% - 21.87% = 78.13%) on *time*while the minimum saving is 2.15% (100% - 97.85% = 2.15%) on *mockito*. It means that the checking number of statements could be reduced from 2.15% to 78.13% when using CNN architecture versus MLP architecture for fault localization. Furthermore, the average saving is 38.97%, which also shows a significant improvement of CNN-FL over MLP-FL.

Fig. 12 shows the *RImp* score of CNN-FL approach versus RNN-FL approach. With CNN architecture, except for *chat*, the statements that

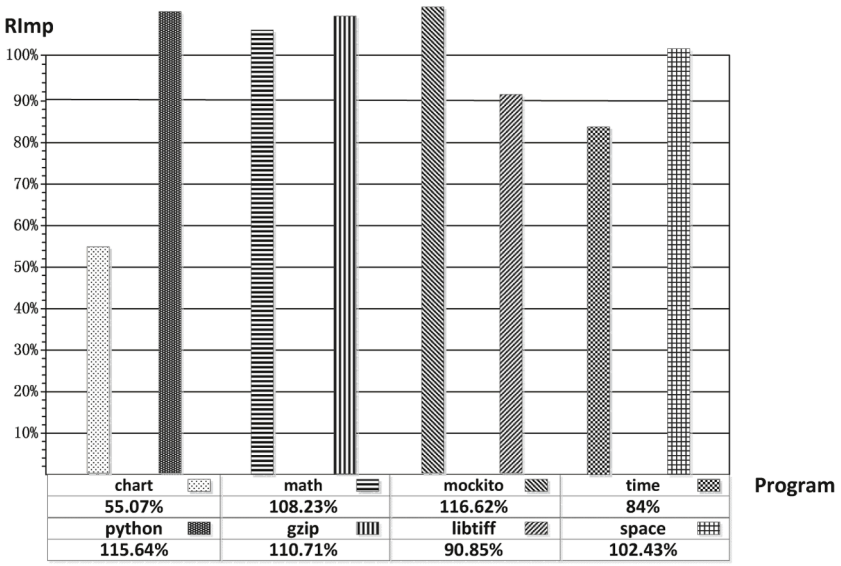


Fig. 13. *RImp* of RNN-FL versus MLP-FL.

need to be examined are cut down ranging from 26.04% on *time* to 84.75% on *python*. The maximum saving is 73.96% on *time*, and the average saving is 26.22%. Except for *chart*, the minimum saving is 19.25% on *python*. Therefore, CNN-FL significantly outperform RNN-FL.

Fig. 13 shows the *RImp* score of RNN-FL versus MLP-FL. With RNN architecture, the statements that need to be examined are cut down in *chart, time* and *libtiff*, the values of *RImp* are 55.07%, 84% and 90.85% respectively. But in other programs, the values of *RImp* are larger than 100%, meaning that MLP-FL performs better than RNN-FL.***Statistical comparison.*** Although *RImp* can show a detailed improvement, the analysis using *RImp* evaluates effectiveness from the overview of the results, and may miss other detailed view of the results. For example, as shown in Fig. 11, the *RImp* score of *mockito*is very close to 100%, which means that CNN-FL performs closely to MLP-FL in this program. However, a case may happen. Suppose that CNN-FL has higher but not quite higher effectiveness than MLP in each faulty version of a program, the *RImp* score will show that CNN performs closely to MLP. Nevertheless, in this case, it is difficult to conclude CNN-FL performs closely to MLP-FL because CNN-FL performs better than MLP-FL in each faulty version of the program. For another example, suppose that CNN-FL just has very higher effectiveness than MLP-FL in several faulty versions of a program. However, MLP-FL has moderately higher effectiveness in most faulty versions of the programs. The sheer high effectiveness of CNN-FL in the several faulty versions may make its *RImp* score lower than MLP, showing that CNN-FL performs better than MLP-FL. In such case, we cannot conclude that CNN-FL performs better than MLP-FL.

