An MLNN Based Intelligent Model for Software Quality Prediction

[1]Ritu, [2]O. P. Sangwan

[1]PhD Scholar, [2]Professor

[1, 2] Department of Computer Science & Engineering

[1, 2] Guru Jambheshwar University of Science & Technology, Hisar, India

Abstract: With the advent of soft-computing technologies, ecommerce and OTT platforms have gained immense market capturing capabilities. Earlier, the ability to predict the effectiveness of a campaign had to be done using expert supervision which cost a lot of money. Neural network systems have almost been successful in eliminating the need for such a specialist. This paper deals with the development of a neural network scheme for software quality prediction where a feed-forward neural network has been employed for prediction purposes. Two feed-forward neural network algorithms such as Radial Basis Function Network (RBFN) and Multilayered Perception Network (MLNN) are presented and the effectiveness of both the approaches have been analyzed. To enhance the training accuracy, a momentum term has been added in MLNN. Both the approaches are hard-coded and implemented in MATLAB where the performance of the available NN toolbox in MATLAB has also been compared with both the approaches. The comparison has been performed on 128 software dataset for which quality has to be predicted. Out of 128 dataset, 100 data pairs are used for training, and 28 data input is used for testing purposes. Furthermore, it is shown that the results obtained by the MLNN are closely matched with the actual software quality and deliver better results among all three approaches.

***IndexTerms* - MATLAB, MLNN, Neural Networks, RBFN, Software Quality Prediction.**

I. INTRODUCTION

In the present time, computer software is a key asset that has a huge impact on our daily life. With this continuously growing digital world, usage of the software has been greatly enlarged and thus the quality of software becomes more important whether in terms of usability or resilience against any malicious attack. The quality of the software [1] can be accessed with many parameters and it may be userdependent as well e.g. some users want lower maintenance cost and some wants it for confidential data, so, the cost may be increased in that case. It also becomes difficult to select particular software as various factors are included. So, it will be very useful if there is an automated quality prediction scheme that can save time and money for a firm or individual who otherwise asks for an expert. Various approaches have been implemented in the literature for automatic software quality prediction like fuzzy logic [2], neural networks, etc. [3]. The main disadvantage of fuzzy scheme is its nature of rule-based implementation and the performance is highly dependent on fuzzy rules. Nevertheless, a robust and effective approach for software quality prediction is still an interesting area. The main aim of this paper is to predict the software quality using a neural network where feed-forward neural network approaches have been employed for software prediction. Two feed-forward neural network algorithms namely RBFN and MLNN have been presented and the performance of both approaches has been compared. The update laws for weight parameters and RBFN centers have been derived using a back-propagation algorithm where a momentum term has also been used to enhance the tracking accuracy. Both of the approaches are hard-coded and simulation has been performed in the programming language MATLAB. Both of these approaches have also been compared with the results obtained with the MATLAB NN Toolbox. Simulation results have been performed on a dataset of 128 various software, the results show that the proposed MLNN gives the lowest Mean Square Error (MSE) among these three approaches. The manuscript is organized as follows: Related work with the presented approach has been described in section II. Section III describes various factors affecting the quality of software which is showing that five parameters namely Reliability, Usability, Efficiency, Maintainability, Portability are sufficient to access the software quality. A detailed description of the feed-forward neural network architecture of RBFN and MLNN and their training algorithm can be found in section IV. This is followed by extensive simulation results along with the comparative analysis of RBFN, MLNN, and MATLAB NN Toolbox in section V, Finally, concluding remarks are given in section VI.

II. RELATED WORKS

Over the last decade, the concept of neural networks and learning systems has led immense growth. Ranging from academia to industries, the study of neural networks is quite essential in cases that involve systems with unknown dynamics, systems with several constraint parameters or both. With the ability to increase the number of measurable attributes for prediction, neural network systems have been successfully implemented by popular websites such as Flipkart, Snapdeal, Netflix and many more. Several neural network architectures namely feed-forward neural network, feedback neural network, self-organizing feature map have been developed to suit the needs of the application based on the available computational resources [4]. Feed-forward neural networks are widely used in the literature due to their simplicity and capability of approximation. A common architecture in this category is the multi-layered neural network (MLNN) trained with a back-propagation algorithm. The concept relies upon the theory of gradient descent algorithm and tunes its weights till the energy function is minimized [5]. It must be noted that the theory of back-propagation can be applied to other network architectures as well. MLNNs trained using backpropagation have the issue of the vanishing gradient and hence cannot be appropriately implemented to networks with more than one hidden layer. However, several applications have been found in the academic as well as industries. Alternative network architecture to MLNNs is the radial-basis function network (RBFN) which relies on a single hidden layer structure and a limited number of weights [6]. RBFN has been used in several applications in the past. Some Authors in [7] use Recurrent RBFN to perform time series prediction. It is designed such that it can predict two temporal series that help in nonlinear system identification. The application of RBFNs in recognizing human emotions using motion is investigated in [8]. The RBFN is designed such that it learns the correlation of human expressions with facial feature motion patterns. The developed network provided an accuracy of 88% in

identifying expressions. Artificial intelligence has also been extensively used in gauging the efficacy of software passed to the public domain. Authors in [9] use neural networks to determine fault modules in software that require special attention. This is done at an earlier stage of the development in order to improve the final product. Authors [10] proposed the use of statistical methods to assess the software quality. These measures are made based on software attributes, also named software metrics. In [11], authors proposed suitable measurement metrics to capture the quality of object-oriented code and design for detecting fault-proneness of classes using neural networks. In [12], authors have constructed a model to predict the class that will turn out faulty contents, in future releases of a java application. The model was then validated on a subsequent release of the same application with high accuracy. In paper [13], authors developed a fully automated software fault prediction model. Here, there is no need of an expert for prediction. The authors used X-Mean clustering algorithm. In [14], authors presented a mathematical model for back-propagation study on Lagrangian formalism. Here, it is suggested various extensions of the basic algorithm. Authors in [15] used a solution for estimating software defect fixing effort using a Self-organizing Neural Network. Authors in [16] suggested the use of principal component analysis for enhancing the performance of neural networks. The authors obtained raw data from a large commercial software using software metrics. Two neural nets were trained, one with raw data and one with PCA data. These two nets were compared and it was concluded that the data with PCA produced more insightful results. A similar study is presented by the authors in [17]. The authors conclude that neural network models are better at predicting software quality as compared to statistical methods. In [18], neural networks are used for the effective measurement of service quality. The authors conclude that the perception-only model is more accurate in predicting service quality as compared to the perception-minus-expectation model. Various studies in the area of neural networks for various factors related to the software are listed in Table 1.

Table 1. NN Based Various Studies in Software Assessment

Sr. No.	Objective	Method	Comment
1	Prediction of Software Reliability [19]	Back-Propagation	Uses 4 performance criteria
2	Prediction of effort in developing a software [20]	Back-propagation	MATLAB NN Toolbox
3	Assessment of software reusability [21]	Back-propagation	MATLAB NN Toolbox
4	fault prediction in software [22]	Multilayer Perceptron Neural Network	Fuzzy bell-shaped activation function
5	Prediction of quality of software [23]	NN with PCA	Uses Software fault analysis

With the above discussion and Table I, we see that Neural Network-based approaches have been widely used in the software assessment however most of the abovediscussed methods use the ANN toolbox, and hence still a comprehensive analysis of software prediction is lacking. By hard-coding the ANN architecture, we have more flexibility depending on the application and thus better results can be obtained. Therefore first, we analyze two feed-forward algorithms i.e. RBFN, MLNN and comparisons are performed with MATLAB NN toolbox thereafter.

