Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2016, Article ID 1820454, 11 pages
http://dx.doi.org/10.1155/2016/1820454

Research Article

Fault Localization Analysis Based on Deep Neural Network

Wei Zheng, Desheng Hu, and Jing Wang

College of Software & Microelectronics, Northwestern Polytechnical University, Xi'an, China

Correspondence should be addressed to Desheng Hu; hudeshengnpu@gmail.com

Received 24 December 2015; Accepted 31 March 2016

Academic Editor: Wen Chen

With software's increasing scale and complexity, software failure is inevitable. To date, although many kinds of software fault localization methods have been proposed and have had respective achievements, they also have limitations. In particular, for fault localization techniques based on machine learning, the models available in literatures are all shallow architecture algorithms. Having shortcomings like the restricted ability to express complex functions under limited amount of sample data and restricted generalization ability for intricate problems, the faults cannot be analyzed accurately via those methods. To that end, we propose a fault localization method based on deep neural network (DNN). This approach is capable of achieving the complex function approximation and attaining distributed representation for input data by learning a deep nonlinear network structure. It also shows a strong capability of learning representation from a small sized training dataset. Our DNN-based model is trained utilizing the coverage data and the results of test cases as input and we further locate the faults by testing the trained model using the virtual test suite. This paper conducts experiments on the Siemens suite and Space program. The results demonstrate that our DNN-based fault localization technique outperforms other fault localization methods like BPNN, Tarantula, and so forth.

1. Introduction

Many efforts have been made for debugging a generic program, especially in the stage of identifying where the bugs are, which is known as fault localization. It has been proved that fault localization is one of the most expensive and time-consuming debugging activities. With software's increasing scale and complexity, the debugging activities are more difficult to perform; thus, there is a high demand for automatic fault localization techniques that can guide programmers to the locations of faults [1]. In this way, it will facilitate the software development process and reduce maintenance cost [2]. At present, many kinds of software fault localization methods have been proposed. Probabilistic program dependence graph (PPDG) is presented by Baah et al. in [3] and it gives out the conditional probability of each node to locate fault. Tarantula, which is proposed by Jones and Harrold in [4], indicates that a program entity executed by failed test cases should be suspected and they use different colors to represent the degree of suspiciousness. SOBER, which is proposed by Liu et al. in [5], uses the differences of predicated truth value between successful and failed execution to guide the activities of finding faults. CBT, namely, crosstab-based technique, is proposed by Wong et al. [6] to calculate the suspiciousness of each executable statement as the detected priority. Meanwhile, other fault localization techniques are presented in [7, 8], such as delta debugging and predicate switching. By modifying the variables or their values at a particular point during the execution to change program state, these techniques are able to identify the factor that triggers the program failure. Specifically, Zeller and Hildebrandt [7] propose the delta debugging to reduce the factors triggering the failures to a small set of variables by comparing the program state difference between the failed test case and successful test case. Predicate switching, which is presented by Zhang et al. [8], alters the program state of failed execution by changing the predicate. The predicate is labeled as critical predicate if its switch can make program execute successfully.

Being robust and widely applied, many machine learning and data mining techniques have been adopted to facilitate the fault localization in recent years [9]. Wong and Qi propose a backpropagation (BP) neural network in [10], which utilizes the coverage data of test cases (e.g., the coverage data with respect to which statements are executed by which test case) and their corresponding execution results to train the network and, then, input the coverage of a set of virtual test cases (e.g., each test case covers only one statement) to the trained network, and the outputs are regarded as the suspiciousness (i.e., likelihood of containing the bug) of each executable statement. But the BP neural network leads to problems like local minima. Aiming to solve this problem, Wong et al. propose another fault localization method based on radial basis function (RBF) network in [11], which is less sensitive to those problems. Although machine learning techniques exhibit good performance in the field of fault localization, the models they used for fault location are all shallow architecture. With shortcomings like limited ability to express complex function under limited amount of sample data as well as restricted generalization ability for intricate problem, the faults cannot be analyzed exactly via those methods.

This paper proposes a fault localization method based on deep neural network (DNN). With the capability of estimating complicated functions by learning a deep nonlinear network structure and further attaining distributed representation of input data, this method exhibits strong ability to learn representation from minority sample data. Moreover, DNN is one of the deep learning models that has been successfully applied in many other areas of software engineering [12, 13]. For example, the researchers in Microsoft Research adopt DNN to decrease the error rate of speech recognition in [12], which is one of the greatest breakthroughs in that field in the recent ten years. We use the Siemens suite and Space program as platforms to evaluate and demonstrate the effectiveness of DNN-based fault localization technique. The remainder of this paper is organized as follows. Section 2 provides an introduction of some related studies. Then, in Section 3, we elaborate our DNN-based fault localization method and provide a toy example to help readers understand this method. We conduct empirical studies on two suites (i.e., Siemens suite and Space program) in Section 4. Then, Section 5 follows where we report the result of the performance (effectiveness) comparison between our method and others and make a discussion. Section 6 lists possible threats to the validity of our approach. We present our conclusion and future work in Section 7.

2. Related Work

Recent years have witnessed the successful application of machine learning techniques in the field of fault localization. However, many models with shallow architectures encounter drawbacks like their restricted ability to express complex function under limited amount of sample data, such as BP neural network and support vector machine. And the generalization ability for intricate problem is also restrained. With the rapid development of the deep learning, many researchers begin to adopt deep neural networks to tackle the limitations of shallow architectures gradually. Before presenting our DNN-based fault localization technique, we introduce some related work that contributes to our novel approach.

2.1. Deep Neural Network Model. In 2006, deep neural network is firstly presented by Hinton et al. in the journal Science [14]. The rationale of DNN is that the neural network model is firstly divided into a number of two-layer models before we learn the whole model, and then we train the two-layer neural network model layer by layer and finally get the initial weights of multilayer neural networks by composing the trained two-layer neural networks, the whole process of which is called layerwise pretraining [15]. The hidden layer of neural network can extract features from the input layer due to its abstraction. Thus, the neural networks with multiple hidden layers are better at network processing and network generalization and achieve faster convergence rate. We elaborate on the theory of deep neural networks which is cited as the basis of our technique here.

