# Pre-Training of an Artificial Neural Network for Software Fault Prediction

Moein Owhadi-Kareshk[1], Yasser Sedaghat[2], Mohammad-R. Akbarzadeh-T.[1*]

[1] Center of Excellence on Soft Computing and Intelligent Information Processing (SCIIP)
[2] Dependable Distributed Embedded Systems (DDEmS) Laboratory
Department of Computer Engineering
Ferdowsi University of Mashhad (FUM), Mashhad, Iran
*owhadi@mail.um.ac.ir, y_sedaghat@um.ac.ir, akbazar@um.ac.ir*

*Abstract*— **Software fault prediction is one of the significant stages in the software testing process. At this stage, the probability of fault occurrence is predicted based on the documented information of the software systems that are already tested. Using this prior knowledge, developers and testing teams can better manage the testing process. There are many efforts in the field of machine learning to solve this classification problem. We propose to use a pre-training technique for a shallow, i.e. with fewer hidden layers, Artificial Neural Network (ANN). While this method is usually employed to prevent over-fitting in deep ANNs, our results indicate that even in a shallow network, it improves the accuracy by escaping from local minima. We compare the proposed method with four SVM-based classifiers and a regular ANN without pre-training on seven datasets from NASA codes in the PROMISE repository. Results confirm that the pre-training improves accuracy by achieving the best overall ranking of 1.43. Among seven datasets, our method has higher accuracy in four of them, while ANN and support vector machine are the best for two and one datasets, respectively.**

***Keywords-component; Pre-Training; Shallow Artificial Neural Network; Software Fault Prediction.***

## I. INTRODUCTION

Software systems play a key role in today's life. There are many software applications whose fault occurrence can lead to critical problems. In these safety-critical applications, the number of faults, errors, and failures should be minimized as much as possible. One of the most effective stages to build a fault-free software is predicting faults in under-developing software systems. Employing this prediction, the testing teams can manage their resources and focus on the software modules with higher expected probability of faults. The software fault prediction is based on the data of software systems that are already developed and tested. The features of an under-testing software are compared with the developed software systems; and based on the faults of those systems, an estimation for the number of faults in the under-testing software is calculated.

From the statistics and machine learning viewpoints, if we aim to predict the number of faults, the problem is a regression problem; and if determining the faulty or fault-free status of a module is important, the problem is a binary classification problem. In this paper, we focus on the second case. There are many efforts to predict the faults of a new code or software in the literature. In the most basic studies, the logistic regression is used [1] [2] [3]. More advanced learning algorithms such as Support Vector Machines (SVMs) [4] [5], Decision Trees [6] [7] [8] and Naïve Bayes [9] [10] [11] [12] [13] are also employed. ANNs have also been successfully applied to the problem [14] [15]. Recently, several machine learning techniques are proposed for software fault prediction problem [16] [17] [18] [19]. In one of the state-of-the-art research works [20], several feature selection and data balancing methods are employed to enhance the prediction. In continuation of using ANNs, we use the idea of pre-training in a shallow network to improve the accuracy. Pre-training in shallow networks is proved to be beneficial previously in other problems as well [21].

In this paper, we propose using a pre-trained ANN to solve the software fault prediction as a binary classification problem. Pre-training is conventionally employed for preventing overtraining in the deep neural networks [22] [23] [24], but we use this technique for a shallow ANN and show that it practically improves the classification accuracy. In the pre-training technique, an unsupervised neural network determines an estimation of the weights and bias before training the ANN using the labels. For this purpose, a 2-layer unsupervised neural network, such as Autoencoders (AEs) [23] and Restricted Boltzmann Machines (RBMs) [22] [23] are used. These shallow ANNs are stacked on each other by greedy layer-wise algorithm [25] and create a multi-layer unsupervised ANN. In this algorithm, a 2-layer neural network is trained first and then, its weights and bias are fixed. After that, the output of this neural network is employed to train another subsequent 2-layer neural network. This procedure is continued until the final ANN with an appropriate number of layers is created. After that, an output layer is added to this ANN. Lastly, the network is trained and the weights and bias of the other layers are fine-tuned using labels in a supervised fashion. The initial point of searching for

* Also currently with Department of EECS, University of California, Berkeley, CA 94720-1776, USA.

