Journal of Theoretical and Applied Information Technology
15th August 2016. Vol.90. No.1
© 2005 - 2016 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

EFFECTIVE SOFTWARE FAULT LOCALIZATION USING GA-RBF NEURAL NETWORK

[1]DR. R.P. MAHAPATRA, [2]ANURAG NEGI
[1]Prof., Head Of Department, Department Of Computer Science and Engineering, SRM University, Delhi NCR, INDIA
[2] M.Tech Student, Department Of Computer Science and Engineering, SRM University, Delhi NCR, INDIA
E-mail: [1]mahapatra.rp@gmail.com, [2]anurag.negi1729@gmail.com

ABSTRACT

This paper proposes application of GA-RBF Neural Network Algorithm in context of software fault localization. A neural network is trained on the basis of Code Coverage information of a test case and the corresponding execution result, successful execution or failure. The weights and structure of the RBF neural network is then optimized using Genetic algorithm. The hidden layer neurons number and connection weights are encoded using binary encoding and real encoding respectively. For further leaning, LMS (Least Mean Square) algorithm is used. A set of virtual test cases (each covering a single statement) is then given input to the trained and optimized network. The output of the network is considered to be "degree of suspiciousness" of the corresponding statement. Finally the statements are ranked on the basis of their corresponding degree of suspiciousness.

Keywords: *Software Fault Localization, GA-RBF Neural Network, Software Debugging, Genetic algorithm, Radial Basis Function.*

1. INTRODUCTION

Fault localization is the activity of identifying the exact locations of program faults. In program debugging, this is considered most tedious and time consuming activity. In larger and complex programs a high degree of time and effort is dedicated to the identification of a fault in the software. Automatic software fault localization techniques are used by programmers to find out the exact location of the fault in least amount of time. Major advantage of the software fault localization techniques is the identification of faults in complex programs with more accuracy and less effort. This paper proposes the application of GA-RBF neural network as fault localization technique. Using Genetic Algorithm for the optimization of a RBF neural network improves the operating efficiency in dealing with complex problems and also improves the precision of the recognition.

The ability to learn is one of the several advantage of Neural-Network based models over other comparable models. Neural Networks are considered to be more tolerant because of the information distribution among the weights on the connections. The capability of Neural Networks to adapt and re-train to deal with minor changes in the operating environment also makes neural networks more popular among researchers.

Radial Basis Function (RBF) is a three-layer feed-forward network with a single hidden layer. This structure can be trained to learn an input-output relationship based on a data set. In this paper, the statement coverage of a test case is passed as the input and the output is result (success or failure) of corresponding program execution. The network is further optimized using Genetic Algorithm ad finally a virtual test case with only one statement covered is used as an input to computer the degree of suspiciousness of the corresponding statement in terms of its likelihood of containing bugs. The statements can then be ranked in descending order of their suspiciousness, such that the statements can be examined one by one.

The framework is similar to [3], however the RBF- Neural Network is further optimized using Genetic algorithm in order to improve efficiency and precision of the technique.

2. GENETIC ALGORITHM AND RBF NEURAL NETWORK

2.1 Basic Theory Of Genetic Algorithm

In a Genetic Algorithm, a population of candidate (potential solution set) to an optimization problem is evolved towards better solutions. The population is composed of a certain number of encoded gene individuals, which is the entities with characteristic chromosome. The major setback in the construction of a Genetic Algorithm based approach is the solvable encoding method and design of genetic operator. The successful application of the GA is determined by the choice of genetic operator, usage of different encoding method and degree of understanding of problems to be solved.

Traditionally, solutions are represented in binary as strings of 0's and 1's, but other encodings are equally possible. The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation). The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

GA is an iterative procedure which tends to retain a candidate solution and sorts them in accordance to some indicator, generally referred as the fitness function, and uses genetic operators to compute it to produce a new generation of candidate solutions. The process is repeated until it meets some index of convergence.