III. DATA PREPARATION

There has been a lot of research on the parameters which can influence the quality of software. It is claimed that the following five inputs are sufficient enough to estimate the software quality, these parameters are as follows:

1) Reliability: The ability of the software product can sustain its level of performance under stated conditions for a stated period of time. For better quality reliability should be highly released.

2) Usability: The ability to which the software product makes it easy for users to operate and control it. For better quality usability should be highly released.

3) Efficiency: The ability to which the software product provides appropriate performance, relative to the number of resources used, under stated conditions.

4) Maintainability: The ability to which the software product can be modified. In modifications included corrections, improvements, or adaptations of the software to changes in the environment and in the requirements and functional specifications (the effort needed to be modified). For better quality maintainability should be less released.

5) Portability: The ability of the software product to be transferred from one environment to another. The environment may include extended hardware or software environment. The component should have high portability for better quality. Thus to predict the software quality, the following five inputs namely Reliability, Efficiency, Usability, Maintainability, and Portability are used in this research. Software quality is the output of ANN which is calculated based on these five inputs. A total of 128 data-set of input-output pair has been collected where 100 data-set are used for ANN training and 28 data-set is used for testing purpose. The dataset has been generated by various studies and verified via expert comments. Due to space constraint, only 20 data-set used in this study are shown in Appendix A.

IV. PROPOSED METHODOLOGY

A. Radial Basis Function Network

Figure 1 depicts a generic RBFN architecture. As mentioned earlier, it consists of a single hidden layer with neurons that use radial-basis functions as activation functions. Radial basis functions are functions whose value depends upon the distance between the input and the fixed point. Usually, the computed distance is known as the Euclidean distance. Gaussian RBF is a commonly used activation function [6], is employed here as well: $\boldsymbol{\Phi(x) = e^{(-\epsilon x^2)}}$.

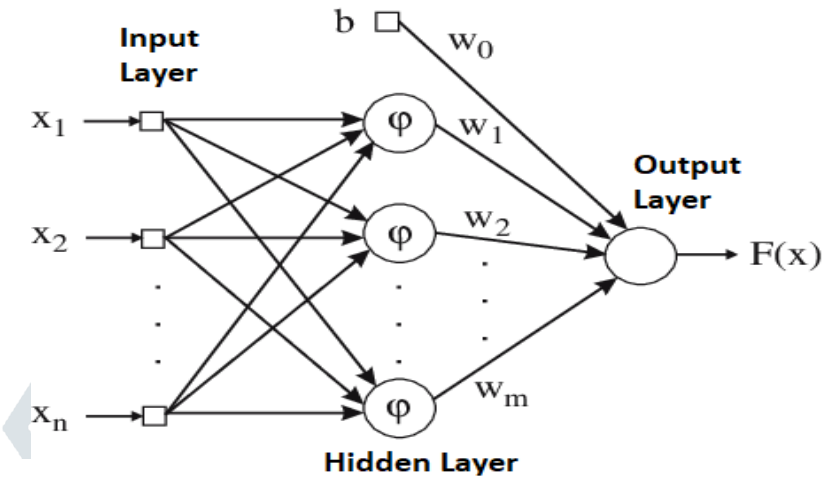

Fig. 1 RBFN Architecture

where, x represents the Euclidean distance between the input and the fixed pointed center. In the succeeding subsection, we present the weight update formula using the gradient descent algorithm [4].

As mentioned earlier, the back-propagation algorithm will be used to update the weights and the centers for the RBFN. If

- $x_1, x_2, ..., x_n$: Represent the inputs such that $x_i \in R$.
- $c_1, c_2, ..., c_h$: Represent the centers of the RBFN. Each center is such that $ci \in R_n$.
- $w_1, w_2, ..., w_h$: Represent the weights associated with each center such that $w_i \in R$.
- y: Represents the output of RBFN.

and the term $x = [x_1, x_2, x_3, ..., x_n]$ then the output of RBFN can be calculated as follows:

$$y = w_1 e^{\frac{-(||c_1 - x||)^2}{2\sigma^2}} + w_2 e^{\frac{-(||c_2 - x||)^2}{2\sigma^2}} + \ldots + w_h e^{\frac{-(||c_h - x||)^2}{2\sigma^2}}$$

If σ is the width of the Gaussian function then above equation can be rewritten as:

$$y = \sum_{i=1}^{h} w_i \, e^{\frac{-(||c_i - x||)^2}{2\sigma^2}} = \sum_{i=1}^{h} w_i \, f(c_i, x) \qquad (1)$$

Remark 1: RBFN constitutes linear in weight parameters as shown in Equation 1 which is sometimes a very good property, makes it easier to find the weight update rules. Due to this nature, the RBFN is widely used in the system and control. To perform weight updates using gradient-descent, an energy function has to be considered. In the case of a single output RBFN, the energy function is as shown below:

$$J = \frac{1}{2}(y^d - y)^2 \qquad (2)$$

The idea is to find new weights such that this energy function should be minimized. Hence, from the gradient descent-based back-propagation algorithm, the weight update equation is considered as given below. Figure 2 represents the gradient descent algorithm that is used for weight and center updates.

$$w_{i,\text{new}} = w_{i,\text{old}} - \zeta \frac{\partial J}{\partial w_i} = w_{i,\text{old}} - \zeta \frac{\partial J}{\partial y} \frac{\partial y}{\partial w_i} \qquad (3)$$

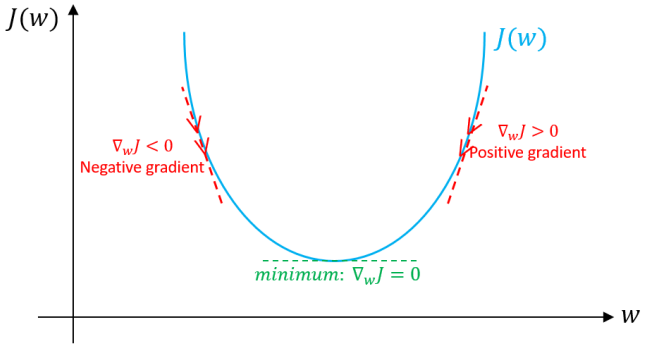

Fig. 2 Gradient Descent Algorithm

The above expression can be simplified as:

$$\begin{aligned} w_{i,new} &= w_{i,old} + \zeta(y^d - y)\left(||c_i - x||\right)^2 \log\left(||c_1 - x||\right) \\ &= w_{i,old} + \zeta(y^d - y) f(c_i, x) \end{aligned} \quad (4)$$