DNN is a sort of feed-forward artificial neural network with multiple hidden layers, and each node at the same hidden layer can use the same nonlinear function to map the feature input from the layer below to the current node. DNN structure is very flexible due to the multiple hidden layers and multiple hidden nodes, so DNN demonstrates excellent capacity to fit the highly complex nonlinear relationship between inputs and outputs.

Generally, DNN model can be utilized for regression or classification. In this paper, the model is considered to be a classification model, but in order to output continuous suspiciousness values, we do not normalize the incentive value of DNN's last layer into integer. The relationship between inputs and outputs in DNN model can be interpreted as follows:

$$\begin{aligned} v^0 &= \text{input}, \\ v^{l+1} &= \rho\left(z^l\left(v^l\right)\right), \\ z^l\left(v^l\right) &= w^l\left(v^l\right) + b^l, \quad 0 \leqslant l < L, \\ \text{output} &= v^L. \end{aligned} \tag{1}$$

According to the above formulas, we can obtain the final output by transforming the features vector of the first layer v^0 into a processed feature vector v^l through L layers of nonlinear transformation. During the training process of DNN model, we need to determine the weight matrix w^l and offset vector b^l of lth layer. By utilizing the difference between the target outputs and the actual outputs to construct a cost function, we can then train the DNN by backpropagation (BP) algorithm.

Mainly, the design of DNN model includes steps like designing the number of network layers, the number of nodes in each layer, the transfer function between the layers, and so forth.

2.1.1. The Design of the Network Layer. Generally, deep neural network consists of three parts: the input layer, hidden layer, and output layer. The study of the neural network layers mainly aims to identify the number of hidden layers to determine the number of layers of a network. In neural networks, hidden layer has an effect of abstraction and can extract features from the input. At the same time, the number of hidden layers directly determines the processing ability of network to extract feature. If the number of hidden layers is too small, it cannot represent and fit the intricate problems with limited amount of sample data, while, with increased number of hidden layers, the processing ability of the entire network will improve. However, too many hidden layers also produce some adverse effects, such as the increased complexity of calculation and local minima. Thus, we draw a conclusion that too many or too few hidden layers are all unfavorable for the network training. We must choose the appropriate number of network layers to adapt to the different practical problems.

2.1.2. The Design of Number of Nodes. Usually, the nodes of input layer and output layer can be determined directly when the training samples of deep neural network are confirmed; thus, determining the number of nodes in the hidden layer is most critical. If we set up too many hidden nodes, it will increase the training time and train the neural network excessively (i.e., remember some unnecessary information) and lead to problems like overfitting, whereas the network is not able to handle complex problem as its ability to obtain information gets poorer with too few hidden nodes. The number of hidden nodes has a direct relationship with inputs and outputs, and it needs to be determined according to several experiments.

2.1.3. The Design of Transfer Function. Transfer function reflects the complex relationship between the input and its output. Different problems are suited to different transfer functions. In our work, we use the common sigmoid function as a nonlinear transfer function:

$$\rho(s) = \frac{1}{1+e^{-s}}. \tag{2}$$

Here, s is the input and $\rho(s)$ is the output.

2.1.4. The Design of Learning Rate and Impulse Factor. The learning rate and impulse factor have an important impact on DNN as well. A smaller learning rate will increase training time and slow down the speed of convergence; on the other hand, it generates network shocks further. For the choice of impulse factor, we can use its "inertia" to cushion the network shocks. We choose the appropriate learning rate and impulse factor according to the size of the sample.

2.2. Learning Algorithm of DNN. There exist various pretraining methods for DNN and mainly they are divided into two categories, namely, the unsupervised pretraining and supervised pretraining.

2.2.1. Unsupervised Pretraining. For the unsupervised pretraining, it usually adopts Restricted Boltzmann Machine (RBM) to initialize a DNN. RBM is proposed by Hinton in 2010 [16]. The structure of a RBM is depicted in Figure 1. The RBM is divided into two layers: the first one is visible layer with visible units and the second one is hidden layer with hidden units. Once the unsupervised pretraining has been

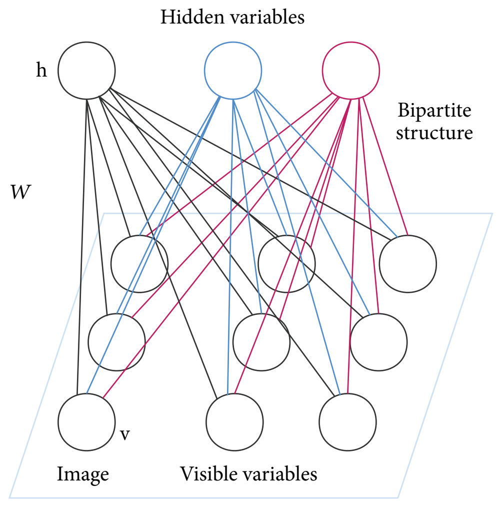

Figure 1: Architecture of a Restricted Boltzmann Machine.

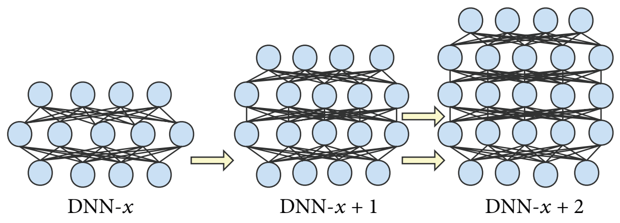

Figure 2: Architecture of supervised pretraining.

completed, the obtained parameters will be used as the initial weights of the DNN.