**978-1-5386-0804-3/17/$31.00 ©2017 IEEE**

Table I. MCCABE COMPLEXITY METRICS

| No. | Sub- Metrics | Description |
|---|---|---|
| 1 | Cyclomatic complexity | This metric is calculated by $v(g) = e - n + 2$<br>Where, $G$ is the flow graph of the code, $e$ is the number of arcs and $n$ is the number of nodes. |
| 2 | Essential complexity | The equation $ev(g) = v(g) - m$ is employed for calculating this metric where, $m$ is the number of D-structured primes |
| 3 | Design complexity | The flow graph is reduced to eliminate the chunks that have not influence on the interrelationship between the code segments. This metric is indicated by $ev(g')$, where $g'$ is the reduced graph. |
| 4 | Lines of Code | The number of lines regarding to McCabe metrics |

the parameters is assigned randomly in the last layer. For the other layers, the initial point is set to the parameters of the pre-trained network, which is generated by the greedy layer-wise algorithm.

To create an unsupervised ANN using the greedy layer-wise algorithm, two types of 2-layer ANN can be employed, RBMs and AEs. RBMs are a special version of Boltzmann Machines, which there is no connection between the neurons of each layer. Using this restriction, these models can be trained by Contrastive Divergence (CD) [26], Persistent CD (PCD) [27], fast PCD [28] and parallel tempering [29]. An energy function is defined for RBMs, training of an RBM means finding the weights and bias such that the energy function is minimized. AEs are trained by stochastic gradient descent [30]. These networks aim to create a new representation of data and then, using this representation, reconstruct the original input. Minimizing the difference between the reconstruction and the original input is the target of AEs. If the size of the new representation is bigger than the input (i.e. have more neurons) the AE is so-called over-complete and otherwise, the model is under-complete.

A regular AE usually cannot extract meaningful features to create a good representation. In the over-complete AEs, the model can simply copy the original input to the new representation. Also, an under-complete AE does not necessarily extract a suitable representation. For solving this problem, an extra constraint is considered for the training of the AEs. One of the popular methods is sparsity regularization [31] that aims to force the neurons of the hidden layer to be zero [32] [33]. The Contractive AEs (CAEs) [34] aim to minimize the derivatives of the hidden layer and in the other words, minimize the sensitivity of the model to its input. Denoising AE (DAE) [35] [36] is among the most effective ways to regularize AEs. In this variant, an artificial noise in mounted on the data and it is expected that the model can reconstruct the original input from this noisy version of the data. Therefore, the model cannot minimize the reconstruction error by *memorizing* the input and it should learn the overall structure (manifold) of the data. We use a DAE to pre-train a shallow ANN for predicting the faults of software systems. Moreover, our method has only six neurons in its hidden layer and therefore, the computational cost is low.

We study our method with seven datasets from PROMISE [37] data repository of NASA [38]. The results show that the idea of pre-training improves the accuracy in comparison with the traditional ANN and SVM with different feature extraction methods. According to the definition of our problem, the execution time is not a critical measure and hence, it is not reported here. In the next section, the metrics for extracting data from code are explained. In Section 3, we introduce the pre-training technique for shallow networks. Section 4 consists of the results and their analysis. Finally, in Section 5, we have a conclusion and introduce some suggestions to improve the results of this paper.

## II. SOFTWARE FAULT PREDICTION AND METRICS

In the process of developing a software system, unit testing is applied when a module is created. In this phase, a testing team or the developers aim to find the faults of the modules and document these faults and their properties. The idea behind the software fault prediction is that we can use these documents to have more effective software testing for under-developing modules. Employing the data of previous tests, the existence or even the number of faults of an under-developing module can be estimated. Using this estimation, the testing team can divide the resources based on the expected number of faults. To have a suitable prediction, several appropriate features should be extracted from the code, which have direct and meaningful effects on the occurrence of faults. Several feature sets are proposed, but the McCabe [39] and Halstead [40] are among the most well-known ones.