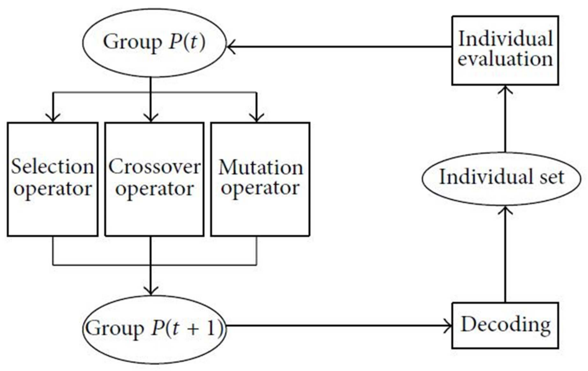


Figure 1 : The flow chart of genetic algorithm

2.2 Basic Theory Of RBF Neural Network

An RBF is a real-valued function whose value depends only on the distance from its receptive field center **μ** to the input **x**. It is a strictly positive radially symmetric function, where the center has the unique maximum, and the value drops off rapidly to zero away from the center. When the distance between **x** and **μ** (denoted as ||**x-μ**||) is smaller than the receptive field width **σ**, the function has an appreciable value.

A typical RBF neural network has a three-layer feed-forward structure. The first layer is the input layer, which passes inputs to the (second) hidden layer without changing their values. The hidden layer is where all neurons simultaneously receive the n-dimensional real-valued input vector. Each neuron in this layer uses an RBF as the activation function. We made use of the Gaussian basis function [1], as it is one of the most popular choices for employment in RBF networks [2].

$$R_j(\mathbf{x}) = \exp\left(\|\mathbf{x}-\boldsymbol{\mu}_j\|^2 / 2\sigma_j^2\right) \qquad 1)$$

Usually the distance in (1) is the Euclidean distance between **x** and **μ**, but in this paper we use a weighted bit-comparison based dissimilarity, and to make the distinction, we use ||x-μ|| to represent a *generic* distance, and $\|x-\mu\|_{WBC}$ for the weighted bit-comparison-based dissimilarity, same as [3]. The third layer is the output layer. The output can be expressed as $\mathbf{y} = [y_1, y_2, y_3, \ldots, y_k]$ with y_i as the output of the *i*th neuron given by:

$$y_i = {}^{h}\textstyle\sum_{j=1} w_{ji} R_j(x) \text{ for } i=[1,k] \qquad 2)$$

An RBF network implements a mapping from the dimensional real-valued input space to the dimensional real-valued output space with a hidden layer space in between. The transformation from the input space to the hidden-layer space is nonlinear, whereas the transformation from the hidden-layer space to the output space is linear [4]. Fig. 2 shows an RBF network with *m* neurons in the input layer, *h* neurons in the hidden layer, and *k* neurons in the output layer. The parameters to be trained are the centers ($\mu_1, \mu_2, \ldots, \mu_h$) and widths ($\sigma_1, \sigma_2, \ldots, \sigma_h$) of the receptive fields of hidden layer neurons, and the output layer weights.

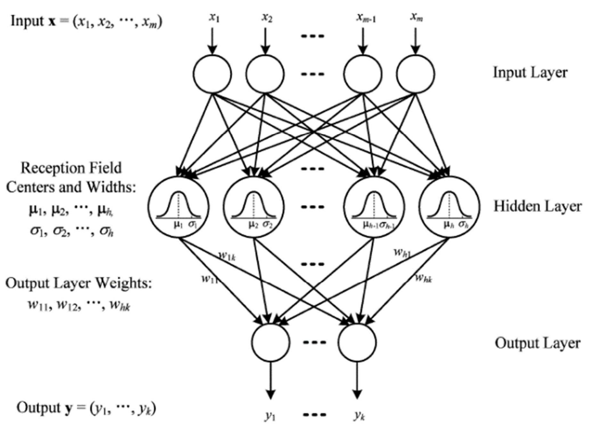

Input $\mathbf{x} = (x_1, x_2, \cdots, x_m)$
x_1
x_2
x_{m-1}
x_m
Input Layer
Reception Field Centers and Widths:
$\mu_1, \mu_2, \cdots, \mu_h$
$\sigma_1, \sigma_2, \cdots, \sigma_h$
Hidden Layer
$\mu_1\ \sigma_1$
$\mu_2\ \sigma_2$
$\mu_{h-1}\sigma_{h-1}$
$\mu_h\ \sigma_h$
Output Layer Weights:
$w_{11}, w_{12}, \cdots, w_{hk}$
w_{1k}
w_{11}
w_{h1}
w_{hk}
Output Layer
Output $\mathbf{y} = (y_1, \cdots, y_k)$
y_1
y_k

Figure 2 : A sample three-layer RBF neural network

2.3 Optimized RBF Algorithm Based on Genetic Algorithm

RBF networks can self-adaptively adjust the hidden layer in the training stage according to specific problems. Hidden layer allocation can be decided by the capacity, the category, and the distribution of the training samples. It is capable of dynamically identifying center points and width of the hidden layer's neurons and the hidden layer.