The next stage is to update the centers which can be performed as follows. On similar lines for w_i, the center c_i can be updated as:

$$c_{i,new} = c_{i,old} - \zeta \frac{\partial J}{\partial c_i} \quad (5)$$

As done previously, it is being used the chain rule to expand the partial derivative. Hence, it is got the updated law for $\mathbf{c_i}$ as:

$$c_{i,new} = c_{i,old} + \zeta(y^d - y)(||c_i - x||)(\log(||c_i - x||^2) + 1) w_i \quad (6)$$

In certain cases, as in Figure 1, a bias term would be added at the output layer. In such cases, Equation 4 gets modified with an extra term. This can be written as:

$$y = \sum_{i=1}^{h} w_i \, e^{\frac{-(||c_i - x||)^2}{2\sigma^2}} + w_b b = w_{b,old} + \zeta(y^d - y) b \quad (7)$$

To update the weight of the bias, Equation 3 is modified,

$$w_{b,new} = w_{b,old} - \zeta \frac{\partial J}{\partial w_b} \quad (8)$$

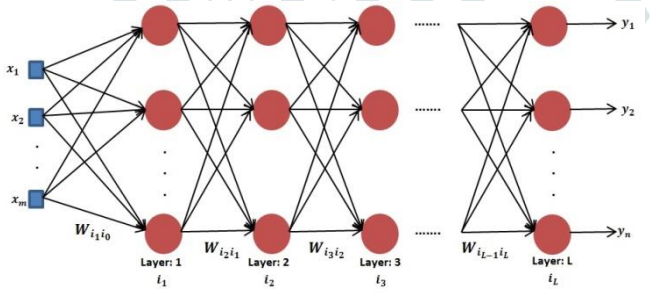

Fig. 3 L-Layered Feed-Forward Neural Network

Using Equations 4, 6 and 7, any RBFN can be trained to fit any dataset. MATLAB simulations shown in the succeeding sections prove the working of the same.

B. Multi-Layered Neural Network

A multilayered Neural network (MLNN) can have more than one hidden layers as opposed to the RBFN. The architecture of MLNN is shown in Figure 3 where:

x : $m \times 1$ input vector.
y : $n \times 1$ output vector.
i_n : $n = 1,2,3,\ldots,L$; index for representing a neuron in the n^{th} layer.
i_0 : Index for representing a neuron in the input layer.
h_{i_n} : Weighted sum of the input to the i_n^{th} neuron in the n^{th} layer.
v_{i_n} : Response of the i_n^{th} neuron in the n^{th} layer.
$W_{i_n i_{n-1}}$: Weight connecting the i_n^{th} neuron in the n^{th} layer and the i_{n-1}^{th} neuron in the $(n-1)^{th}$ layer.

All the weight parameters need to be updated to meet the performance index which is given same as that of Equation 2.

$$J = \frac{1}{2}(y^d - y)^2 \quad (9)$$

These weight parameters are updated using gradient descent update law as shown in previous section 4.1. Generalized weight update law can be derived as:

$$W_{i_n i_{n-1}}(k+1) = W_{i_n i_{n-1}}(k) + \eta \delta_{i_n} v_{i_{N-1}}(k) \quad (10)$$

Where, $v_{i_{n-1}}$ is the output of the i^{th} neuron of the Layer n-1 and δ_{i_n} for each layer is expressed as:

$$\delta_{i_n} = \begin{cases} \left[y_{i_L}^d(k) - y_{i_L}(k)\right]\left[y_{i_L}(k)\left(1 - y_{i_L}(k)\right)\right], \text{I} \\ v_{i_n}(k)\left[1 - v_{i_n}(k)\right]\sum_{i_{n+1}=1}^{n_{n+1}} \delta_{i_{n+1}} W_{i_{n+1} i_n}(k), \text{II} \end{cases} \quad (11)$$

Where, I is for the output Layer L and II is for the other hidden layers. It is shown in literature that one hidden layer is sufficient enough to approximate a nonlinear function and thus it has been used one hidden layer in MLNN architecture which is called as a 3-layered MLNN with one Input layer, one hidden layer and one output layer. For a 3-layered MLNN, output of MLNN can be calculated as follows: As shown in Figure 3, the input to the $\mathbf{i_1^{th}}$ neuron of hidden Layer is given as:

$$h_{i_1} = \sum_{i_0=1}^{m} W_{i_1 i_0} x_{i_0} \quad (12)$$

The output of the i_1^{th} neuron of hidden Layer, to the input, is given below. Note that, Log-sigmoidal activation function ($\Phi(z) = \frac{1}{1+e^{-z}}$) is used in this process.

$$v_{i_1} = \Phi(h_{i_1}) = \frac{1}{1+e^{-h_{i_1}}} \quad (13)$$

On similar patterns, computing the net output, it's got:

$$h_{i_2} = \sum_{i_1=1}^{n_1} W_{i_2 i_1} x_{i_1} \quad (14)$$

And the output of MLNN is obtained as:

$$y_{i_2} = v_{i_2} = \Phi(h_{i_2}) = \frac{1}{1+e^{-h_{i_2}}} \quad (15)$$

Where, n_1 is the number of neurons in the hidden layer.

Thus, the weight update rule is given as:

$$\left.\begin{aligned} W_{i_2 i_1}(k+1) &= W_{i_2 i_1}(k) + \eta \delta_{i_2} v_{i_1}(k) \\ W_{i_1 i_0}(k+1) &= W_{i_1 i_0}(k) + \eta \delta_{i_1} x_{i_0}(k) \end{aligned}\right\} \quad (16)$$

Where, η is the learning rate, $\delta_{i_2} = [y_{i_2}^d(k) - y_{i_2}(k)]\left[y_{i_2}(k)\left(1 - y_{i_2}(k)\right)\right]$ is the error back-propagated from the output layer whereas $\delta_{i_1} = v_{i_1}(k)[1 - v_{i_1}(k)]\sum_{i_2=1}^{n_2} \delta_{i_2} W_{i_2 i_1}(k)$ is the error back-propagated from the hidden hidden Layer.

A momentum term has been added to enhance the training accuracy and improve the learning. By using the momentum, weight update laws are expressed as:

$$\begin{aligned} W_{i_2 i_1}(k+1) &= W_{i_2 i_1}(k) + \eta \delta_{i_2} v_{i_1}(k) + \mu(W_{i_2 i_1}(k) \\ &\quad - W_{i_2 i_1}(k-1)) \\ W_{i_1 i_0}(k+1) &= W_{i_1 i_0}(k) + \eta \delta_{i_1} x_{i_0}(k) + \mu(W_{i_1 i_0}(k) \\ &\quad - W_{i_1 i_0}(k-1)) \end{aligned} \quad (17)$$

Where, μ is the learning rate for momentum term. The above update laws will be used to train the proposed MLNN architecture.

V. Simulation Study

A. Training of ANN

There are various parameters for training the ANN which can have a great impact on the accuracy of the trained ANN, for example, a very small value of learning rate may give slower convergence of parameters i.e. slow learning whereas a larger value may result in instability of the ANN parameters i.e. ANN instability occurs in this case hence these should be carefully selected. We have chosen the ANN parameters by trial and error starting from very small values and then gradually increasing until we get no further improvement in the performance.