The McCabe metrics consist of four subsets as Table I. The essential concept in these metrics is the flow graph. The nodes in this directed graph are the code statements and the arcs are the flow control between the statements. A D-structured prime also indicates the sub-graphs of the flow graph which has only one input and one output. The Halstead metrics are also introduced to measure the complexity of the code and do not consider the platforms. Its parameters can be divided into two main groups, basic and derived metrics. Table II illustrates these metrics. The first four cases are the basic metrics and the rest of the table consists of the derived ones.

Table II. HALSTEAD COMPLEXITY MEASURES

| Item | Feature |
|---|---|
| $N_1$ | The total number of operators ($N_1$) |
| $N_2$ | The total number of operands ($N_2$) |
| $\eta_1$ | The number of distinct operators ($\eta_1$) |
| $\eta_2$ | The number of distinct operands ($\eta_2$) |
| $N$ | Program length ($N = N_1 + N_2$) |
| $\eta$ | Program vocabulary ($\eta = \eta_1 + \eta_2$) |
| $V$ | Volume ($V = N * \log_2 \eta$) |
| $D$ | Difficulty ($D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$) |
| $E$ | Effort ($E = D * V$) |
| $\widehat{N}$ | Calculated program length ($\widehat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$) |
| $T$ | Time required to program ($T = \frac{E}{18}$) |
| $B$ | Number of delivered bugs ($B = \frac{V}{3000}$) |

## III. PRE-TRAINING OF ANNS WITH DAES

In this section, we introduce the pre-training technique which we use for software fault prediction. The pre-training technique is employed for initialization of an ANN. Therefore, instead of starting the gradient descent from a random point, a data-driven starting point is calculated in an unsupervised fashion. In this paper, we employ DAEs that are a variation of AEs that mount artificial noise on data as a regularization approach. These mathematical models can extract meaningful information from raw data. Firstly, a DAE is trained by gradient descent. After the training phase, this network can create a new transformation from its input data. Afterwards, the weights and bias of this network can be considered as a good unsupervised estimation for the weights and bias of the network in the supervised fashion. Therefore, the searching procedure of the parameters in the supervised neural network is started from the weights and bias of DAE. After that, an output layer is added to the network. In the last step, the learning process is starting again for fine-tuning the weights and bias of the hidden layer together with calculating the weights and bias of the last layer. Traditionally, the pre-training method is employed for deep neural networks, but we employ it in a shallow network with just one hidden layer and show that this method improves the results in comparison with randomly standard ANNs and several SVM-based classifiers.

AE is a 2-layer unsupervised ANN. As Fig. 1, this network receives the input data $x$ and the latent variable $y$ is created using the transformation $f(x)$. This procedure is so-called *encoding* and layer $y$ is often called a new representation or *code*. After that, the network aims to reconstruct its input by transformation $g(y)$ and creates the reconstruction layer $z$, which is called the *decoding* process.

As in (1), a nonlinear function $s$ followed by an affine function (with weight $w$ and bias $b$) are applied on $x$ for creating a new representation of data.

$$y = f(x) = s(w^T x + b) \quad (1)$$

After that, by using a nonlinear function $g$, weights $w'$ and bias $d$, the input is reconstructed as shown in (2). The tied weights, $w'=w^T$, is used in (2). We use the Sigmoid function as $s$ and $g$ for both encoding and decoding phases.

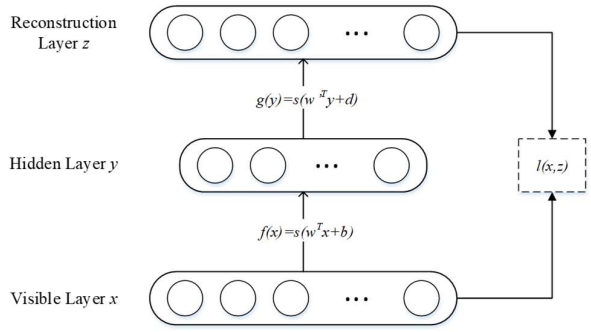


Figure 1. Regular AutoEncoder [41]

$$z = g(y) = g(w'^T y + d) \quad (2)$$

If representation $y$ consists of meaningful information about the input $x$, it is expected that $x$ and its reconstruction $z$ are close as much as possible. Hence, the cost function of AE is the difference between the input and its reconstruction. Equation (3) is employed for this purpose.