The main content of using genetic algorithm to optimize RBF network includes the chromosome coding, the definition of fitness function, and the construct of genetic operators. The use of GA-RBF optimization algorithm can be seen as an adaptive system; it is to automatically adjust its network structure and connection weights without human intervention and make it possible to combine genetic algorithm with the neural network organically. [5]

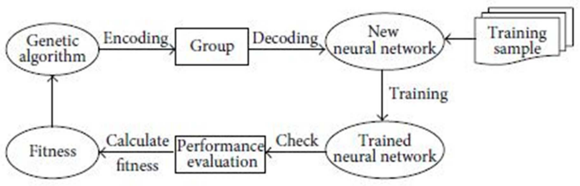


Figure 3 : The flow chart of GA-RBF algorithm

2.3.1. Chromosome Encoding. Suppose the number of RBF neural network's maximum hidden neurons is s and the number of output neurons is m.

Hidden layer's neurons with binary coding, and the coding scheme are as follows:

$$c_1c_2 \ldots c_s. \qquad 3)$$

Here, the number of hidden layer neurons is encoded by binary encoding method, represented by c_i, the value of which is 0 or 1. When c_i =1, it means that the neuron exists; while c_i=0 means that the neuron does not exist, and s represents the upper limit.

The weights with real encoding, coding scheme are as follows:

$$w_{11}w_{21}\ldots w_{s1}w_{12}w_{22}\ldots w_{s2}\ldots w_{1m}w_{2m}\ldots w_{sm} \qquad 4)$$

Here, the weights from hidden layer to output layer was encoded by real number encoding method, and w_{ij} represents the connection weight from the ith output neuron to the jth hidden neuron.

The threshold also with real encoding scheme is as follows:

$$\theta_1\ \theta_2\ldots\ \theta_m \qquad 5)$$

Here, the threshold of output layer neuron is also encoded by real number encoding method; θj represents the threshold of jth output neuron.

So, in conclusion, the complete coding strand of one chromosome is the combination of the structure, connection weight, and threshold, and it is as follows:

$$c_1c_2\ \ldots\ c_s w_{11}w_{21}\ldots w_{s1}w_{12}w_{22}\ldots w_{s2}\ldots w_{1m}w_{2m}\ldots w_{sm}\ \theta_1\ \theta_2\ldots\ \theta_m \quad 6)$$

2.3.2. Constructing Genetic Operator.

(1) Selection operator. We are choosing proportional selection operator and using roulette wheel selection, which is the most commonly used method in genetic algorithm. The individuals with higher fitness will more likely be selected, while the individuals with lower fitness also have the chance to be selected, so that it keeps the diversity of the population under the condition of "survival of the fittest".

(2) Crossover Operator. We use single-point crossover operator as the crossover operator; each time we choose two individuals of parent generation to crossover so as to generate two new individuals, which are added into the new generation. We will repeat this procedure until the new generation population reaches the maximum size. We use single-point crossover although the complete procedure uses hybrid encoding; however, the crossover operation for binary encoding and real encoding is the same.

The strategy of elitism selection is used here, that is, to retain several individuals with highest fitness to the next generation directly; this strategy prevents the loss of the optimal individual during the evolution.

(3) Mutation Operator. Mutation operator uses reversal operator, as it uses hybrid encoding; different operations are applied to different code system. Binary encoding uses bit flipping mutation; that is to say, some bit of the chromosome may turn from1 to 0 or 0 to 1. For real encoding, we use Gaussian mutation; that means some gene of the chromosome will add a random Gaussian number.

2.3.3. Calculate Fitness. Fitness function evaluation is the basis of genetic selection, so it will directly affect the performance of genetic algorithm. Therefore, the selection of fitness is very important as it directly affects the speed of genetic algorithm convergence and whether we can find the optimal solution.