1) RBFN training: Various parameters of RBFN are selected as follows:

Learning Rate for weight parameters update: 0.3
Learning Rate for centers update: 0.2
No. of Centers in hidden layer: 50
No. of weight parameters: 50

All of the 50 weight parameters are initialized with small values randomly selected between -0.1 to 0.1. We have written the whole training algorithm in MATLAB from scratch. A total of 50000 iterations are used to train the ANN where one iteration corresponds to the one passage of the whole data-set to ANN. All the weights are updated at each instant of the data set. Stopping criteria for ANN training is either root mean square error (RMSE) < 0.001 or 50000 iterations i.e. the training algorithm is stopped when RMSE reaches under 0.001 or 50000 iterations are reached. As all the weights, as well as centers, are also updated instantaneously, all the weights and center values are converged to their respective optimal values. The training results for all 100 samples are shown in Figure 4 as follows where the 'blue' line shows the actual quality of the software i.e. desired quality to be predicted and the 'red' line shows the software quality predicted by the ANN. From the Figure 4, we see that the desired quality has been correctly estimated by the ANN which constitutes the successful training of the ANN.

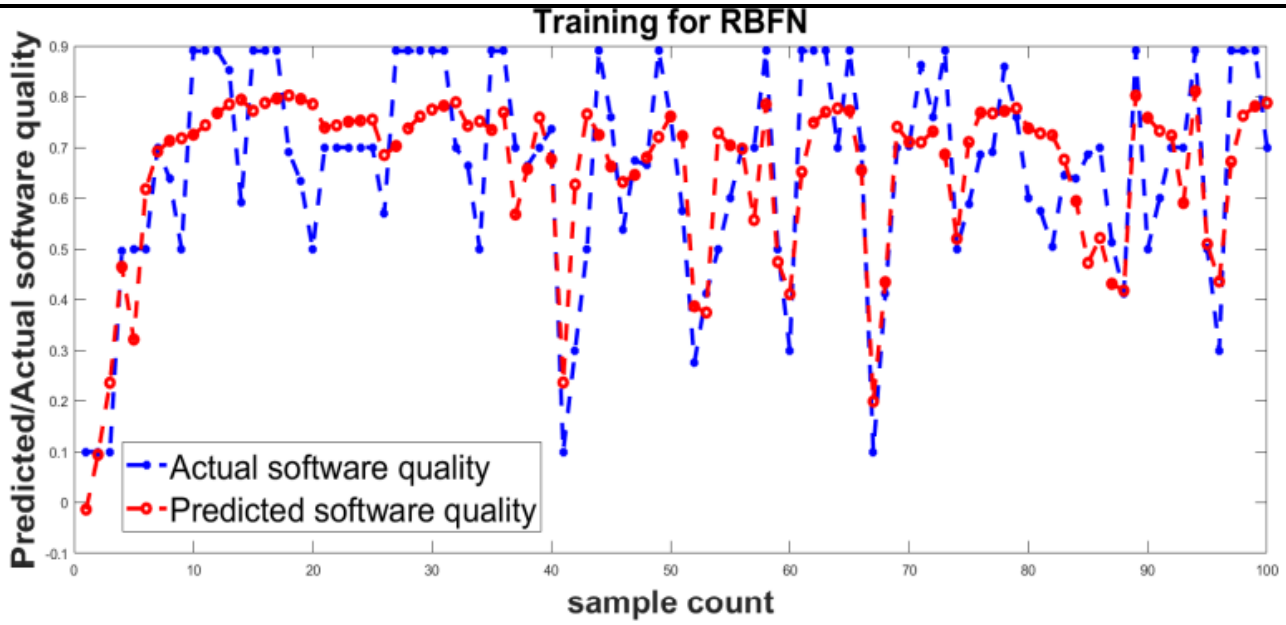

Fig. 4. Training results of RBFN

2) MLNN training:

The parameters of MLNN are selected as follows:

Learning Rate for weight parameters update: 0.3
Momentum: 0.2 No. of hidden Layer neurons: 8
No. of weight parameters: 48 Activation function: Sigmoid
Stopping criteria: RMSE < 0.001 or 50000 iterations
No. of Layers: 3 i.e. Input Layer, Hidden Layer and Output Layer

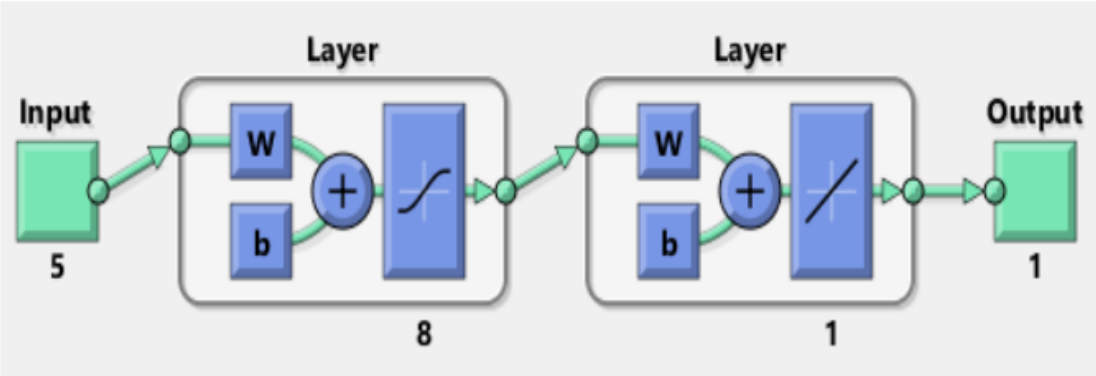

Fig. 5. ANN architecture using MATLAB NN Toolbox

No. of weight parameters in MLNN can be calculated as No. of weights = No. of neurons in Hidden Layer (No. of inputs + No. of outputs) 8 × 5 + 8 × 1 = 48. The presented 3-layered MPLN has been implemented in MATLAB R2018a on a core-i5 processor. The MATLAB code is written from the scratch to implement the MLNN algorithm to check the efficacy of the presented architecture instead of using the MATLAB NN toolbox directly.

Training results of both the NN toolbox and our approach have been presented here. At first, MATLAB NN toolbox training results are given here. The NN architecture is shown in Figure 5 which consists of 5 neurons in the input layer, 50 neurons in the hidden layer, and one output layer neuron. For a fair comparison, the NN parameters are kept the same as those of the presented approach which are described at the start of this section.

The training results obtained using NN toolbox is shown in Figure 6.