2.2.2. Supervised Pretraining. In order to reduce the inaccuracy of unsupervised training, we adopt the supervised pretraining proposed in [17]. The general architecture is shown in Figure 2. It works as follows: the BP algorithm is utilized to train a neural network, and the topmost layer is replaced by a randomly initialized hidden layer and a new random topmost layer after every pretraining. The network is unceasingly trained again until convergence or reaching the expected number of hidden layers.

In our work, the number of nodes in the input layer of DNN model is equal to the dimension of the input feature vector. The output layer has only one output node, that is, suspiciousness value. After a distinctive pretraining, the BP algorithm is used to fine-tune the parameters of the model. Here, $y_{1:T}$ is the training sample, and the goal is to minimize the squared error sum between the training samples $y_{1:T}$ and labeling $x_{1:T}$. The objective function can be interpreted as follows:

$$E(W,b) = \frac{1}{2}\sum_{t}\left\| f(y_t, W, b) - x_t \right\|^2. \tag{3}$$

Suppose that layer index is $0, 1, \ldots, L$, node layer is represented by $v_0, v_1, \ldots, w_0, w_1, \ldots, w_{L-1}$ denotes the weight layer, and k_L is the index of nodes in the Lth layer and then take the derivative of the weight matrix W^{L-1} and offset vector b^{L-1}:

$$\begin{aligned} E(W,b) &= \frac{1}{2}\sum_t \left\|\rho\left(z^{L-1} - x_t\right)\right\| \\ &= \frac{1}{2}\sum_t\sum_{k_L}\left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)^2 \\ &= \frac{1}{2}\sum_t\sum_{k_L}\left(\rho\left(\sum_{k_{L-1}} w_{k_L k_{L-1}}^{L-1} v_{k_{L-1}}^{L-1} + b_{k_L}^{L-1}\right) - x_{t,k_L}\right)^2. \end{aligned} \tag{4}$$

Take the derivative for each component of W^{L-1} and b^{L-1}:

$$\begin{aligned} \frac{\partial E(W,b)}{\partial W_{k_L k_{L-1}}^{L-1}} &= \sum_t \left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)\rho'\left(z_{k_L}^{L-1}\right)\left(v_{k_{L-1}}^{L-1}\right), \\ \frac{\partial E(W,b)}{\partial b_{k_L}^{L-1}} &= \sum_t \left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)\rho'\left(z_{k_L}^{L-1}\right). \end{aligned} \tag{5}$$

Then, take the derivative of the weight matrix W^{L-2} and offset vector b^{L-2}:

$$\begin{aligned} E(W,b) &= \frac{1}{2}\sum_t \left\|\rho\left(z^{L-1} - x_t\right)\right\| = \frac{1}{2}\sum_t\sum_{k_L}\left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)^2 = \frac{1}{2}\sum_t\sum_{k_L}\left(\rho\left(\sum_{k_{L-1}} w_{k_L k_{L-1}}^{L-1} v_{k_{L-1}}^{L-1} + b_{k_L}^{L-1}\right) - x_{t,k_L}\right)^2 \\ &= \frac{1}{2}\sum_t\sum_{k_L}\left(\rho\left(\sum_{k_{L-1}} w_{k_L k_{L-1}}^{L-1}\rho\left(z_{k_L}^{L-1}\right) + b_{k_L}^{L-1} - x_{t,k_L}\right)^2\right) \\ &= \frac{1}{2}\sum_t\sum_{k_L}\left(\rho\left(\sum_{k_{L-1}} w_{k_L k_{L-1}}^{L-1}\rho\left(\sum_{k_{L-2}} W_{k_{L-1}k_{L-2}}^{L-2} v_{k_{L-2}}^{L-2} + b_{k_{L-1}}^{L-2}\right) + b_{k_L}^{L-1} - x_{t,k_L}\right)^2\right). \end{aligned} \tag{6}$$

Take the derivative for each component of W^{L-2} and b^{L-2}:

$$\begin{aligned} &\frac{\partial E(W,b)}{\partial W_{k_{L-1}k_{L-2}}^{L-2}} \\ &\quad = \sum_t\sum_{k_L}\left[\left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)\rho'\left(z_{k_L}^{L-1}\right)W_{k_L k_{L-1}}^{L-1}\right] \\ &\quad \cdot \rho'\left(z_{k_{L-1}}^{L-2}\right)v_{k_{L-2}}^{L-2}, \\ &\frac{\partial E(W,b)}{\partial b_{k_{L-1}}^{L-2}} \\ &\quad = \sum_t\sum_{k_L}\left[\left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)\rho'\left(z_{k_L}^{L-1}\right)W_{k_L k_{L-1}}^{L-1}\right] \\ &\quad \cdot \rho'\left(z_{k_{L-1}}^{L-2}\right). \end{aligned} \tag{7}$$

The result is as follows:

$$\begin{aligned} \frac{\partial E(W,b)}{\partial W^l} &= \sum_t e^{l+1}(t)\left(v^l(t)\right)^T, \\ \frac{\partial E(W,b)}{\partial b^l} &= \sum_t e^{l+1}(t). \end{aligned} \tag{8}$$

Here,

$$\begin{aligned} e_{k_L}^L(t) &= \left(\rho\left(z_{k_L}^{L-1}\right) - x_{t,k_L}\right)\rho'\left(z_{k_L}^{L-1}\right), \\ e_{k_{L-1}}^{L-1}(t) &= \sum_{k_L} e_{k_L}^L(t)\,W_{k_L k_{L-1}}^{L-1}\rho'\left(z_{k_{L-1}}^{L-2}\right). \end{aligned} \tag{9}$$

After vectorization,

$$\begin{aligned} e^L(t) &= \operatorname{diag}\left(\rho'\left(z_{k_L}^{L-1}\right)\right)\left(\rho\left(z^{L-1}\right) - x_t\right), \\ e_k^L(t) &= \operatorname{diag}\left(\rho'\left(z_{k_{L-1}}^{L-2}\right)\right)\left(W^{L-1}\right)^T e^L(t). \end{aligned} \tag{10}$$