The original data sets are divided into training data sets and testing data sets, using the network training error and the number of hidden neurons to determine the RBF neural networks' corresponding fitness of the chromosomes.

Suppose the training error is E, the number of hidden layer neurons is s, and upper limit of the number of hidden layer neurons is $s_{\max}$. So the fitness F is defined by

$$F = C - E \times \frac{s}{s_{max}} \qquad 7)$$

In the formula, C is a constant number; this formula ensures that the smaller the network size (fewer hidden layer neurons) and the smaller the training error, the higher the corresponding fitness of chromosome.

2.3.4. Parameters of RBF Neural Network.

Three parameters of the RBF Neural Network can be adjusted: centers and its width of the hidden layer's basis function and the connection weights between hidden layer and output layer.

1. **Basis Function Centers**. We can select the s centers according to the experience; the spacing is d; the width of the selected Gaussian function is

$$\sigma = \frac{d}{\sqrt{2s}} \qquad 8)$$

2. **Basis Function.** We use K-mean cluster method to select the basis function; the center of each cluster is regarded as the center of basis functions. As the output is linear unit, its weights can be calculated directly by LMS method. We use the iterative formula (9) to modify the training error, so we can get the following optimal neural network algorithm:

$$e = \sum_{k=1}^{n} (t_k - y_k)^2 \qquad 9)$$

Here, e is the error fraction, t_k is the actual value, and y_k is the output of the neural network.

2.3.5. The Basis Steps of GA-RBF Algorithm [5]

Step 1. Set the RBF neural network, according to the maximum number of neurons in the hidden layers; use K-clustering algorithm to obtain the center of basis function; use formula (8) to calculate the width of the center.

Step 2. Set the parameters of the GA, the population size, the crossover rate, mutation rate, selection mechanism, crossover operator and mutation operator, the objective function error, and the maximum number of iterations.

Step 3. Initialize populations P randomly; its size is N (the number of RBF neural network is N); the corresponding network to each individual is encoded by formula (6).

Step 4. Use the training sample to train the initial constructed RBF neural network, whose amount is N; use formula (7) to calculate the network's output error E.

Step 5. According to the training error E and the number of hidden layer neurons s, use formula (7) to calculate the corresponding chromosome fitness to each network.

Step 6. According the fitness value, sort the chromosome; select the best fitness of the population, denoted by Fb; verify $E < E\text{min}$ or $G \geq G\text{max}$; if yes, turn to Step 9; otherwise turn to Step 7.

Step 7. Select several best individuals to be reserved to the next generation New$P$ directly.

Step 8. Select a pair of chromosomes for single-point crossover, to generate two new individuals as members of next generation; repeat this procedure, until the new generation reaches the maximum size of population Ps; at this time, the coding will be done separately.

Step 9. Mutate the population of new generation; binary coding part and real number coding part should use different mutation strategies. Then the new population is generated; set P = New$P$, $G = G + 1$; return to Step 4.

Step 10. Get the optimal neural network structure, and the iteration of genetic algorithm is terminated, which means the optimizing stopped.

Step 11. The new neural network's weight learning is not sufficient, so use LMS method to further learn the weights.

End of the algorithm.

3. FAULT LOCALIZATION USING GA-RBF NEURAL NETWORK

Suppose we have a program P with *m* statements, executed on *n* test cases. Let s_j be the *j*th statement of P. The vector c_{t_i} provides us with information on how the program P is covered by test t_i. In this paper, such coverage is reported in terms of which statements in P are executed by t_i. We have

Journal of Theoretical and Applied Information Technology
15th August 2016. Vol.90. No.1
© 2005 - 2016 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

$$c_{t_i} = [(c_{t_i})_1, (c_{t_i})_2, \ldots, (c_{t_i})_m]$$

where

$$(c_{t_i})_j = \begin{cases} 0, if\ statement\ s_j\ is\ not\ covered\ by\ test\ t_i \\ 1, if\ statement\ s_j\ is\ covered\ by\ test\ t_i \end{cases}$$

for $1 \le j \le m$

The value of result r_{t_i} depends on whether the program execution of t$_i$ succeeds or fails. It has a value of 1 if the execution fails, and a value of 0 if the execution succeeds. We construct an RBF neural network with m input layer neurons, each of which corresponds to one element in a given c_{t_i} and one output layer neuron, corresponding to r_{t_i}, the execution result of test t$_i$. In addition, there is a hidden layer between the input and output layers. The overall network structure and the number of hidden neurons can be determined by using GA-RBF algorithm (III-A).