Thus, we need a more rigourous method to obtain a detailed result and adopt Wilcoxon-Signed-Rank Test [31] to achieve this goal. Wilcoxon-Signed-Rank Test is a non-parametric statistical hypothesis test for testing the differences between pairs of measurements F(x) and G(y). At the given significant level σ, we can use both 2-tailed and 1-tailed p-value to obtain a conclusion. For the 2-tailed p-value, if $p \geq \sigma$, the null hypothesis H_0 that F(x) and G(y) are not significantly different is accepted; otherwise, the alternative hypothesis H_1 that F(x) and G(y) are significantly different is accepted. For 1-tailed p-value, there are two cases: the right-tailed case and the left-tailed case. In the right-tailed case, if $p \geq \sigma$, H_0 that F(x) does not significantly tend to be greater than the G(y) is accepted; otherwise, H_1 that F(x) significantly tends to be greater than the G(y) is accepted. And in the left-tailed case, if $p \geq \sigma$, H_0 that F(x) does not significantly tend to be less than the G(y) is accepted; otherwise, H_1 that F(x) significantly tends to be less than the G(y) is accepted.

The experiments performed one paired Wilcoxon-Signed-Rank test between each two localization models by using *Eaxm* as the pairs of measurements F(x) and G(y). Specifically, each test uses both the 2-tailed and 1-tailed checking at the σ level of 0.05. Given two Localization models (M_1 and M_2), we use the list of *Exam* of one localization model M_1 in all faulty versions of all programs as the list of measurements of F(x), while the list of measurements of G(y) is the list of *Exam* of the other localization model M_2 in all faulty versions of all programs. Hence, in the 2-tailed test, M_1 has SIMILAR effectiveness as M_2 when H_0 is accepted at the significant level of 0.05. In the 1-tailed test (right), M_1 has WORSE effectiveness than M_2 when H_1 is accepted at the significant level of 0.05. Finally, in the 1-tailed test (left), M_1 has BETTER effectiveness than M_2 when H_1 is accepted at the significant level of 0.05.

Table 3, 4 and 5 show the statistical results using Wilcoxon-Signed-Rank test. Each row shows the p values and the conclusion for each program. The 'Total' row illustrates the statistical results of Wilcoxon-Signed-Rank test on all programs. Table 3 shows the statistical results of Wilcoxon-Signed-Rank test on CNN-FL versus MLP-FL. Take *chart* as an example. The p values of 2-tailed, 1-tailed(right) and 1-tailed(left) are 0.011, 0.969 and 0.009 respectively. It means that the *EXAM* of CNN-FL is significantly less than that of MLP-FL. Therefore, we obtain a BETTER conclusion, that is, CNN-FL performs better than MLP-FL in the program *chart*. Based on the results in Table 3, we could see that CNN-FL obtain all BETTER results except for *mokito*, and thus conclude that CNN-FL significantly outperforms than MLP-FL.

Table 4 shows the statistical results on CNN-FL versus RNN-FL. We can observe that CNN-FL obtain BETTER results over RNN-FL in each subject program and in total comparison, that is, the Exam of CNN-FL is significantly less than that of RNN-FL in all subject programs. Therefore,

Table 3
Statistical results (CNN-FL versus MLP-FL).

Program	Wilcoxon-Signed-Rank test				A-Test
	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	
chart	0.011	0.969	0.009	BETTER	0.78
gzip	0.028	0.989	0.018	BETTER	0.72
libtiff	0.018	0.993	0.011	BETTER	0.63
math	0.012	0.971	0.010	BETTER	0.78
mokito	0.686	0.394	0.705	SIMILAR	0.36
python	0.011	0.995	0.007	BETTER	0.69
space	0.029	0.986	0.015	BETTER	0.61
time	0.018	0.963	0.019	BETTER	1.00
Total	2.26E-03	0.999	1.14E-03	BETTER	0.62

Table 4
Statistical results (CNN-FL versus RNN-FL).

Program	Wilcoxon-Signed-Rank test				A-Test
	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	
chart	0.029	0.909	0.021	BETTER	0.78
gzip	0.014	0.947	0.009	BETTER	0.84
libtiff	0.036	0.863	0.041	BETTER	0.57
math	0.045	0.814	0.043	BETTER	0.75
mokito	0.047	0.791	0.030	BETTER	0.60
python	0.032	0.808	0.043	BETTER	0.56
space	0.008	0.962	0.020	BETTER	0.60
time	0.018	0.963	0.019	BETTER	1.00
Total	0.016	0.991	9.03E-03	BETTER	0.61

Table 5
Statistical results (RNN-FL versus MLP-FL).