The recursive process is as follows:

$$\begin{aligned} e^L(t) &= Q^L(t)\left(f\left(y_t, (W,b)^0\right) - x_t\right), \\ e^l(t) &= Q^L(t)\left(W^l\right)^T e^{l+1}(t), \\ Q^l(t) &= \operatorname{diag}\left(\rho'\left(z^{l-1}\left(v^{l-1}(t)\right)\right)\right), \\ \rho'(z) &= \rho(z)\cdot(1 - \rho(z)). \end{aligned} \tag{11}$$

In order to calculate the error between the actual output and the expected output, we output samples $y_{1:T}$ to DNN and then execute the forward process of DNN. Meanwhile, we calculate the outputs of all the hidden layer nodes and output nodes, and now we can compute to get the error $e^L(t)$. Next, backpropagation procedure is executed and error of the nodes on each hidden layer is calculated iteratively after obtaining $e^L(t)$. The parameters of DNN can be updated layer by layer according to the following formula:

$$\begin{aligned} \left(W^l, b^l\right)^{m+1} &= \left(W^l, b^l\right)^m + \Delta\left(W^l, b^l\right)^m, \quad 0 \leqslant l \leqslant L, \\ \Delta\left(W^l, b^l\right)^m &= (1-\alpha)\,\varepsilon\left(\frac{\partial E}{\partial\left(W^l, b^l\right)}\right) \\ &\quad + \alpha\Delta\left(W^l, b^l\right)^{m-1}, \quad 0 \leqslant l \leqslant L. \end{aligned} \tag{12}$$

Table 1: The coverage data and execution result of test case.

	s_1	s_2	s_3	$\cdots$	s_{m-1}	s_m	R
t_1	1	0	1	$\cdots$	0	1	0
t_2	1	1	1	$\cdots$	0	1	0
t_3	1	0	1	$\cdots$	1	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
t_{n-1}	1	0	0	$\cdots$	1	0	1
t_n	1	1	1	$\cdots$	0	0	0

Here, ε is the learning rate, α is the impulse factor, and m denotes the mth iteration. In the process of fault localization, we input the virtual test matrix Y into the DNN model and then execute the forward process of DNN, and finally the output is the suspiciousness value of each statement.

3. DNN-Based Fault Localization Technique

While the previous section provides an overview of the DNN model, this part will present the methodology of DNN-based fault localization technique in detail. With a focus on how to build a DNN model for fault localization problem, meanwhile, we will summarize the procedures of localizing faults using DNN and provide a concrete example for demonstration at the end of this part.

3.1. Fault Localization Algorithm Based on Deep Neural Network. Suppose that there is a program P with m executable statements and n test cases to be executed, t_i denotes the ith test case, while vectors c_{t_i} and r_{t_i} represent the corresponding coverage data and execution result, respectively, after executing the test case t_i, and s_j is jth executable statement of program P. Here, $c_{t_i} = [(c_{t_i})_1, (c_{t_i})_2, \ldots, (c_{t_i})_m]$. If the executable statement s_j is covered by t_j, we assign a real value 1 to $(c_{t_i})_j$; otherwise, we assign 0 to it. If the test case t_i is executed successfully, we assign a real value 0 to r_{t_i}; otherwise, it is assigned with 1. We depict the coverage data and execution result of test case through Table 1. For instance, from Table 1, we can draw a conclusion that s_2 is not covered with failed test case t_3, while s_3 is covered with failed test case t_2 according to Table 1.

The network can reflect the complex nonlinear relationship between the coverage data and execution result of test case when the training of deep neural network is completed. We can identify the suspicious code of faulty version by trained network. At the same time, constructing a set of virtual test cases is necessary, as shown in the following equation.

The Virtual Test Set. Consider

$$\begin{bmatrix} c_{v_1} \\ c_{v_2} \\ \vdots \\ c_{v_m} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}; \qquad (13)$$

the vector $c_{v_1}, c_{v_2}, \ldots, c_{v_m}$ denotes the coverage data of test cases $v_1, v_2, \ldots, v_m$.

In the set of virtual test cases, the test case v_j only covers executable statement s_j. Since each test case only covers one statement, thus the statement is highly suspicious if the corresponding test case is failed. For example, s_j is very likely to contain bugs with a failed test case v_j, and that means we should preferentially check the statements which are covered by failed test cases. But such set of test cases does not exist in reality; thus, the execution result of test case in the virtual test set is difficult to get in practice. So we input c_{v_j} to the trained DNN, and the output r_{v_j} represents the probability of test case execution result. The value of r_{v_j} is proportional to the suspicious degree of containing bugs of s_j.

Specific fault location algorithm is as follows:

(1) Construct the DNN model with one input layer and one output layer, and identify the appropriate number of hidden layers according to the experiment scale. Suppose the number of input layer nodes is m, the number of hidden layer nodes is n, the number of output layer nodes is 1, and the adopted transfer function is sigmoid function $\rho(s) = 1/1 + e^{-s}$.

(2) Utilize the coverage data and execution result of test case as training sample set which is inputted into the DNN model, and then train the DNN model to obtain the complex nonlinear mapping relationship between the coverage data c_{t_i} and execution result r_{t_i}.

(3) Input the coverage data vector of virtual test set c_{v_j} $(1 \leqslant j \leqslant m)$ into the DNN model and get the output r_{v_j} $(1 \leqslant j \leqslant m)$.

(4) The output r_{v_j} reflects the probability that executable statement s_j contains the bugs, that is, suspiciousness value, and then conduct descending ranking for r_{v_j}.

(5) Rank s_j $(1 \leqslant j \leqslant m)$ according to their corresponding suspiciousness value and check the statement one by one from the most suspicious to the least until the faults are finally located.

3.2. The Example of Fault Localization Algorithm Based on Deep Neural Network. Here, we illustrate the application of fault localization analysis based on deep neural network by a concrete example. It is depicted in Table 2.