Once the neural network is trained, the mapping between the input and the output (test case and the corresponding execution result) can be predicted by it. Thus the trained neural network can now be used to identify suspicious code of a given program in terms of likelihood of containing bugs. In order to determine the degree of suspiciousness, a set of virtual test cases $v_1, v_2 \ldots v_m$ whose coverage vectors are $c_{v_1}, c_{v_2}, \ldots, c_{v_m}$,(Fig.4, (a)) where each test v$_j$ covers only one statement s$_j$.

$$\begin{bmatrix} \mathbf{c}_{v_1} \\ \mathbf{c}_{v_2} \\ \mathbf{c}_{v_3} \\ \mathbf{c}_{v_4} \\ \mathbf{c}_{v_5} \\ \mathbf{c}_{v_6} \\ \mathbf{c}_{v_7} \\ \mathbf{c}_{v_8} \\ \mathbf{c}_{v_9} \\ \mathbf{c}_{v_{10}} \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

(a) Input coverage Vector for virtual test case

$\hat{r}_{v_1}$	0.0384	$\hat{r}_{v_6}$	0.0179
$\hat{r}_{v_2}$	0.0481	$\hat{r}_{v_7}$	0.0157
$\hat{r}_{v_3}$	0.1246	$\hat{r}_{v_8}$	0.2900
$\hat{r}_{v_4}$	0.0768	$\hat{r}_{v_9}$	0.0066
$\hat{r}_{v_5}$	0.0173	$\hat{r}_{v_{10}}$	0.0782

(b) Output of the trained network are the suspiciousness with respect to corresponding statement

Figure 4 : Input and output based on the example.

[6],[7],[8] suggest that , if the execution of a test case fails, program bugs responsible for this failure are most likely to be contained in the corresponding execution slice, the statements executed by failed test case. Therefore, If the execution of v$_j$ fails, the probability that bugs are contained in s$_j$ is high. Therefore, we should examine the statements whose virtual test case fail. Finally the output corresponding to the virtual test case indicates that the larger value of result v$_j$ ($\hat{r}_{v_j}$) implies higher degree of suspiciousness.

Thus, the overall procedure for using the GA-RBF algorithm in context of fault localization can be described as follows.

- Build a modified RBF neural network with *m* input neurons and one output neuron. Each neuron is the hidden layer uses the Gaussian basis function as its activation function.
- Use GA to optimize the network structure and weights of the RBF algorithm simultaneously.
- Use LMS method for weights further learning.
- Use the virtual coverage vectors c_{v_j}, $1 \le j \le m$ as the inputs to the trained network to produce the outputs $\hat{r}_{v_j}$, $1 \le j \le m$.
- Assign $\hat{r}_{v_j}$ as the suspiciousness of the jth statement.

The statement can now be examined one by one in descending order of suspiciousness, until a fault is located. The proposed technique can be represented in a diagram (Fig 5).