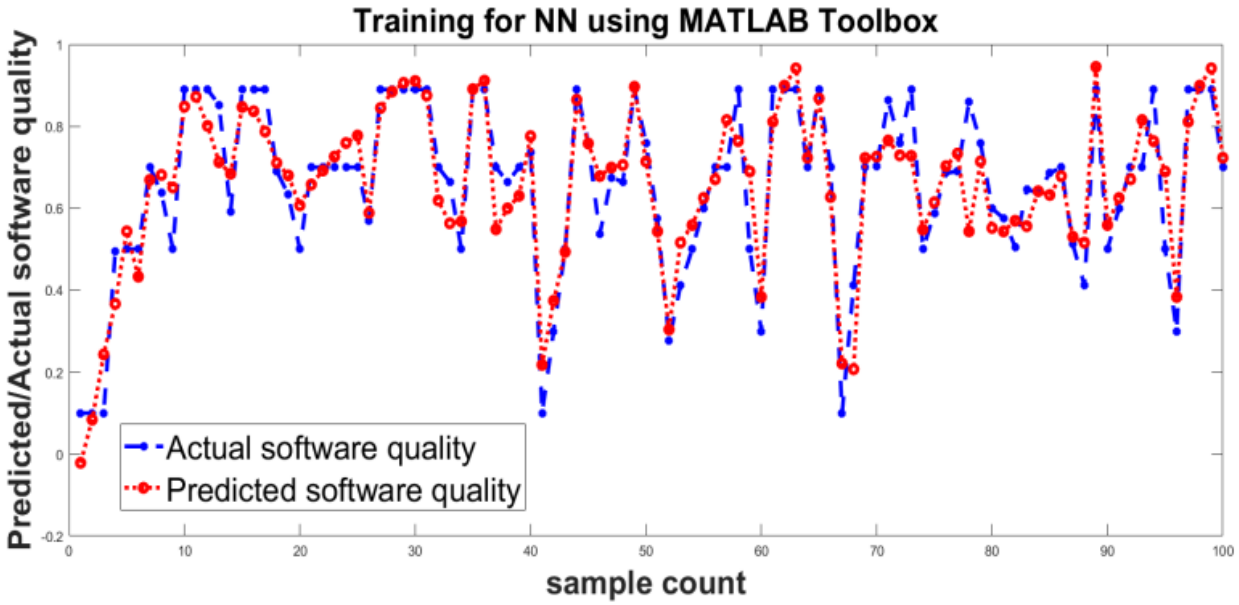

Fig. 6. MATLAB NN Tool training results

The training accomplished by proposed MLNN shows the training results in Figure 7.

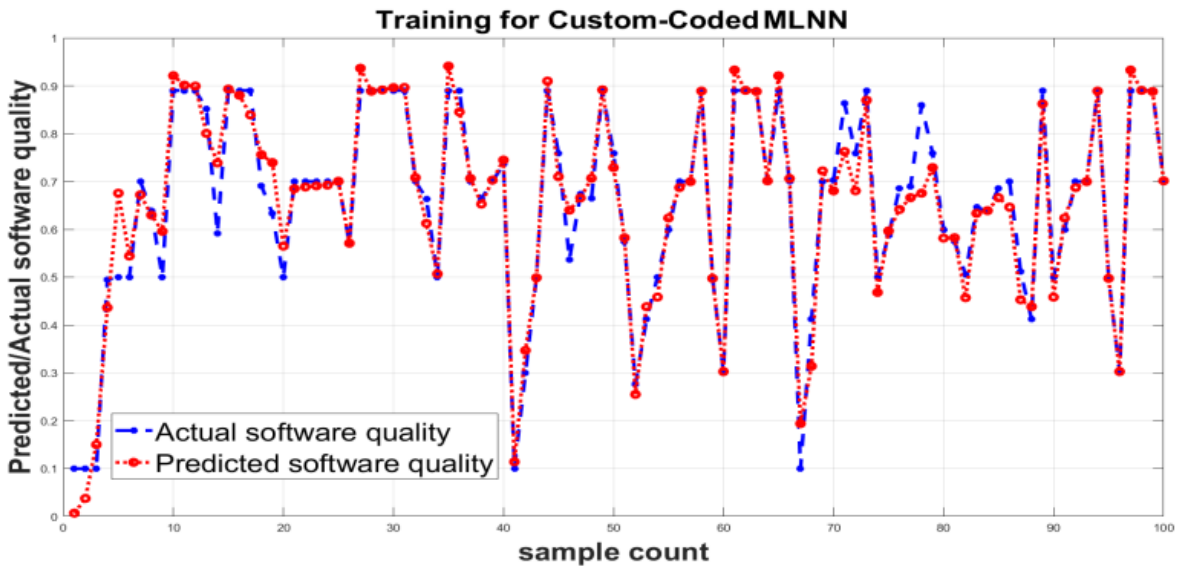

Fig. 7. Proposed MLNN training results

We have also plotted the RMSE per iteration during the training of the MLNN. The RMSE per iteration has been shown in Figure8 where, we see that ANN has been trained successfully as the RMSE is reached to 0.0236892 in 50000 iterations.

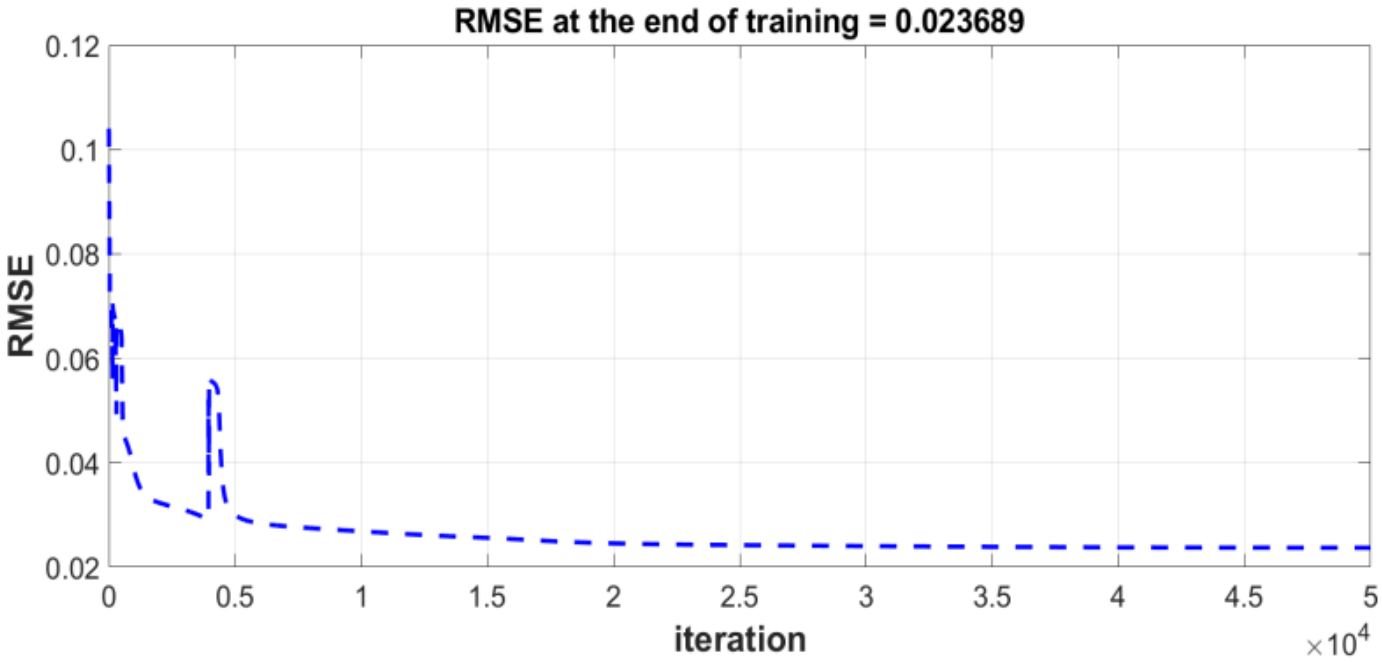

Fig. 8. Error plot of ANN training

B. Testing Results and Discussion

After training the ANN, optimal weight i.e. input to hidden layer weights and hidden to output layer weights are stored. These stored weights are used to estimate the software quality for the given five parameters i.e. Reliability, Efficiency, Usability, Maintainability, and Portability. A total of 28 samples for different software are taken where quality needs to be estimated using the proposed ANN.