Program	Wilcoxon-Signed-Rank test				A-Test
	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	
chart	0.039	0.789	0.040	BETTER	0.78
gzip	0.686	0.394	0.705	SIMILAR	0.50
libtiff	0.044	0.764	0.042	BETTER	0.63
math	0.655	0.510	0.814	SIMILAR	0.78
mokito	0.023	0.014	0.911	WORSE	0.36
python	1.00	0.572	0.572	SIMILAR	0.69
space	0.782	0.394	0.613	SIMILAR	0.60
time	0.045	0.814	0.041	BETTER	1.00
Total	0.842	0.422	0.580	SIMILAR	0.60

we obtain a BETTER conclusion: CNN-FL significantly performs better than RNN-FL.

Table 5 shows the statistical results on RNN-FL versus MLP-FL. We can see that RNN-FL obtain three BETTER results over MLP-FL in *chart, libtiff* and *time*, four SIMILAR results in *gzip, mth, python* and *space*, one WORSE result in *mokito*. It seems that RNN-FL is more effective than MLP-FL due to more BETTER and SIMILAR results over MLP-FL. However, in total, RNN-FL has a SIMILAR result over MLP-FL. Thus, we conclude that the effectiveness of RNN-FL is comparable to that of MLP-FL even if the effectiveness of RNN-FL is marinally higher than MLP-FL.

To further assess the difference quantitatively, we leverage the nonparametric Vargha-Delaney A-test, which is recommended in [32], to evaluate the magnitude of the difference by measuring effect size (scientific significance). For A-test, the bigger deviation of A-statistic is from the value of 0.5, the greater difference is between the two studied groups. Vargha and Delaney [33] suggest that A-test of greater than 0.64 (or less than 0.36) is indicative of "medium" effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising "large" effect size. The 'A-test' column of Tables 3–5 show the effect size of the A-test on the three scenarios (*i.e.* CNN-FL versus MLP-FL, CNN-FL versus RNN-FL and RNN-FL versus MLP-FL). Tables 3 and 4 show that CNN-FL often arriving at the promising "large" effect size, *i.e.* CNN-FL both obtains 4 "large" effect size results on MLP-FL and RNN-FL respectively. Table 5 shows that the effectiveness of RNN-FL is comparable to that of MLP-FL.

In summary, we have the conclusion on the effectiveness of using three deep learning architectures in locating real faults, *i.e.* , CNN shows the highest localization effectiveness while RNN and MLP have the comparable effectiveness in locating real faults.

Comparison on state-of-the-art approaches. The recent study [2] empirically summarized the state-of-the-art fault localization approaches, and we use the top three techniques (*i.e.* Dstar [14], Barinel [15] and Ochia [16]) for our study. The three techniques are spectrum-based fault localization (SFL) [1]. Since CNN-FL shows the highest localization effectiveness, we compare CNN-FL with the three state-of-art fault localization techniques to evaluate the improvement.

The benchmark Defects4J (*i.e. chart, math, mockito* and *time*) used by our study is over-fitting for SFL including the three state-of-the-art techniques [34,35], *i.e.* SFL shows inconsistencies between the benchmark Defects4J and other benchmarks in terms of fault localization effectiveness. For example, 44% and 43% of bugs in Defects4J are localized at top 10 using Ochiai and Dstar while none of the bugs in other benchmarks can be localized even in top 100 [34,35] with these techniques. Due to the over-fitting problem, it is extremely difficult for other localization approaches to outperform SFL on Defects4J (see Tables 6–8). To alleviate the over-fitting problem, we further add the recommended benchmark BEARS [36] (*i.e. apache/incubator-dubb, INRIA/spoon, FasterXML/jackson-databind* and *apache/jackrabbit-oak*) for the comparison between CNN-FL and the three localization techniques. The benchmark BEARS has 251 real faults and each faulty version has an average 212 KLOC (*i.e.* the number of thousands of lines of statements).

Tables 6 –8 show the statistical results of CNN-FL versus the three localization techniques using Wilcoxon-Signed-Rank test and Vargha-Delaney A-test. Except for the WORSE results in Defects4J caused by the benchmark over-fitting problem, CNN-FL obtains BETTER and "large" effect size results in almost all programs. Thus, CNN-FL significantly outperforms the three localization techniques.