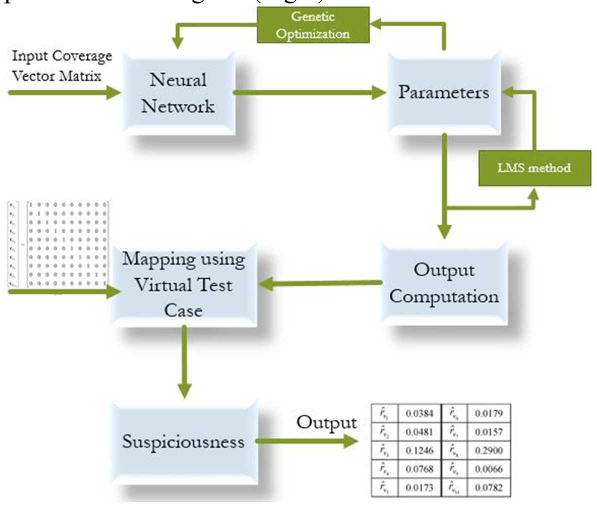


Figure 5: Overall Diagram of proposed Fault Localization Model

4. EMPERICAL EVALUATION AND COMPARISION

A comprehensive study and comparison is provided by [3] and [9] in context of software fault localization. Studies suggest that RBF neural network can be used effectively for fault localization. Comparisons in terms of effectiveness on the basis of less statements examined suggests that RBFBest is empirically better than Crosstab. In terms of number of faults versions, it is observed that RBF performs better than both Ochiai[11] and Jaccard[11]. Combining the results presented in [3] and those from previous study[12] which shows that RBF is also more effective than techniques such as Tarantula[13].Liblit05[14],and SOBER[15]. [3] suggests that RBF is more effective than 7 different competing fault localization techniques: Tarantula, SOBER,Liblit05, Ochiai, Jaccard, Crosstab, and H3C.

However, the reduction in sampling rate adversely affects the ability of fault localization to distinguish one statement from another. The author[3] suggested to measure the effectiveness as the function of sampling rate. Thus indicating that higher number of the training sample would provide better mapping and effective classification by RBF neural Network.

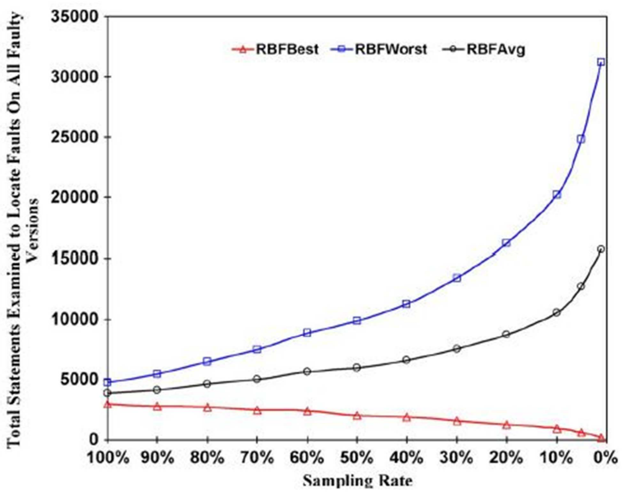
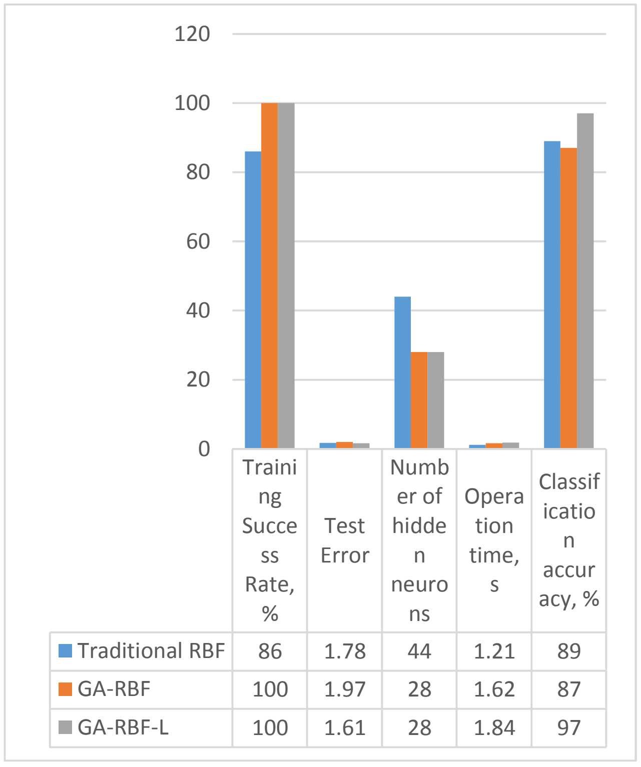


Figure 6: Total number of statements that must be examined as a function of sampling rate.

The RBF approach is similar to a support vector machine (SVM) methodology in that an input vector is mapped into a feature space (in our case the features are the neurons at the hidden layer of the modified RBF), and then a linear model is used to compute a weighted sum of features [10].