Table 2 shows that the function of program Mid(·) is to get the middle number by comparing three integers. The program has twelve statements and ten test cases. There is one fault in the program, which is contained by statement (6). Here, "•" denotes that the statement is covered by the corresponding test case. The blank denotes that the statement is not covered by test case. P represents that the test case is executed successfully while F represents that the execution results of test case are failed.

The coverage data and execution results of test cases are shown in Table 2. Table 3 correlates with Table 2. Here, number 1 replaces "•" and indicates that the statement is covered by the corresponding test case, and number 0

Table 2: Coverage and execution result of Mid function.

Function	Test cases									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Mid(x, y, z) {int m;	3, 3, 5	1, 2, 3	3, 2, 2	5, 5, 5	1, 1, 4	5, 3, 4	3, 2, 1	5, 4, 2	2, 1, 3	5, 2, 6
(1) $m = z$	•	•	•	•	•	•	•	•	•	•
(2) if $(y < z)$	•	•	•	•	•	•	•	•	•	•
(3) if $(x < y)$	•	•			•	•			•	•
(4) $m = y$		•								
(5) else if $(x < z)$	•				•	•			•	•
(6) $m = y$ //bug	•				•				•	•
(7) else			•	•			•	•		
(8) if $(x > y)$			•	•			•	•		
(9) $m = y$			•				•	•		
(10) else if $(x > z)$				•						
(11) $m = x$										
(12) printf (m);}	•	•	•	•	•	•	•	•	•	•
Pass/fail	P	P	P	P	P	P	P	P	F	F

Table 3: Switched Table 2.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	r
t_1	1	1	1	0	1	1	0	0	0	0	0	1	0
t_2	1	1	1	1	0	0	0	0	0	0	0	1	0
t_3	1	1	0	0	0	0	1	1	1	0	0	1	0
t_4	1	1	0	0	0	0	1	1	0	1	0	1	0
t_5	1	1	1	0	1	1	0	0	0	0	0	1	0
t_6	1	1	1	0	1	0	0	0	0	0	0	1	0
t_7	1	1	0	0	0	0	1	1	1	0	0	1	0
t_8	1	1	0	0	0	0	1	1	1	0	0	1	0
t_9	1	1	1	0	1	1	0	0	0	0	0	1	1
t_{10}	1	1	1	0	1	1	0	0	0	0	0	1	1

replaces the blank and represents that the statement is not covered by the corresponding test case. In the last column, number 1 replaces F and indicates that the execution results of test case are failed, while number 0 replaces P and represents that the test case is executed successfully.

The concrete fault localization process is as follows:

(1) Construct the DNN model with one input layer, one output layer, and three hidden layers. The number of input layer nodes is 12, and the number of hidden layer nodes is simply set as 4. The number of output layer nodes is 1, and the transfer function adopted is the sigmoid function $\rho(s) = 1/1 + e^{-s}$.

(2) Use the coverage data and execution result of test case as training sample set to input into the DNN model. First, we input the vector $(1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1)$ and its execution result 0 and, next, input the second vector $(1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1)$ and its execution result 0 until the coverage data and execution results of the ten test cases are all inputted into the network. We then train the DNN model utilizing the inputted training data, and further we input the next training dataset (i.e., another ten test cases' coverage data and execution results) iteratively to optimize the DNN model until we reach the condition of convergence. The final DNN model we obtained after several times' iteration reveals the complex nonlinear mapping relationship between the coverage data and execution result.

(3) Construct the virtual test set with twelve test cases and ensure that each test case only covers one executable statement. The set of virtual test cases is depicted in Table 4.

(4) Input the virtual test set into the trained DNN model and get the suspiciousness value of each corresponding executable statement, and then rank the statements according to their suspiciousness values.

(5) Table 5 shows the descending ranking of statements according to their suspiciousness value. According to the table of ranking list, we find that the suspiciousness value of the sixth statement which contains the bug in program Mid$(\cdot)$ is the greatest (i.e., rank first); thus, we can locate the fault by checking only one statement.

4. Empirical Studies

So far, the modeling procedures of our DNN-based fault localization technique have been discussed in the previous section. In this section, we empirically compare the performance of this technique with that of Tarantula localization technique [4], PPDG localization technique [3], and BPNN localization technique [10] on the Siemens suite and Space program to demonstrate the effectiveness of our DNN-based fault localization method. The Siemens suite and Space

Table 4: Virtual test suite.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}
t_1	1	0	0	0	0	0	0	0	0	0	0	0
t_2	0	1	0	0	0	0	0	0	0	0	0	0
t_3	0	0	1	0	0	0	0	0	0	0	0	0
t_4	0	0	0	1	0	0	0	0	0	0	0	0
t_5	0	0	0	0	1	0	0	0	0	0	0	0
t_6	0	0	0	0	0	1	0	0	0	0	0	0
t_7	0	0	0	0	0	0	1	0	0	0	0	0
t_8	0	0	0	0	0	0	0	1	0	0	0	0
t_9	0	0	0	0	0	0	0	0	1	0	0	0
t_{10}	0	0	0	0	0	0	0	0	0	1	0	0
t_{11}	0	0	0	0	0	0	0	0	0	0	1	0
t_{12}	0	0	0	0	0	0	0	0	0	0	0	1

Table 5

Statement	Suspiciousness	Rank
1	0.0448	11
2	0.1233	2
3	0.0879	5
4	0.0937	4
5	0.0747	7
6 (fault)	0.1593	1
7	0.1191	3
8	0.0758	6
9	0.0661	8
10	0.0436	12
11	0.0650	9
12	0.0649	10

program can be downloaded from [18]. The evaluation standard based on statements ranking is proposed by Jones and Harrold in [4] to compare the effectiveness of the Tarantula localization technique with other localization techniques. Tarantula localization technique can produce the suspiciousness ranking of statements. The programmers examine the statements one by one from the first to the last until the fault is located and the percentage of statements without being examined is defined as the score of fault localization technique, that is, EXAM score. Since the DNN-based fault localization technique, Tarantula localization technique, and PPDG localization technique all produce a suspiciousness value ranking list for executable statements, we adopt the EXAM score as the evaluation criterion to measure the effectiveness.