4.4. Discussion

The experimental results show that CNN is the most effective one in locating real faults among the 3 representative deep learning architectures. Although deep learning is a black box technology, we still try to understand what factors lead to the advantage of CNN over the other two deep learning architectures. It is natural to seek the factors from the characteristics of CNN over the other architectures. CNN has the three characteristics distinct from the others, *i.e.* local connections, parameter sharing and down-sampling in pooling. Local connections in CNN are accomplished by making a kernel much smaller than the input. The kernel could catch sight of small but meaningful features. That can improve learning efficiency for reducing the memory requirements by leveraging fewer parameters and computing the output with fewer operations. Parameter sharing uses the same parameters for more than one function in a model. It also means that the model only learns one set of parameters rather than learning multiple-sets of parameters. Parameter sharing makes the learning more efficient than dense matrix multiplication. Down-sampling in pooling further reduces the amount of output parameters. It also gives the tolerance of the model's slight deformation and enhances the generation ability of the model.

We can observe that the three unique characteristics of CNN focuses on reducing the size of parameters while preserving the learning quality without loss of features. In fault localization practice, the program statements are always existing in great numbers and the input data sets

Table 6
Statistical results (CNN-FL versus Dstar).

Program	Wilcoxon-Signed-Rank test				A-Test
	2-tailed	1-tailed (right)	1-tailed (left)	Conclusion	
Defects4J	0.010	0.009	0.972	WORSE	0.09
gzip	0.018	0.876	0.011	BETTER	0.73
libtiff	0.003	0.914	0.002	BETTER	0.78
python	0.002	0.945	0.004	BETTER	0.76
space	8.11E-04	0.999	4.34E-04	BETTER	0.83
apache/incubator-dubbo	0.032	0.859	0.039	BETTER	0.72
INRIA/spoon	0.996	0.605	0.605	SIMILAR	0.50
FasterXML/jackson-databind	0.015	0.899	0.014	BETTER	0.73
apache/jackrabbit-oak	0.001	0.969	0.002	BETTER	0.80
Total	9.46E-05	1.00E+00	4.79E-05	BETTER	0.74

Table 7
Statistical results (CNN-FL versus Barinel).

Program	Wilcoxon-Signed-Rank test				A-Test
	2-tailed	1-tailed (right)	1-tailed (left)	Conclusion	
Defects4J	0.014	0.015	0.969	WORSE	0.17
gzip	0.034	0.860	0.021	BETTER	0.74
libtiff	0.041	0.723	0.034	BETTER	0.69
python	0.027	0.899	0.018	BETTER	0.72
space	0.001	0.999	5.34E-04	BETTER	0.81
apache/incubator-dubbo	0.039	0.789	0.040	BETTER	0.71
INRIA/spoon	0.002	0.909	0.011	BETTER	0.81
FasterXML/jackson-databind	0.029	0.824	0.022	BETTER	0.72
apache/jackrabbit-oak	0.008	0.985	0.003	BETTER	0.76
Total	1.94E-05	1.00E+00	9.82E-06	BETTER	0.77

Table 8
Statistical results (CNN-FL versus Ochiai).

Program	Wilcoxon-Signed-Rank test				A-Test
	2-tailed	1-tailed (right)	1-tailed (left)	Conclusion	
Defects4J	0.017	0.012	0.974	WORSE	0.12
gzip	0.002	0.961	0.005	BETTER	0.81
libtiff	0.012	0.823	0.036	BETTER	0.74
python	0.026	0.916	0.015	BETTER	0.76
space	0.001	0.999	5.19E-4	BETTER	0.79
apache/incubator-dubbo	0.033	0.817	0.035	BETTER	0.72
INRIA/spoon	0.013	0.936	0.009	BETTER	0.78
FasterXML/jackson-databind	0.038	0.817	0.023	BETTER	0.71
apache/jackrabbit-oak	0.002	0.969	0.009	BETTER	0.80
Total	3.95E-05	1.00E+00	2.00E-05	BETTER	0.75

of the network are accordingly tremendous. The three unique characteristics of CNN will not cause a rapid expansion of the number of parameters to be trained. However, without the characteristics of reducing the size of parameters, it causes the number of hidden layers to be very large for RNN and MLP, accompanying with the rapid expansion of the number of parameters to be trained, and potentially leading to the disadvantage in fault localization.