[5] Suggests that the training success rate aspect of GA optimized RBF algorithm is superior to the traditional RBF algorithm. Operationally, GA-RBF algorithm takes more time and training error is equivalent to traditional RBF algorithm. From the recognition precision aspect, the GA-RBF-L algorithm's classification precision is the best. (Fig7)



	Training Success Rate, %	Test Error	Number of hidden neurons	Operation time, s	Classification accuracy, %
🟦 Traditional RBF	86	1.78	44	1.21	89
🟧 GA-RBF	100	1.97	28	1.62	87
▨ GA-RBF-L	100	1.61	28	1.84	97

Figure : 7 Comparision of the performance of RBF-variations for waveform database

5. CONCLUSION AND DISCUSSION

In this paper, we propose a model and usage of a Genetically Optimized RBF neural network in context of software fault localization. The GA optimizes neural network structure and connect weight simultaneously and then uses LMS method to adjust the network further. GA-RBF algorithm takes more time in optimization, but it can reduce the time which is spent in constructing the network [5]. Thus, implementation in various fields [16] and competitive study suggests that GA-RBF algorithm is a self-adapted and intelligent algorithm [5] and provides better performance than traditional RBF techniques. Thus, the usage of GA-RBF network in context of fault localization is recommended by us and we intend to further optimize and improve the technique by inspecting on other aspects of same technique.

The major challenge to implement the study is, first, the coverage data collection and second the neural network implementation. The code coverage analysis of a code plays a vital role in order to train the neural network and the neural network itself may have various implementations. The effectiveness of the fault localization techniques based upon machine learning approaches is sensitive to Data Set size and also the neural network design and implementation.

REFERENCES :

[1] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*. Cambridge, MA: MIT Press, 1995.

[2] EP. Singla, K. Subbarao, and J. L. Junkins, “Direction-dependent learning approach for radial basis function networks,” *IEEE Trans. Neural Networks*, vol. 18, no. 1, pp. 203–222, January 2007.

[3] W.Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham, “*Effective Software Fault Localization Using an RBF Neural NEtwork*”, IEEE TRANSACTIONS ON RELIABILITY, VOL.61, NO.1, MARCH 2012

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. New York: Prentice Hall, 1999.

[5] “*A New Optimized GA-RBF Neural Network Algorithm*” Weikuan Jia, Dean Zhao, Tian Shen, Chunyang Su, Chanli Hu, and Yuyan Zhao, Computational Intelligence and Neuroscience
Volume 2014 (2014), Article ID 982045, Hindawi 13 October 2014.

[6] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, “Fault localization using execution slices and dataflow tests,” in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, p. 143-15.

[7] W. E. Wong and Y. Qi, “*Effective program debugging based on execution slices and inter-block data dependency*,” *Journal of Systems and Software*, vol. 79, no. 7, pp. 891–903, Jul. 2006.

[8] W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, “*Smart debugging software architectural design in SDL*,” *Journal of Systems and Software*, vol. 76, no. 1, pp. 15–28, April 2005.

[9] *Survey of Software Fault Localization for Web Application* [Vol.5, No.3, June 2015], *International Journal of Current Engineering and Technology*, Swati B. Ghaawate and Sharmila Shinde

[10] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Berlin, Germany: Springer, 2001.

[11] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, “*A Practical Evaluation of Spectrum-based Fault Localization*,” Journal of Systems and Software, vol. 82, no. 11, pp. 1780–1792, 2009.

[12] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, “Using an RBF neural network to locate program bugs,” in *Proceedings of the 19th IEEE Intl. Symposium on Software Reliability Engineering*, Seattle, USA, November 2008, pp. 27–38.

[13] J. A. Jones and M. J. Harrold, “Empirical evaluation of the Tarantula automatic fault-localization technique,” in *Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering*, Long Beach, California, USA, Dec. 2005, pp. 273–282.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “*Scalable statistical bug isolation*,” in Proceedings of the 2005 ACM SIGPLAN *Conference on Programming Language Design and Implementation*,Chicago, Illinois, USA, Jun. 2005, pp. 15–26.

[15] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, “*Statistical debugging: a hypothesis testing-based approach,*” IEEE Trans. Software Engineering, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[16] C.Harpham, C.W.Dawson, M.R. Brown, *A review of genetic algorithms applied to training radial basis function networks*, *Neural computing and Applications*, September 2004, Volume 13, Issue 3, pp 193-201