$$l(x, z) = \|z - x\|^2 \quad (3)$$

A regular AE without any extra constraints usually cannot extract meaningful and robust features and therefore, several approaches for its regularizing are proposed. In this paper, we use DAE as one of the most effective approaches that mounts an artificial noise on the input data by function $q$. A robust model should be capable of denoising, i.e. removing the artificial noise, by learning the main structure (manifold) of the data. If the model tries to just memorize its input, it cannot reconstruct it after mounting the artificial noise. This noise can be Masking, Salt & Pepper (for images), Additive white Gaussian noise (AWGN), etc. In this paper, the masking noise is employed. Fig. 2 illustrates a DAE. It is important to note that adding noise to the data is only applied in the training phase and after that, in the testing phase, the clean data are fed to the network.

Using a DAE, we estimate the weights and bias of the final supervised ANN in an unsupervised fashion. The pre-training technique prevents the over-fitting in ANNs. In this technique, the network actually learns the distribution of data and can distinguish between the actual input data and random input. The reconstruction error is low for the data on the manifold of the training dataset. In the subsequent fine-tuning phase, the weights and bias of the first layer are improved and the weights and bias of the second supervised layer are learned. The error back-propagation is much effective in the last layer and the learning is slower for the first layers. It is because the derivatives are becoming close to zero in these layers. Hence, the pre-training is a good option for the first layers.

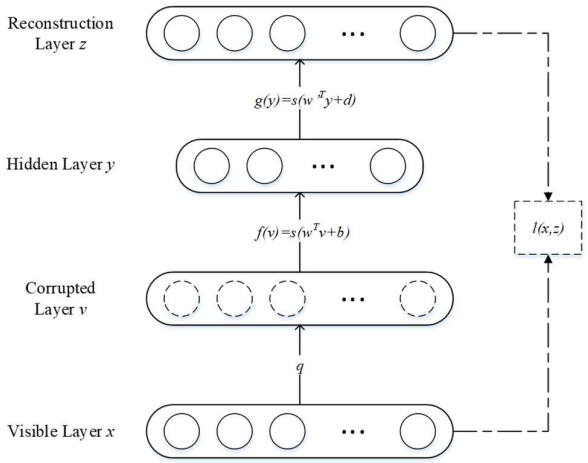


Figure 2. Denoising AutoEncoder [41]

## IV. EXPERIMENTS

In this section, we bring the results of the implementation of the pre-trained ANN for software fault prediction. The configuration and conditions of experiments are described in Subsection *A*. In Subsection *B*, we present the results and analyze them. Finally, the sensitivity of the proposed method to its main hyper-parameter is investigated in Subsection *C*.

### *A. Configuration of Experiments*

There are two main groups of employed datasets for software fault prediction, private and public. Although several private datasets are available for us, we prefer to use public datasets because comparing different methods are easier on these types of datasets. We employ seven datasets of PROMISE repository [37] that are introduced in Table III. These datasets are created by NASA [38] and consist of McCabe [39], Halstead [40] and some other basic features. Since the labels are just set to False and True values, this is a binary classification problem.

The proposed method is implemented in MATLAB [42] and to have a trusted comparison with SVM, LibSVM [43], as a well-known library is used. A system with Quad-Core Core i-7 4.00 GHz CPU, 8GB of RAM and 100GB SSD hard disk is employed to execute the code. The results are compared with the percentage of accuracy as (4).

$$\text{Accuracy} = 100 * \frac{\#\ Correctly\ classified\ Data}{\#\ All\ Data} \qquad (4)$$

The proposed method is compared with the following methods:

- *SVM*: The Support Vector Machine (SVM) algorithm with Radial Basis Function (RBF) kernel.
- *PCA-SVM*: Extracting features by Principal Component Analysis (PCA) and then, classifying with kernel SVM.
- *KPCA-SVM*: Extracting features by Kernel PCA (KPCA) and then, using SVM for classification.
- *AE-SVM*: Extracting features by AE and then, using SVM for classification.
- *ANN*: A regular ANN without pre-training.