4.1. Data Collection. The physical environment on which our experiments were carried out included 3.40 GHz Intel Core i7-3770 CPU and 16 GB physical memory. The operating systems were Windows 7 and Ubuntu 12.10. Our compiler was gcc 4.7.2. We conducted experiments on the MATLAB R2013a. To collect the coverage data of each statement, toolbox software named Gcov was used here. The test case execution results of faulty program versions can be obtained by the following method:

(1) We run the test cases on fault-free program versions to get the execution results.

(2) We run the test cases on faulty program versions to get the execution results.

(3) We compare the results of fault-free program versions and results of faulty program versions. If they are equal, the test case is successful; otherwise, it is failed.

4.2. Programs of the Test Suite. In our experiment, we conducted two studies on the Siemens suite and Space program to demonstrate the effectiveness of the DNN-based fault localization technique.

4.2.1. The Siemens Suite. The Siemens suite has been considered as a classical test sample which is widely employed in studies of fault localization techniques. And, in this paper, our experiment begins with it. The Siemens suite contains seven C programs with the corresponding faulty versions and test cases. Each faulty version has only one fault, but it may cross the multiline statements or even multiple program functions in the program. Table 6 provides the detailed introduction of the seven C programs in Siemens suite, including the program name, number of faulty versions of each program, number of statement lines, number of executable statements lines, and the number of test cases.

The function of the seven C programs is different. The Print_tokens and Print_tokens 2 programs are used for lexical analysis, and the function of Replace program is pattern replacement. The Schedule and Schedule 2 programs are utilized for priority scheduler while the function of Tcas is altitude separation, and the Tot_info program is used for information measure.

The Siemens suite contains 132 faulty versions, while we choose 122 faulty versions to perform experiments. We omitted the following versions: versions 4 and 6 of Print_tokens, version 9 of Schedule 2, version 12 of Replace, versions 13, 14, and 36 of Tcas, and versions 6, 9, and 21 of Tot_info. We eliminated these versions because of the following:

(a) There is no syntactic difference between the correct version and the faulty versions (e.g., only difference in the header file).

(b) The test cases never fail when executing the programs of faulty versions.

(c) The faulty versions have segmentation faults when executing the test cases.

(d) The difference between the correct versions and the faulty versions is not included in executable statements of program and cannot be replaced. In addition, there exists absence of statements in some faulty versions and we cannot locate the missing statements directly. In this case, we can only select the associated statements to locate the faults.

Table 6: Summary of the Siemens suite.

Program	Number of faulty versions	LOC (lines of code)	Number of executable statements	Number of test cases
Print_tokens	7	565	172	4130
Print_tokens 2	10	510	146	4115
Replace	32	563	175	5542
Schedule	9	412	140	2650
Schedule 2	10	307	115	2710
Tcas	41	173	59	1608
Tot_info	23	406	100	1052

4.2.2. The Space Program. As the executable statements of Siemens suite programs are only about hundreds of lines, the scale and complexity of the software have increased gradually and the program may be of as many as thousands or ten thousands of lines. So we verify the effectiveness of our DNN-based fault localization technique on large-scale program set, that is, Space program.

The Space program contains 38 faulty versions; each version has more than 9000 statements and 13585 test cases. In our experiment, due to similar reasons which have been depicted in Siemens suite, we also omit 5 faulty versions and select another 33 faulty versions.

4.3. Experimental Process. In this paper, we conducted experiments on the 122 faulty versions of Siemens suite and 33 faulty versions of Space program. For the DNN modeling process, we adopted three hidden layers; that is, the structure of the network is composed of one input layer, three hidden layers, and one output layer, which is based on previous experience and the preliminary experiment. According to the experience and sample size, we estimate the number of hidden layer nodes (i.e., num) by the following formula:

$$\text{num} = \text{round}\left(\frac{n}{30}\right) * 10. \tag{14}$$

Here, n represents the number of input layer nodes. The impulse factor is set as 0.9, and the range of learning rate is $\{0.01, 0.001, 0.0001, 0.00001\}$; we select the most appropriate learning rate as the final parameter according to the sample scale.

Now, we take Print_tokens program as an example and introduce the experimental process in detail. The other program versions can refer to the Print_tokens. The Print_tokens program includes seven faulty versions and we pick out five versions to conduct the experiment. The experiment process is as follows:

(1) Firstly, we compile the source program of Print_tokens and run the test case set to get the expected execution results of test cases.

(2) We compile the five faulty versions of Print_tokens program using GCC and similarly get the actual execution results of test cases and acquire the coverage data of test cases by using Gcov technique.

(3) We construct the set of virtual test cases for the five faulty versions of Print_tokens program, respectively.

(4) Then, we compare the execution results of test cases between the source program and the five faulty versions of Print_tokens to get the final execution results of the test cases of five faulty versions.

(5) We integrate the coverage data and the final results of test cases of the five faulty versions, respectively, as input vectors to train the deep neural network.

(6) Finally, we utilize the set of virtual test cases to test the deep neural network to acquire the suspiciousness of the corresponding statements and then rank the suspiciousness value list.

5. Results and Analysis

According to the experimental process described in the previous section, we perform the fault localization experiments on deep neural network and BP neural network and statistically analyze the experimental results. We then compare the experimental results with Tarantula localization technique and PPDG localization technique. The experimental results of Tarantula and PPDG have been introduced in [3, 4], respectively.

5.1. The Experimental Result of Siemens Suite. Figure 3 shows the effectiveness of DNN, BP neural network, Tarantula, and PPDG localization techniques on the Siemens suite.