For example, let us consider an input whose dimension is 100,000, connecting a hidden layer of the same size. There will be 100,000 * 100,000 = 10 billion connections. Due to the larger number of connections, there will be 10 billion parameters to be calculated and trained. However, the computing power and training data cannot satisfy this requirement. Therefore, it is necessary to reduce the number of parameters to be trained, to reduce the computational complexity, and to prevent over-fitting. After using the network structure of the CNN, the fully connected mode is changed into local connections. Assuming that the size of the local receptive field is 100, each dimension only needs to be connected with 100 instead of 100,000. Hence, the network only requires 100,000*100 = 10,000,000 connections, reduced by 100 times. When the number of layers is increased, the number of parameters is reduced even more. Therefore, we conjecture that it is the reason why CNN has the advantage over the other two architectures in fault localization. We suggest that reducing the number of parameters may be a key factor of deep learning for improving fault localization. Our future work will seek the optimization on the reduction of parameters (*e.g.* the optimization on the input model and the learning process).

To verify the above discussion, we use the Kendall rank correlation coefficient [37] (denoted by r) to measure the correlation between localization effectiveness and the size of the parameters in all the three fault localization approaches using deep learning. The Kendall rank correlation coefficient is a statistic used to measure the correlation between two measured variables. The value of the correlation coefficient r is in the range of $[-1, +1]$. When r lies around +1 or -1, then it means a perfect degree of correlation between the two variables. As the correlation coefficient value goes towards 0, the correlation between the two variables will be weaker. Based on the absolute value of the correlation coefficient r, five cases are usually used:

1. $r \in [0.8, 1.0]$, very strong correlation;
2. $r \in [0.6, 0.8)$, strong correlation;
3. $r \in [0.4, 0.6)$, medium correlation;
4. $r \in [0.2, 0.4)$, weak correlation;
5. $r \in [0.0, 0.2)$, very weak correlation or independence;

Table 9 shows the Kendall rank correlation coefficient between the localization effectiveness and the size of parameters on all the three deep-learning-based fault localization techniques in each program. Except one medium correlation in *libtiff*, all the results at least show strong correlation. This means that there is statistically significant correlation between the localization effectiveness and the size of parameters used by deep-learning-based fault localization approaches. Thus, reducing the number of parameters may be a key factor of deep learning for improving fault localization.

4.5. Threats to validity

There are some threats to the validity of our experiments. We adopted deep neural networks, meaning the fault localization results are not the same through different training times. It is the characteristic of neural network technologies. To make the results more reliable, we followed the convention by repeating the fault localization process, *i.e.* , we computed ten times and used the average score as the results for the experimental study.

Another threat to external validity is the subject programs used for our experiments. We use those large-sized programs equipped with all real faults from the real-world development, commonly used in the field of software debugging. However, the experimental results may not apply to all programs because there are still many unknown and complicated factors in realistic debugging that could affect the experimented results. Thus, it is worthwhile to conduct experiments on more subject programs to further strengthen the experimental results.

We adopt the widely used metrics, *Exam* and *RImp*, to evaluate the effectiveness of deep-learning-based fault localization. According to the extensive use of the measurements, so the threat is acceptably mitigated.

5. Related work

This section surveys closely fault localization studies, especially coverage-based fault localization and fault localization using machine learning. More work on fault localization can refer to a survey [38] by Wong et al.

Coverage-based fault localization techniques convert program spectrum data from test executions to suspiciousness score of program entities and rank them in descending order [1]. When using these techniques, we do not need to know the details of a program and just run the program with passed and failed test cases. Among existing

Table 9
Kendall rank correlation coefficient results.

Program	Coefficient r	Program	Coefficient r
chart	0.73	mokito	0.61
gzip	0.67	python	0.65
libtiff	0.54	space	0.62
math	0.73	time	0.87

coverage-based localization methods, spectrum-based fault localization (SBFL) is the most popular one by using spectrum-based suspiciousness formulas to assign suspiciousness values of being faulty on program statements. Chen et al. [39] proposed the Jaccard technique. Jones et al. [40] proposed the tarantula technique that is a widely used and compared technique in the subsequent studies. Abreu et al. [41] applied the Ochiai for locating single-fault programs. Wong et al. [42] used data and control flow and presented several metrics such as Wong1-3, Wong3'. Wong et al. [14,43] also proposed an approach named DStar (D^*) based on crosstab with utilization of statements' coverage and execution information. Xie et al. [44,45] theoretically summarized the maximal formulas from many existing SBFL formulas. Furthermore, Pearson et al. [2] empirically summarized the SBFL formulas, showing different localization effects between artificial faults and real faults. SBFL is also adopted and evaluated in different applications, software product lines [46].