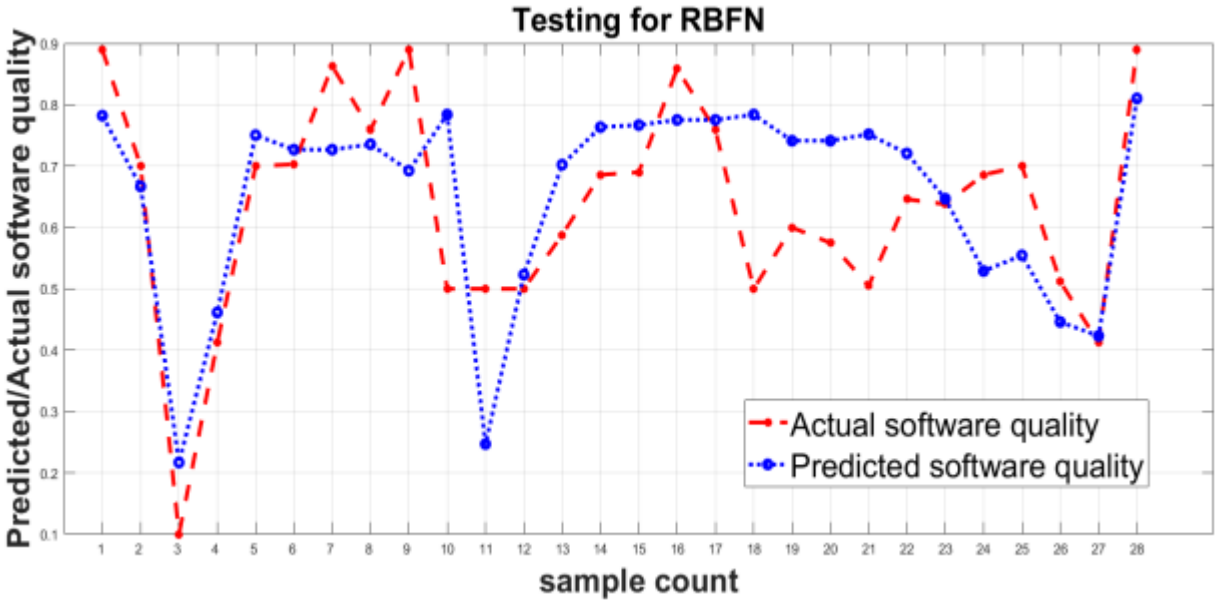

Fig. 9. RBFN testing results

Testing results for these 28 input samples are shown in Figure 9, 10 and 11 where we see that the estimated quality by ANN matches with the actual quality of software (desired quality), and better results are obtained with the customcoded MLNN approach. Some performance indexes [20] are used to show the efficacy of the proposed approach. All these approaches are also compared in terms of performance index Mean square error (MSE) and Mean Absolute Relative Error (MARE), Mean Relative Error (MRE) as these are popular measures to access the quality of prediction. MSE and MARE are estimated as follows:

$$MSE = \frac{\sum_{i=1}^{n}(y_d(i)-y(i))^2}{n} \quad (18)$$

$$MARE = \frac{\sum_{i=1}^{n}\left|\left(\frac{y_d(i)-y(i)}{y(i)}\right)\right|}{n} \quad (19)$$