The *Pre-ANN* is our proposed method, which is an ANN with pre-training by DAE. In the above methods, the number of latent variables is considered as same as the proposed method that has six neurons in its hidden layer. We use 5-fold cross-validation to ensure the robustness of the results.

### *B. Results*

Table IV presents the results of the experiments. As the table indicates, our proposed method has higher accuracy on four datasets (including the biggest one) and for the other three datasets; it stands in the second rank. The SVM is ranked first only on the pc1 dataset that the proposed method and the regular neural network are in the second place. The Mean of Ranking for our proposed method is 1.43.

### *C. Sensitivity to the Hyper-Parameter*

If an algorithm is too sensitive to its hyper-parameters, finding the appropriate values become a challenge. Therefore, analyzing of the hyper-parameters is important. The main hyper-parameter of our proposed method is the number of neurons in the hidden layer or equivalently, the number of latent variables in the DAE. Based on the illustrated results in the Fig. 3, most datasets are not sensitive to the number of hidden neurons. However, the sensitivity is high for kc2 and kc3.

Table III. THE DATASETS FROM PROMISE REPOSITORY

| No. | Datasets | No. of Instances | No. of Features |
|---|---|---|---|
| 1 | pc1 | 705 | 37 |
| 2 | pc2 | 745 | 36 |
| 3 | pc3 | 1077 | 37 |
| 4 | pc4 | 1458 | 37 |
| 5 | kc2 | 522 | 21 |
| 6 | kc3 | 194 | 39 |
| 7 | jm1 | 7782 | 21 |

Table IV . THE ACCURACY (RANK) OF OUR PROPOSED METHOD AND PREVIOUS METHODS

| Datasets | *SVM* | *PCA-SVM* | *KPCA-SVM* | *AE-SVM* | *ANN* | *Pre-ANN* |
|---|---|---|---|---|---|---|
| *pc1* | **92.62±2.2 (1)** | 78.72±2.2 (4) | 80.57±5.3 (3) | 77.45±2.5 (5) | 91.06±3.1 (2) | 91.06±1.6 (2) |
| *pc2* | 97.58±1.5 (3) | 92.62±0.9 (4) | 91.14±0.8 (5) | 86.31±5.4 (6) | 97.72±1.0 (2) | **97.82±1.1 (1)** |
| *pc3* | 86.35±1.8 (2) | 75.20±3.0 (6) | 77.44±2.5 (4) | 75.40±4.0 (5) | 86.07±3.0 (3) | **87.88±0.4 (1)** |
| *pc4* | 87.38±1.6 (3) | 78.05±6.7 (6) | 78.25±6.6 (5) | 78.53±7.7 (4) | **89.17±2.3 (1)** | 87.66±3.6 (2) |
| *kc2* | 77.77±3.7 (6) | 79.32±5.0 (3) | 77.96±3.2 (5) | 78.14±4.9 (4) | **83.51±3.2 (1)** | 81.06±5.3 (2) |
| *kc3* | 80.94±4.6 (2) | 68.03±7.0 (5) | 64.97±5.8 (6) | 68.54±3.9 (4) | 79.38±8.7 (3) | **81.46±6.1 (1)** |
| *jm1* | 70.78±0.9 (3) | 70.08±1.6 (4) | 69.74±0.9 (5) | 68.93±1.6 (6) | 78.56±1.6 (2) | **78.82±0.7 (1)** |
| *Mean of Ranking* | *2.86* | *4.57* | *4.71* | *4.86* | *2* | ***1.43*** |

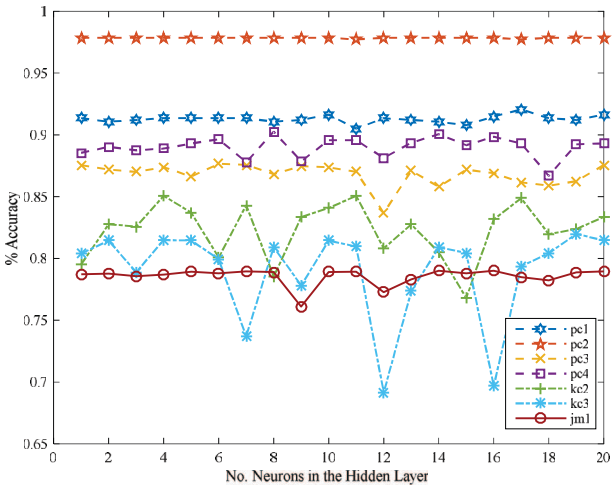


Figure 3. The Sensitivity of the proposed method to the number of neurons in the hidden layer