Besides, Papadakis et al. [47] first proposed mutation-based fault localization techniques, where the use of mutation analysis for locating faults was advocated. Then Papadakis et al. [48] proposed another mutation-based fault localization technique named Metallaxis-FL, which did not require finding a mutant that makes all the test cases pass and always suggested a possible suspiciousness ranking. Moon et al. [49] presented MUSE, which utilized two groups of mutants, one mutated a faulty statement and the other mutated a correct statement. They also proposed a new evaluation metric called Locality Information Loss. Different from those aforementioned studies, our study focuses on the localization effectiveness of using deep learning in locating real faults.

Machine learning techniques are used in the context of fault localization based on statement coverage and execution results of test cases. Wong et al. [11] proposed a fault localization approach based on back-propagation (BP) neural network, which has a simple structure to implement. Due to the drawbacks of BP networks (*e.g.* paralysis), Wong et al. [50] proposed another approach based on radial basis function (RBF) networks. Recently, deep learning methods have witnessed a rapid development to tackle the limitations of traditional machine learning techniques and is utilized in many disciplines such as computer vision and natural language processing. Based on the methods proposed by Wong and the advantage of deep learning methods, Zheng et al. [9] presented a fault localization method based on Multi-Layer Perceptrons (MLPs). Zhang et al. [51] enhanced fault localization efficiency based on deep neural network with dynamic slice technology. Briand et al. [52] proposed a fault localization method based on decision tree algorithm constructing rules that classify test cases into various partitions. Li et al. [53] furthermore propose DeepFL using various feature dimensions to locate method-level faults while our study focuses on locating statement-level ones. Deep-learning-based fault localization has shown its promising results over a wide spectrum of the state-of-the-art localization techniques [10,51] (*e.g.* BP neural networks [11], PPDG [12], Tarantula [13], Dstar [14], Barinel [15], and Ochia [16]). Thus, we use more basic and representative deep learning architectures to conduct a large-scale study on effectiveness of using deep learning in locating real faults.

6. Conclusion

Recently, deep-learning-based fault localization has shown its promising results in fault localization. Thus, this paper explores more on using deep learning for fault localization in the context of real faults. We use three representative and popular deep learning architectures (*i.e.* CNN, RNN and MLP) for fault localization, where we propose CNN-FL and RNN-FL. Furthermore, we collect 8 real-world programs with all real faults from the widely used benchmarks (*i.e.* Defects4J, ManyBugs and SIR). We conduct a large-scale study on the 8 large-sized programs by using the three deep-learning-based fault localization approaches. The experimental results show that CNN performs the best among the three deep learning architectures while RNN and MLP have the similar effectiveness in locating real faults. We analyze the underlying advantage of CNN over the other two architectures, and suggest that reducing the number of parameters may be a key factor of deep learning for improving fault localization.

Slightly further in the future, we plan to study more on deep-learning-based fault localization, trying to improve the effectiveness. Moreover, seeking way to extend deep learning to multiple-bugs cases is of great interest to our research.

CRediT authorship contribution statement

Zhuo Zhang: Conceptualization, Methodology, Software, Data curation, Writing - original draft. **Yan Lei:** Conceptualization, Methodology, Writing - review & editing, Supervision. **Xiaoguang Mao:** Methodology, Resources, Writing - review & editing, Supervision. **Meng Yan:** Writing - review & editing. **Ling Xu:** Writing - review & editing. **Xiaohong Zhang:** Writing - review & editing.

Declaration of Competing Interest

None.

Acknowledgments

This work is partially supported by Guangxi Key Laboratory of Trusted Software (No. kx202008), the National Defense Basic Scientific Research Project (No. WDZC20205500308), the National Natural Science Foundation of China (Nos. 62002034, 61602504), and the Fundamental Research Funds for the Central Universities (Nos. 2019CDXYRJ0011, 2020CDCGRJ037, 2020CDCGRJ072).