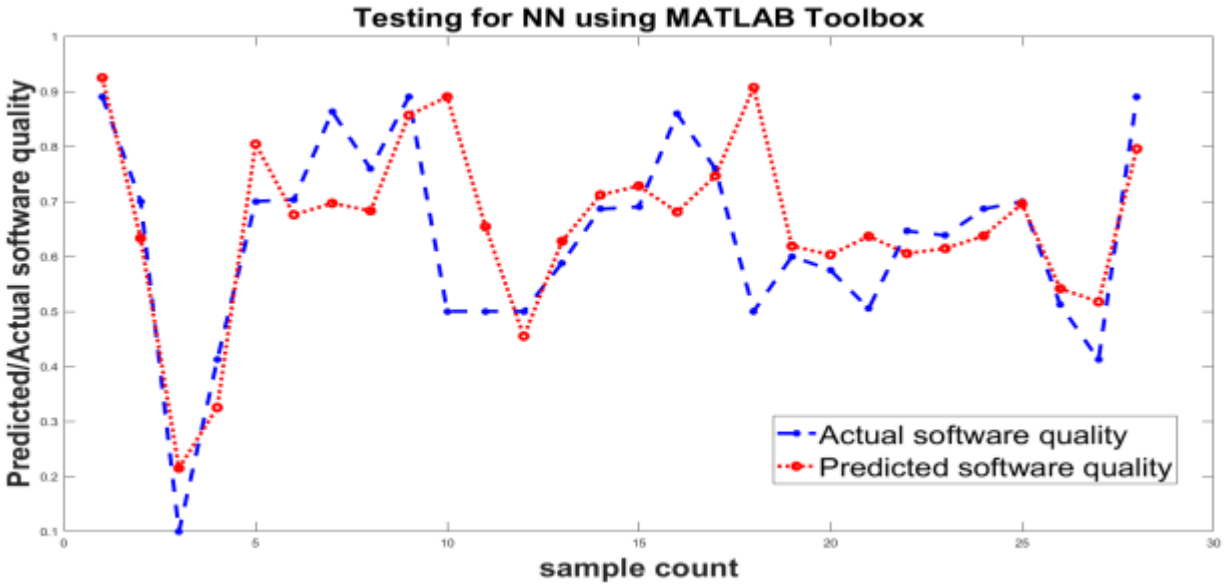

Fig. 10. NN Tool testing results

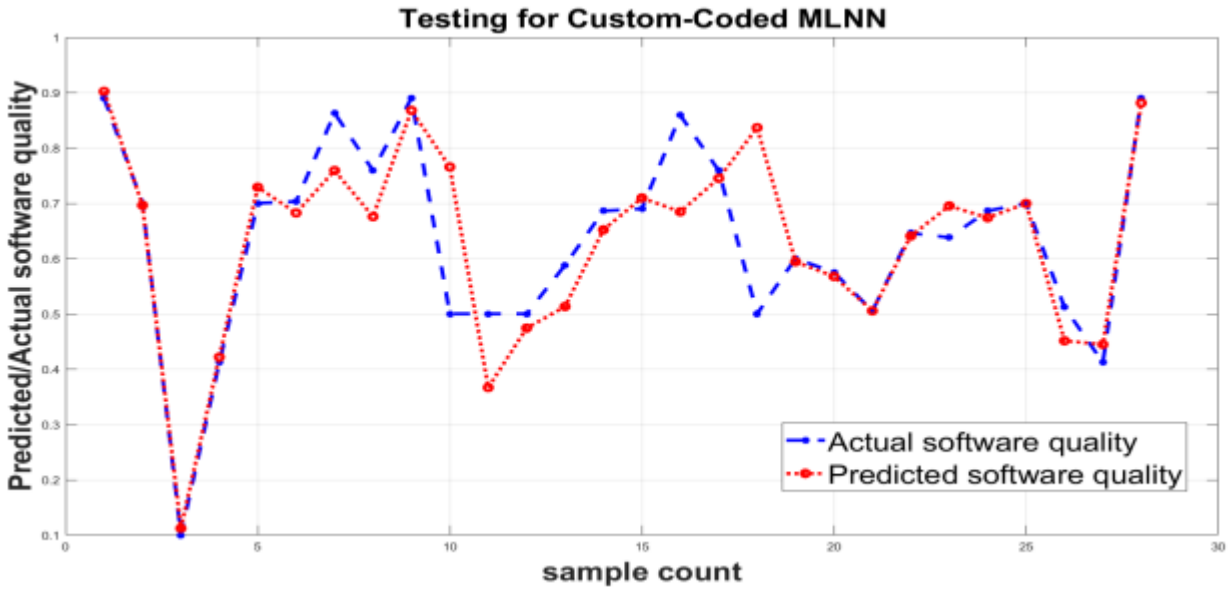

Fig. 11. Proposed MLNN testing results

The measured MSE and MARE for these three methods are shown in Table 2 and Table 3. In table II, we see that the lowest MSE is obtained in case of proposed MLNN whereas highest MSE in case of RBFN.

Table 2. Comparison of MSE among Various Methods

Sr. No.	Method	Mean Square Error (MSE)	
		Training	Testing
1	RBFN	0.0179	0.0198
2	NN Tool	0.0066	0.0121
3	Proposed MLNN	0.0023	0.0099

Table 3. Comparison of MARE among Various Methods

Sr. No.	Method	Mean Absolute Relative Error (MARE)	
		Training	Testing
1	RBFN	0.2185	0.2161
2	NN Tool	0.1327	0.1597
3	Proposed MLNN	0.0770	0.1510

Furthermore, Table 3 shows the MARE values obtained by three sachems where the proposed custom-coded NN achieves the lowest MARE value. Figure 12 and 13 shows the graphical representation of performance comparison among these three techniques. Figure 12 and 13 show that proposed MLNN achieves the lowest MSE, MARE, and MRE values in case of both training and testing phase which ensures the better prediction quality of this approach. Hence, we can conclude that the proposed

ANN architecture predicts the quality of the given unknown data-set which is not used in the training of the ANN with the highest accuracy. Thus the proposed scheme has the potential to be employed in realtime to save money and time for a customer.