## V. Conclusion and Future Works

In this paper, we aim to deal with the problem of software fault prediction. In this challenge, the status of faults of under-developing software systems is predicted based on the information of the software systems that are developed and tested already. This estimation increases the performance of the testing teams since they can focus on the modules that are expected to be faulty. There are several efforts to solve this problem in the field of machine learning. Two main lines of the previous works are the statistical machine learning (e.g. linear regression, logistic regression, SVM, etc.) and soft computing (e.g. ANNs, etc.). The ANNs usually can describe a more complex model and are suitable for today's complicated problems.

We propose to use the pre-training technique to improve the performance of the shallow ANN for software fault predicting. In this method, instead of starting the training procedure from a random vector of weights and bias, the weights and bias of a trained DAE are employed which is learned without labels. The idea of unsupervised pre-training and subsequent supervised fine-tuning can increase the accuracy as the main measure of the software fault prediction. Because of the importance of the accuracy measure in this problem, the execution time is usually not considered. However, although this technique increases the execution time, growing the execution time is not a dilemma since the software fault prediction is not a real time problem. We compare our proposed method with SVM, SVM on the extracted features and ANNs without pre-training. To ensure the robustness of experiments, we use 5-fold cross-validation. The results show that the SVM is the best algorithm only on one dataset and ANN has the highest accuracy for two datasets. Our proposed method is the most effective technique for other four datasets. In the continuation of this work, we suggest improving the accuracy by using deep learning models. Using ensemble learning (e.g. Bagging and Boosting) can also improve the performance.

## VI. References


[1] T. Khoshgoftaar, "An application of zero-inflated Poisson regression for software fault prediction," in: Proc. 12th Intl. Symp. Software Reliability Eng., November, 2001.

[2] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, "Benchmarking classification models for software defect prediction: a proposed framework and novel findings," IEEE Trans. Softw. Eng. 34, 2008.

[3] N. Nagappan, T. Ball, "Static analysis tools as early indicators of pre-release defect density," in: Proc. Intl. Conf. Software Eng., 2005.

[4] B. Twala, "Software faults prediction using multiple classifiers," in: 2011 3rd Int. Conf. Comput. Res. Dev., 4, 2011.

[5] L. Yu, "An evolutionary programming based asymmetric weighted least squares support vector machine ensemble learning methodology for software repository mining," Inf. Sci. 191, 2012.

[6] L. Guo, Y. Ma, B. Cukic, H.S.H. Singh, "Robust prediction of fault-proneness by random forests," in: 15th Int. Symp. Softw. Reliab. Eng., 2004.

[7] A. Kaur, R. Malhotra, "Application of random forest for predicting fault prone classes," in: International Conference on Advanced Computer Theory and Engineering, Thailand, December 20–22, 2008.

[8] T. Khoshgoftaar, E. Allen, "Model software quality with classification trees," Recent Adv. Reliab. Qual. Eng., 2001.

[9] A. Bener, B. Turhan, "Analysis of Naive Bayes' assumptions on software fault data: an empirical study," Data Knowl. Eng. 68, 2009.

[10] K. Dejaeger, T. Verbraken, B. Baesens, "Prediction Models Using Bayesian Network Classifiers," 39, 2013.

[11] B. Diri, C. Catal, U. Sevim, "Practical development of an Eclipse-based software fault prediction tool using Naive Bayes algorithm," Expert Syst. Appl. 38, 2011.

[12] A. Okutan, O.T. Yıldız, "Software defect prediction using Bayesian networks," Empirical Softw. Eng., 2012.

[13] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, "A general software defect-proneness prediction framework," IEEE Trans. Softw. Eng. 37, 2011.