References

[1] L. Naish, Hua, A model for spectra-based software diagnosis, ACM Trans. Softw. Eng.Methodol. (TOSEM) 20 (3) (2011) 1–32.
[2] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, B. Keller, Evaluating and improving fault localization. Proceedings of the International Conference on Software Engineering (ICSE 2017), 2017, pp. 609–620.
[3] C. Parnin, A. Orso, Are automated debugging techniques actually helping programmers?. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2011), 2011, pp. 199–209.
[4] C. Sun, S.C. Khoo, Mining succinct predicated bug signatures. Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE 2013), 2013, pp. 576–586.
[5] T.D.B. Le, R.J. Oentaryo, D. Lo, Information retrieval and spectrum based bug localization: better together. Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE 2015), 2015, pp. 579–590.
[6] J. Bo, Z. Zhang, W.K. Chan, T.H. Tse, T.Y. Chen, How well does test case prioritization integrate with statistical fault localization? Inf. Softw. Technol. 54 (7) (2012) 739–758.
[7] Y. Lei, C. Sun, X. Mao, Z. Su, How test suites impact fault localisation starting from the size, IET Softw. 12 (3) (2018) 190–205.
[8] Y. Lecun, Y. Bengio, G.e. Hinton, Deep learning, Nature 521 (7553) (2015) 436.
[9] W. Zheng, D. Hu, J. Wang, Fault localization analysis based on deep neural network, Math. Prob.Eng. 2016 (2016) 1–11.
[10] Z. Zhang, Y. Lei, X. Mao, P. Li, CNN-FL: An effective approach for localizing faults using convolutional neural networks. Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019), IEEE, 2019, pp. 445–455.
[11] W.E. WONG, Y. QI, Bp neural network-based effective fault localization, Int. J. Softw. Eng. Knowl. Eng. 19 (04) (2009) 573–597.
[12] G.K. Baah, A. Podgurski, M.J. Harrold, The probabilistic program dependence graph and its application to fault diagnosis. International Symposium on Software Testing and Analysis, 2008, pp. 189–200.
[13] J.A. Jones, Empirical evaluation of the tarantula automatic fault-localization technique. Proceedings of the International Conference on Automated Software Engineering (ICSE 2005), 2005, pp. 273–282.
[14] W.E. Wong, V. Debroy, Y. Li, R. Gao, Software fault localization using dstar (d*). Proceedings of the 6th International Conference on Software Security and Reliability, 2012, pp. 21–30.
[15] A. Rui, P. Zoeteweij, A.J.C.V. Gemund, Spectrum-based multiple fault localization. Proceedings of the International Conference on Automated Software Engineering (ASE 2009), 2009, pp. 88–99.
[16] A. Rui, P. Zoeteweij, R. Golsteijn, A.J.C.V. Gemund, A practical evaluation of spectrum-based fault localization, J. Syst. Softw. 82 (11) (2009) 1780–1792.