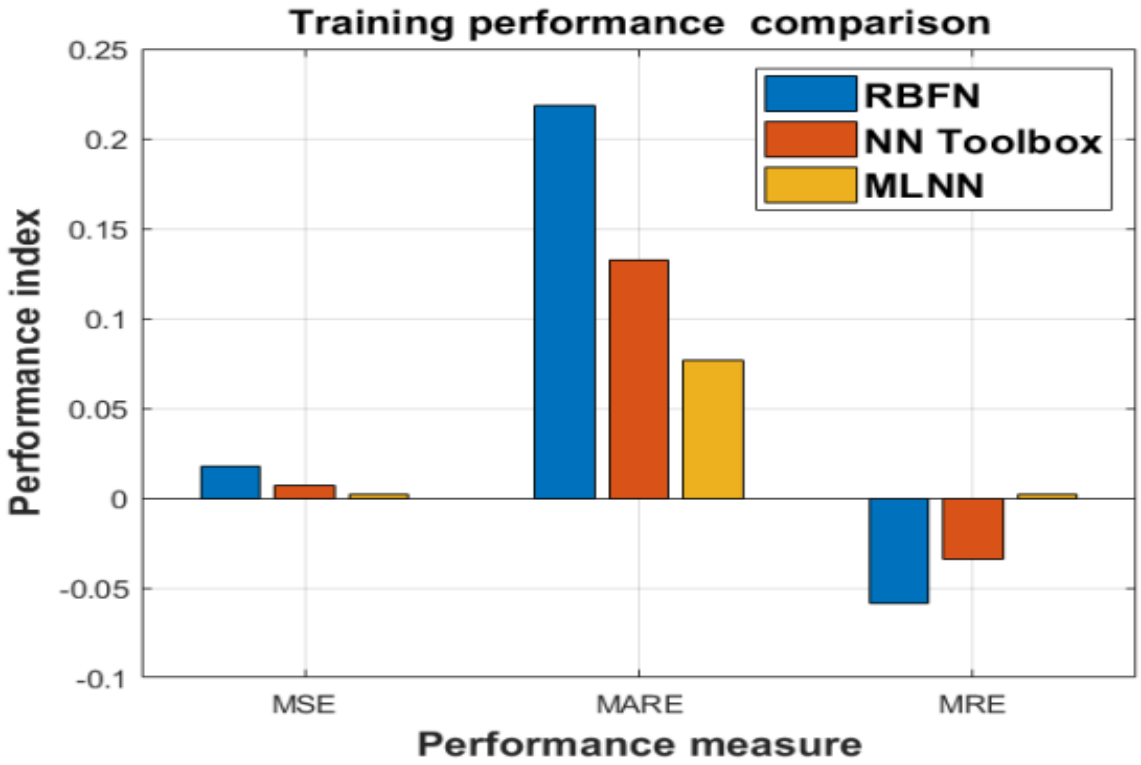

Fig. 12. Training performance comparison

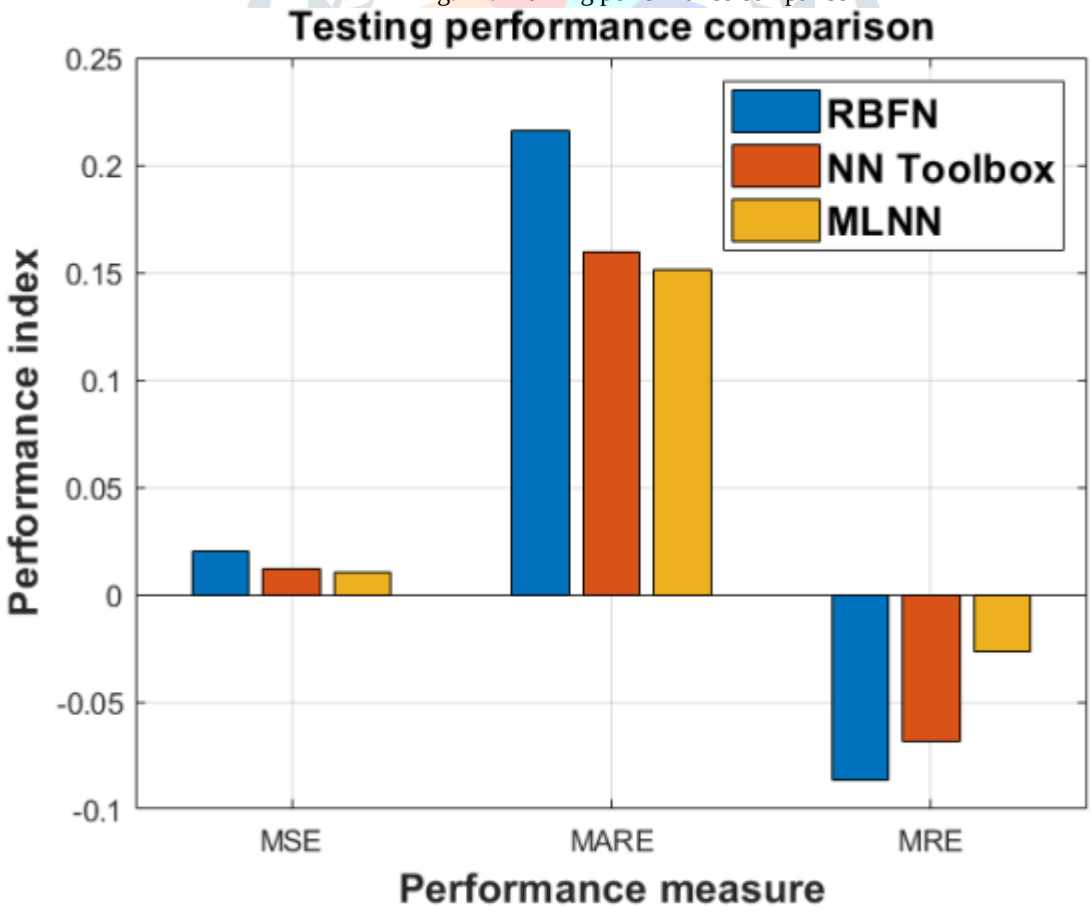

Fig. 13. Testing results comparison

VI. Conclusion

The variants of feed-forward neural networks namely multilayered neural network (MLNN) and RBFN have been proposed in this paper to predict the quality of software on the basis of five parameters namely Reliability, Usability, Efficiency, Maintainability, and Portability. RBFN has been used due to its universal approximation capability and linear parameter weights which gives a simpler form of ANN whereas MLNN can have nonlinear weight parameters which show good approximation performance. A data-set of a total of 128 software have been collected where 6 parameters (Reliability, Usability, Efficiency, Maintainability, Portability, and quality) are stored for each of the software, out of which 100 datasets have been used for training

of the ANN. 28 datasets have been used to test the efficacy and correctness of the trained ANN. The performance of MLNN, RBFN, and MATLAB NN Toolbox have been analyzed and compared which shows that the MLNN shows better performance in prediction which is justified by the obtained lowest values of performance measures MSE, MARE, and MRE. Simulation results show that the introduced MLNN structure correctly predicts the quality of software. In nutshell, the proposed scheme can be very effective for customers (who want to choose a particular software) in real-time to automatically predict the software quality based on the five parameters only. The future direction of this work includes the performance enhancement of the software quality approach by incorporating linguistic information.

REFERENCES

[1] Boehm, B. W., John R. B., and Mlity L. 1976. Quantitative Evaluation of Software Quality. In Proceedings: 2nd International Conference on Software Engineering.

[2] Pal J., and Bhattacherjee V. 2015. Software Quality Prediction using Fuzzy Rule Based System. International Journal of Current Research. 7(12): 24181-24185.

[3] Thwin, M. M. T., and Tong-Seng Q. 2005. Application of Neural Networks for Software Quality Prediction using Object-Oriented Metrics. Journal of systems and software 76(2): 147-156.

[4] Aggarwal, C. C. 2018. Neural Networks and Deep Learning. Springer. 10.

[5] Ruder, S. 2016. An Overview of Gradient Descent Optimization Algorithms. arXiv preprint arXiv: 1609.04747.

[6] Buhmann, M. D. 2003. Radial Basis Functions: Theory and Implementations. Cambridge University Press. 12.

[7] Park, J., and Irwin W. S. 1991. Universal Approximation using Radial-Basis-Function Networks. Neural Computation. 3(2): 246-257.

[8] Rosenblum, M., Yaser Y., and Larry S. D. 1996. Human Expression Recognition from Motion using A Radial Basis Function Network Architecture. IEEE Transactions on Neural Networks. 7(5): 1121-1138.

[9] Khoshgoftaar, T. M., et al. 1997. Application of Neural Networks to Software Quality Modeling of A Very Large Telecommunications System. IEEE Transactions on Neural Networks. 8(4): 902-909.

[10] Schneidewind, N. F. 1992. Methodology for Validating Software Metrics. IEEE Transactions on Software Engineering. 18(5): 410-422.

[11] Briand, L. C., Walcelio L. M., and Jurgen W. 2002. Assessing The Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. IEEE Transactions on Software Engineering. 28(7): 706-720.

[12] El Emam, K., Walcelio M, and Javam C. M. 2001. The Prediction of Faulty Classes using ObjectOriented Design Metrics. Journal of Systems and Software. 56(1): 63-75.

[13] Catal C., Sevim U., and Diri B. 2009. Software Fault Prediction of Unlabeled Program Modules, In Proc. World Congress on Engineering, London, U.K. 1.

[14] LeCun, Y., et al. 1988. A Theoretical Framework for Back-Propagation. In Proc. of Connectionist Models Summer School. 1.

[15] Zeng, H., and David R. 2004. Estimation of Software Defects Fix Effort using Neural Networks." In Proc. of 28th Annual International Computer Software and Applications Conference. IEEE, 2.

[16] Khoshgoftaar, T. M., and Robert M. S. 1994. Improving Neural Network Predictions of Software Quality using Principal Components Analysis. In Proc. of International Conference on Neural Networks, IEEE, 5.

[17] Kumar, R., Suresh R., and Jerry L. T. 1998. NeuralNetwork Techniques for Software-Quality Evaluation. In Proc. of International Symposium on Product Quality and Integrity, IEEE.

[18] Behara, R. S., Warren W. F., and Jos G. L. 2002. Modelling and Evaluating Service Quality Measurement using Neural Networks. International Journal of Operations & Production Management.

[19] Bhuyan, M. K., Durga P. M., and Srinivas S. 2016. Software Reliability Assessment using Neural Networks of Computational Intelligence Based on Software Failure Data. Baltic Journal of Modern Computing. 4(4).

[20] Singh, Y., et al. 2010. Predicting Software Development Effort using Artificial Neural Network. International Journal of Software Engineering and Knowledge Engineering. 20(3): 367-375.

[21] Singh, Y., Bhatia P. K., and Sangwan, O. P. 2011. Software Reusability Assessment using Soft Computing Techniques. ACM SIGSOFT Software Engineering Notes. 36(1): 1-7.

[22] Gayathri, M., and Sudh, A. 2014. Software Defect Prediction System using Multilayer Perceptron Neural Network with Data Mining. International Journal of Recent Technology and Engineering. 3(2): 54-59.

[23] Khoshgoftaar, T. M., and Allen, E. B. 1998. Neural Networks for Software Quality Prediction. Computational Intelligence in Software Engineering. 16: 33-63.