In Figure 3, the x-axis represents the percentage of statements without being examined until the fault is located, that is, EXAM score, while the y-axis represents the percentage of faulty versions whose faults have been located. For instance, there is a point labeled on the curve whose value is (70, 80). This means the technique is able to find out eighty percent of the faults in the Siemens suite with seventy percent of statements not being examined; that is, we can identify eighty percent of the faults in the Siemens suite by only examining thirty percent of the statements.

Table 7 offers further explanation for Figure 3.

According to the approach of experiment statistics from other fault localization techniques, the EXAM score can be divided into 11 segments. Each 10% is treated as one segment, but the caveat is that the programmers are not able to identify

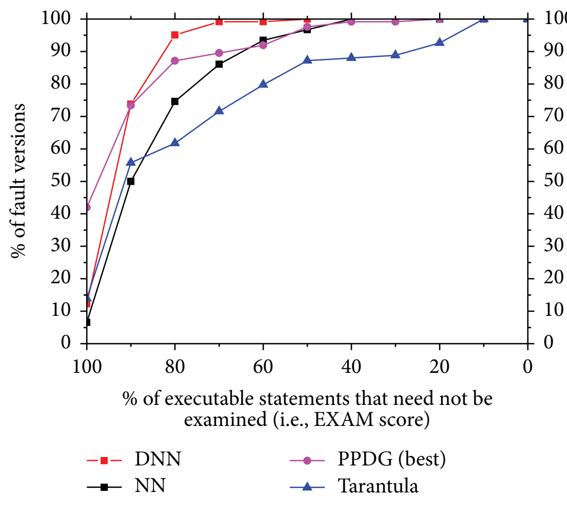

Figure 3: Effectiveness comparison on the Siemens suite.

Table 7: Effectiveness comparison of four techniques.

EXAM score	% of the faulty versions			
	DNN	BPNN	PPDG	Tarantula
100%-99%	12.29%	6.56%	41.94%	13.93%
99%–90%	61.48%	43.44%	31.45%	41.80%
90%–80%	21.31%	24.59%	13.71%	5.74%
80%–70%	4.10%	11.48%	2.42%	9.84%
70%–60%	0.00%	7.38%	2.42%	8.20%
60%–50%	0.82%	3.28%	5.65%	7.38%
50%–40%	0.00%	3.28%	1.61%	0.82%
40%–30%	0.00%	0.00%	0.00%	0.82%
30%–20%	0.00%	0.00%	0.80%	4.10%
20%–10%	0.00%	0.00%	0.00%	7.38%
10%–0%	0.00%	0.00%	0.00%	0.00%

the faults without examining any statement; thus, the abscissa can only be infinitely close to 100%. Due to that factor, we divide the 100%–90% into two segments, that is, 100%–99% and 99%–90%. The data of every segment can be used to evaluate the effectiveness of fault localization technique. For example, for the 99%–90% segment of DNN, the percentage of faults identified in the Siemens suite is 61.48% while the percentage of faults located is 43.44% for the BPNN technique. From Table 7, we can find that, for the segments 50%–40%, 40%–30%, 30%–20%, 20%–10%, and 10%–0% of DNN, the percentage of faults identified in each segment is 0.00%. That indicates that the DNN-based method has found out all the faulty versions after examining 50% of statements.

We analyzed the above data and summarized the following points:

(1) Figure 3 reveals that the overall effectiveness of our DNN-based fault localization technique is better than that of the BP neural network fault localization technique as the curve representing DNN-based approach is always above the curve denoting BPNN-based method. Also, the DNN-based technique is able to identify the faults by examining fewer statements compared with the method based on BP neural network. For instance, by examining 10% of statements, the DNN-based method is able to identify 73.77% of the faulty versions while the BP neural network fault localization technique only finds out 50% of the faulty versions.

(2) Compared with the Tarantula fault localization technique, the DNN-based approach dramatically improves the effectiveness of fault localization on the whole. For example, the DNN-based method is capable of finding out 95.08% of the faulty versions after examining 20% of statements while the Tarantula fault localization technique only identifies 61.47% of the faulty versions after examining the same amount of statements. For the score segment of 100%–99%, the effectiveness of the DNN-based fault localization technique is relatively similar to that of Tarantula; for instance, DNN-based method can find out 12.29% of the faulty versions with 99% of statements not examined (i.e., the abscissa values are 99), and its performance is close to that of Tarantula fault localization technique.

(3) Compared with the PPDG fault localization technique, the DNN-based fault localization technique reflects improved effectiveness as well for the score range of 90%–0%. Moreover, the DNN-based fault localization technique is capable of finding out all faulty versions by only examining 50% of statements while PPDG fault localization technique needs to examine 80% of statements. For the score range of 100%–99%, the performance of DNN-based fault localization technique is inferior to PPDG fault localization technique.

In conclusion, overall, the DNN-based fault localization technique improves the effectiveness of localization considerably. In particular, for the score range of 90%–0%, the improved effectiveness of localization is more obvious compared with other fault localization techniques as it needs less statements to be examined to further identify the faults. The DNN-based fault localization technique is able to find out all faulty versions by only examining 50% of statements, which is superior to BPNN, Tarantula, and PPDG fault localization technique, while, for the score segment of 100%–99%, the performance of our DNN-based approach is similar to that of Tarantula fault localization method while it is a bit inferior to that of PPDG fault localization technique.

5.2. The Experimental Result of Space Program. As the executable statements of Siemens suite programs are only about hundreds of lines, we further verify the effectiveness of DNN-based fault localization technique in large-scale datasets.

Figure 4 demonstrates the effectiveness of DNN and BP neural network localization techniques on the Space program.