[17] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, et al., Recent advances in convolutional neural networks, Pattern Recognit. 77 (2018) 354–377.
[18] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[19] F.A. Gers, N.N. Schmidhuber, J. Schmidhuber, Learning precise timing with LSTM recurrent networks, J. Mach. Learn. Res. 3 (2002).
[20] S.O. G. E. Hinton, Y.-W. Teh, A fast learning algorithm for deep belief nets. Neural Computation, 2006, pp. 1527–1554.
[21] D. Yu, F. Seide, G. Li, Conversational speech transcription using context-dependent deep neural networks. Proceedings of the International Conference on International Conference on Machine Learning (ICML 2012), 2012, pp. 1–2.
[22] K. Jarrett, K. Kavukcuoglu, M. Ranzato, Y. Lecun, What is the best multi-stage architecture for object recognition?. Proceedings of the IEEE International Conference on Computer Vision, 2010, pp. 2146–2153.
[23] Y. Lecun, F.J. Huang, L. Bottou, Learning methods for generic object recognition with invariance to pose and lighting. Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on, 2004.
[24] H. Lee, R. Grosse, R. Ranganath, A.Y. Ng, Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009), 2009, pp. 609–616.
[25] J.D. N. Pinto D. Doukhan, D. Cox, A high-throughput screening approach to discovering good forms of biologically inspired visual representation. PLoS Computational Biology vol. 5, 2009, p. e1000579.
[26] S.C. Turaga, J.F. Murray, V. Jain, F. Roth, M. Helmstaedter, K. Briggman, W. Denk, H.S. Seung, Convolutional networks can learn to generate affinity graphs for image segmentation, Neural Comput. 22 (2) (2010) 511–538.
[27] A. Graves, A.-r. Mohamed, G. Hinton, Speech recognition with deep recurrent neural networks. Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE, 2013, pp. 6645–6649.
[28] D. Jalali, M.D. Ernst, Defects4J: a database of existing faults to enable controlled testing studies for java programs. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2014), 2014, pp. 437–440.
[29] X. Mao, Y. Lei, Z. Dai, Y. Qi, C. Wang, Slice-based statistical fault localization, Journal of Systems and Software 89 (1) (2014) 51–62.
[30] V. Debroy, W.E. Wong, X. Xu, B. Choi, A grouping-based strategy to improve the effectiveness of fault localization techniques. Proceedings of the International Conference on Quality Software (QSIC 2010), 2010, pp. 13–22.
[31] G.W. Corder, D.I. Foreman, Nonparametric statistics for non-statisticians: a step-by-step approach vol. 78, International Statistical Review, 2010.
[32] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering. 2011 33rd International Conference on Software Engineering (ICSE), IEEE, 2011, pp. 1–10.
[33] A. Vargha, H.D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong, J. Educ. Behav. Stat. 25 (2) (2000) 101–132.
[34] D. Zou, J. Liang, Y. Xiong, M.D. Ernst, L. Zhang, An empirical study of fault localization families and their combinations, IEEE Trans. Softw. Eng. (2019).
[35] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. Van Hoorn, A. Filieri, D. Lo, An evaluation of pure spectrum-based fault localization techniques for large-scale software systems, Software: Practice and Experience 49 (8) (2019) 1197–1224.
[36] F. Madeiral, S. Urli, M. Maia, M. Monperrus, Bears: an extensible java bug benchmark for automatic program repair studies. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 468–478.
[37] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.
[38] W.E. Wong, R. Gao, Y. Li, A. Rui, F. Wotawa, A survey on software fault localization, IEEE Trans. Softw. Eng. (TSE) 42 (8) (2016) 707–740.
[39] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: problem determination in large, dynamic internet services. Proceedings of International Conference on Dependable Systems and Networks (DSN 2002), 2002, pp. 595–604.
[40] J.A. Jones, Fault localization using visualization of test information. Proceedings of the International Conference on Software Engineering (ICSE 2004), 2004, pp. 54–56.
[41] R. Abreu, P. Zoeteweij, A.J.C. van Gemund, An evaluation of similarity coefficients for software fault localization. Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, 2006, pp. 39–46.
[42] W.E. Wong, Y. Qi, L. Zhao, K.Y. Cai, Effective fault localization using code coverage. Proceedings of International Computer Software and Applications Conference (COMPSAC 2007), 2007, pp. 449–456.
[43] W.E. Wong, V. Debroy, B. Choi, A family of code coverage-based heuristics for effective fault localization, J. Syst. Softw. 83 (2) (2010) 188–208.
[44] X. Xie, T.Y. Chen, F.C. Kuo, B. Xu, A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization, ACM Trans. Softw. Eng.Methodol. (TOSEM) 22 (4) (2013) 31.
[45] X. Xie, F.-C. Kuo, T.Y. Chen, S. Yoo, M. Harman, Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. International Symposium on Search Based Software Engineering, Springer, 2013, pp. 224–238.
[46] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, L. Etxeberria, Spectrum-based fault localization in software product lines, Inf. Softw. Technol. (2018) 18–31.
[47] M. Papadakis, Y.L. Traon, Using mutants to locate “unknown” faults. Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012), 2012, pp. 691–700.
[48] M. Papadakis, Y. Le Traon, Metallaxis-FL: Mutation-based fault localization, Softw. Test. Verif. Reliab. 25 (5–7) (2015) 605–628.
[49] S. Moon, Y. Kim, M. Kim, S. Yoo, Ask the mutants: mutating faulty programs for fault localization. Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST 2014), 2014, pp. 153–162.
[50] W.E. Wong, V. Debroy, R. Golden, X. Xu, B. Thuraisingham, Effective software fault localization using an RBF neural network, IEEE Trans. Reliab. 61 (1) (2012) 149–169.
[51] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng, X. Chang, Deep learning-based fault localization with contextual information, IEICE Trans. Inf. Syst. E100.D (12) (2017) 3027–3031.
[52] L.C. Briand, Y. Labiche, X. Liu, Using machine learning to support debugging with tarantula. Proceedings of the IEEE International Symposium on Software Reliability, 2007, pp. 137–146.
[53] X. Li, W. Li, Y. Zhang, L. Zhang, DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 169–180.