[14] T. Khoshgaftaar, E.D. Allen, J.P. Hudepohl, S.J. Aud, "Application of neural networks to software quality modeling of a very large telecommunications system," IEEE Trans. Neural Netw. 8 (4), 1997.

[15] A.T. Mısırlı, A.B. Bener, B. Turhan, "An industrial case study of classifier ensembles for locating software defects," Softw. Qual. J. 19, 2011.

[16] W. Li, Z. Huang, and Q. Li, "Three-way decisions based software defect prediction," Knowledge-Based Systems, 91: 263–274, 2016.

[17] H. B. Yadav,and D. K. Yadav, "A fuzzy logic based approach for phase-wise software defects prediction using software metrics," Information and Software Technology, 63: 44-57, 2015.

[18] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," Information and Software Technology, 59: 170-190, 2015.

[19] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," Applied Soft Computing, 27: 504-518, 2015.

[20] Yohannese, Chubato Wondaferaw, and Tianrui Li, "A Combined-Learning Based Framework for Improved Software Fault Prediction," International Journal Of Computational Intelligence Systems 10.1, 2017.

[21] M. Souzanchi-K, M. Owhadi-Kareshk and M. R. Akbarzadeh -T., "Control of elastic joint robot based on electromyogram signal by pre-trained Multi-Layer Perceptron," International Joint Conference on Neural Networks (IJCNN), 2016.

[22] Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets," Neural computation 18, no. 7, 2006.

[23] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," Science 313, no. 5786, 2006.

[24] Bengio, Yoshua, "Learning deep architectures for AI," Foundations and trends in Machine Learning 2, no. 1, 2009.

[25] Bengio, Yoshua, Pascal Lamblin, Dan Popovici, and Hugo Larochelle, "Greedy layer-wise training of deep networks," Advances in neural information processing systems 19, 2007.

[26] Hinton, Geoffrey E, Training products of experts by minimizing contrastive divergence," Neural computation 14, no. 8, 2002.

[27] Tieleman, Tijmen, "Training restricted Boltzmann machines using approximations to the likelihood gradient," In Proceedings of the 25th international conference on Machine learning, ACM, 2008.

[28] Tieleman, Tijmen, and Geoffrey Hinton, "Using fast weights to improve persistent contrastive divergence," In Proceedings of the 26th Annual International Conference on Machine Learning, ACM, 2009.

[29] Cho, KyungHyun, Tapani Raiko, and Alexander Ilin, "Parallel tempering is efficient for learning restricted Boltzmann machines," In IJCNN, 2010.

[30] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams, "Learning internal representations by error propagation," No. ICS-8506. California Univ San Diego La Jolla Inst For Cognitive Science, 1985.

[31] Poultney, Christopher, Sumit Chopra, and Yann L. Cun, "Efficient learning of sparse representations with an energy-based model," In Advances in neural information processing systems, 2006.

[32] Boureau, Y-lan, and Yann L. Cun, "Sparse feature learning for deep belief networks," In Advances in neural information processing systems, 2008.

[33] Zou, Will Y., Andrew Y. Ng, and Kai Yu, "Unsupervised learning of visual invariance with temporal coherence," In NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning. 2011.

[34] Rifai, Salah, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," In Proceedings of the 28th International Conference on Machine Learning, 2011.

[35] Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol, "Extracting and composing robust features with denoising autoencoders," In Proceedings of the 25th international conference on Machine learning, ACM, 2008.

[36] Vincent, Pascal, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," The Journal of Machine Learning Research 11, 2010.

[37] http://openscience.us/repo/defect/

[38] http://nasa.gov

[39] T.A. McCabe, "Complexity measure," IEEE Trans. Softw. Eng. 2 (4) (1976) 308–320.

[40] M. Halstead, "Elements of Software Science," Elsevier, 1977.

[41] M. Owhadi-K., M.-R. Akbarzadeh-T, "Representation Learning by Denoising Autoencoders for Clustering-based Classification," In Computer and Knowledge Engineering (ICCKE), 5th International eConference on, 2015.

[42] MATLAB 2014, The MathWorks, Inc.

[43] C.-C. Chang and C.-J. Lin. "LIBSVM : a library for support vector machines," ACM Transactions on Intelligent Systems and Technology, 2011.