From Figure 4, we can clearly find that the effectiveness of our DNN-based method is obviously higher than that

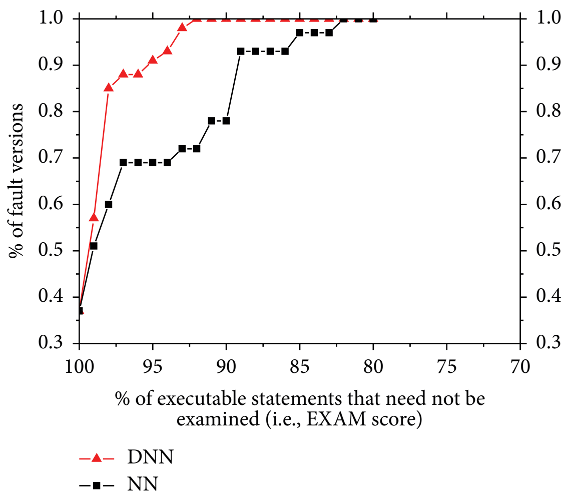

Figure 4: Effectiveness comparison on the Space program.

of the BP neural network fault localization technique. The DNN-based fault localization technique is able to find out all faulty versions without examining 93% of statements while BP neural network identifies all faulty versions with 83% of the statements not examined. That comparison indicates that the performance of our DNN-based approach is superior as it reduces the number of statements to be examined. The experimental results show that DNN-based fault localization technique is also highly effective in large-scale datasets.

6. Threats to the Validity

There may exist several threats to the validity of the technique presented in this paper. We discuss some of them in this section.

We empirically determine the structure of the network, including the number of hidden layers and the number of hidden nodes. In addition, we determine some important model parameters by grid search method. So probably there exist some better structures for fault localization model. As the DNN model we trained for the first time usually is not the most optimal one, thus empirically we need to modify the parameters of the network several times to try to obtain a more optimal model.

In the field of fault localization, we may encounter multiple-bug programs and we need to construct some new models that can locate multiple bugs. In addition, because the number of input nodes of the model is equal to the lines of executable statements, thus when the number of executable statements is very large, the model will become very big as well. That results in a limitation of fault localization for some big-scale software and we may need to control the scale of the model or to improve the performance of the computer we use. Moreover, virtual test sets built to calculate the suspiciousness of each test case which only covers one statement may not be suitably designed. When we use the virtual test case to describe certain statement, we assume that if a test case is predicated as failed, then the statement covered by it will have a bug. However, whether this assumption is reasonable has not been confirmed. As this assumption is important for our fault localization approach, thus we may need to test this hypothesis or to propose an alternative strategy.

7. Conclusion and Future Work

In this paper, we propose a DNN-based fault localization technique as DNN is able to simulate the complex nonlinear relationship between the input and the output. We conduct an empirical study on Siemens suite and Space program. Further, we compare the effectiveness of our method with other techniques like BP neural network, Tarantula, and PPDG. The results show that our DNN-based fault localization technique performs best. For example, for Space program, DNN-based fault localization technique only needs to examine 10% of statements to fully identify all faulty versions, while BP neural network fault localization technique needs to examine 20% of statements to find out all faulty versions.

As software fault localization is one of the most expensive, tedious, and time-consuming activities during the software testing process, thus it is of great significance for researchers to automate the localization process [19]. By leveraging the strong feature-learning ability of DNN, our approach helps to predict the likelihood of containing fault of each statement in a certain program. And that can guide programmers to the location of statements' faults, with minimal human intervention. As deep learning is widely applied and demonstrates good performance in research field of image processing, speech recognition, and natural language processing and in related industries as well [13], with the ever-increasing scale and complexity of software, we believe that our DNN-based method is of great potential to be applied in industry. It could help to improve the efficiency and effectiveness of the debugging process, boost the software development process, reduce the software maintenance cost, and so forth.

In our future work, we will apply deep neural network in multifaults localization to evaluate its effectiveness. Meanwhile, we will conduct further studies about the feasibility of applying other deep learning models (e.g., convolutional neural network, recurrent neural network) in the field of software fault localization.

Competing Interests

The authors declare that they have no competing interests.

References

[1] M. A. Alipour, "Automated fault localization techniques: a survey," Tech. Rep., Oregon State University, Corvallis, Ore, USA, 2012.

[2] W. Eric Wong and V. Debroy, "A survey of software fault localization," Tech. Rep. UTDCS-45-09, Department of Computer Science, The University of Texas at Dallas, 2009.

[3] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[4] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated*

Software Engineering (ASE '05), pp. 273–282, Long Beach, Calif, USA, November 2005.

[5] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–847, 2006.

[6] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST '08)*, pp. 42–51, Lillehammer, Norway, April 2008.

[7] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[8] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 272–281, Shanghai, China, May 2006.

[9] S. Ali, J. H. Andrews, T. Dhandapani et al., "Evaluating the accuracy of fault localization techniques," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pp. 76–87, IEEE Computer Society, Auckland, New Zealand, November 2009.

[10] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 4, pp. 573–597, 2009.

[11] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149–169, 2012.

[12] F. Seide, G. Li, and D. Yu, "Conversational speech transcription using context-dependent deep neural networks," in *Proceedings of the 12th Annual Conference of the International Speech Communication Association (INTERSPEECH '11)*, pp. 437–440, Florence, Italy, August 2011.

[13] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[15] A. Mohamed, G. Dahl, and G. Hinton, "Deep belief networks for phone recognition," in *Proceedings of the NIPS Workshop on Deep Learning for Speech Recognition and Related Applications*, Whistler, Canada, December 2009.

[16] G. E. Hinton, "A practical guide to training restricted boltzmann machines," Tech. Rep. UTML TR 2010-003, Department of Computer Science, University of Toronto, 2010.

[17] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on Audio, Speech and Language Processing*, vol. 20, no. 1, pp. 30–42, 2012.

[18] http://sir.unl.edu.

[19] D. Binldey, "Source code analysis: a road map," in *Proceedings of the Future of Software Engineering (FOSE '07)*, pp. 104–119, IEEE Compurer Society, Minneapolis, Minn, USA, May 2007.