Hindawi
Scientific Programming
Volume 2021, Article ID 6662932, 17 pages
https://doi.org/10.1155/2021/6662932

Research Article

Impact of Parameter Tuning for Optimizing Deep Neural Network Models for Predicting Software Faults

Mansi Gupta, Kumar Rajnish, and Vandana Bhattacharjee

Department of Computer Science and Engineering, BIT Mesra, Ranchi, India

Correspondence should be addressed to Mansi Gupta; jv.mansi@gmail.com

Received 13 October 2020; Revised 21 March 2021; Accepted 2 June 2021; Published 12 June 2021

Academic Editor: Jianping Gou

Copyright © 2021 Mansi Gupta et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Deep neural network models built by the appropriate design decisions are crucial to obtain the desired classifier performance. This is especially desired when predicting fault proneness of software modules. When correctly identified, this could help in reducing the testing cost by directing the efforts more towards the modules identified to be fault prone. To be able to build an efficient deep neural network model, it is important that the parameters such as number of hidden layers, number of nodes in each layer, and training details such as learning rate and regularization methods be investigated in detail. The objective of this paper is to show the importance of hyperparameter tuning in developing efficient deep neural network models for predicting fault proneness of software modules and to compare the results with other machine learning algorithms. It is shown that the proposed model outperforms the other algorithms in most cases.

1. Introduction

Deep neural network (DNN) models have gained a lot of attention due to their outstanding performance in many tasks. The main aim of this study is to build deep neural network models for software fault prediction by focusing on those aspects of training which impact the classifier performance the most. A comparison is made between the performances of deep neural network and other classification techniques such as naïve Bayes, random forest, and decision tree.

Software fault prediction is one of the major areas of investigation in the area of software quality [1]. Fault prediction being an intricate area of research, many software researchers and practitioners have experimented on numerous ways of predicting faults in software [2]. The accurate prediction of faults in code plays a very important role as it can help in reducing test effort and costs and improve the quality of software to an extent. The main cause of failure of a software product is the defect in the code that occurs during the implementation of the software [3]. In an organization where the budget is limited, the software manager instead of going for complete software testing prefers for testing those modules that are fault prone using fault predictors.

Software fault prediction methods initially used code metrics or simply software metrics and statistical approach for fault prediction. Thereafter, the focus shifted to soft computing and machine learning (ML) techniques which took over all the prediction techniques [4]. In software code metrics-based methods, internal attributes of the software were measured for fault prediction. The commonly used software metrics' suites were Quality Model For Object Oriented Design (QMOOD) metric suite [5], Chidamber and Kemerer (CK) metric suite [6], Metrics for Object Oriented Design (MOOD) metric suite [7], etc. From the perspective of machine learning, fault prediction comes under the classification task in which it discriminates faulty and nonfaulty modules [8]. Some representative ML methods are ensemble, support vector machine (SVM), naive Bayes, logistic regression, decision table, etc., and a review of such techniques applied to software fault prediction is given in [9]. In this work, a deep neural network model for software fault prediction is built and also several aspects of the deep neural network design are explored. The role of number of layers, nodes in each layer, learning rate,

loss function, optimizer, and regularization methods has been studied.

The organization of the rest of the paper is as follows: Section 2 presents the related work, and Section 3 gives the theoretical background. Section 4 presents the experimental setup, and Section 5 gives the results and analysis. Finally, Section 6 concludes the paper.

2. Related Work

This section presents the literature review of research papers on the use of machine learning techniques for software fault prediction.

Singh et al. [10] used public dataset AR1 for predicting fault proneness of modules. They compared logistic regression technique with 6 machine learning classifiers (decision tree (DT), group method of data handling polynomial method, artificial neural network (ANN), gene expression programming, support vector machine (SVM), and cascade correlation network). The performance was compared by computing the area under the curve using Receiver Operating Characteristic (ROC) analysis where it was concluded that the value generated by the decision tree was 0.865 which outperformed regression and other ML techniques. Dejaeger et al. [11] considered 15 distinct Bayesian network (BN) classifiers, and comparison was performed with machine learning (ML) techniques. For the purpose of feature selection, Markov blanket principle was used. The area under the ROC curve (AUC) and H-measure was tested using the statistical framework of Demšar. The result showed that simple and comprehensible networks having less number of nodes can be constructed using BN classifiers other than the naive Bayes classifier.

Cahill et al. [12] presented an approach for finding fault proneness in modules where the rank sum representation allowed the user to opt a suitable trade-off between recall and precision. This approach was executed using NASA Metrics Data Program (MDP) datasets, and their performance was compared with classifiers such as the support vector machine (SVM) and naïve Bayes (NB). Arar and Ayan [13], in their study, built a software defect prediction model using artificial neural network (ANN) technique. They optimized ANN connection weights by artificial bee colony (ABC). Through the new error function, the parametric cost-sensitivity feature was added. This model was validated using 5 NASA repository datasets. Results were than compared with noncost-sensitive and cost-sensitive readings.

He et al. [6] experimented on 34 releases of 10 projects of PROMISE repository using 6 classifiers with 3 types of predictors. Their findings showed that predictors built using top-k metrics or the minimum metric subset deliver satisfactory result as compared to benchmark predictors. Also basic classifiers such as naïve Bayes (NB) execute well when the simplified metric set is used for fault prediction. Kumar et al. [14] experimented on 30 open-source projects to build a ML-based model for the software fault prediction model using the least square support vector machine (LSSVM). They applied 10 distinct feature selection techniques. Their prediction model was only appropriate for projects with faulty classes less than the threshold value. Twala [15] performed software fault prediction on 4 NASA public datasets using decision tree (DT), support vector machine (SVM), K-nearest neighbor, and naïve Bayes. He concluded that the naïve Bayes classifier was most robust and decision tree classifier the most accurate. Boucher and Badri [16] investigated 3 thresholds' calculation techniques Alves rankings, VARL (Value of an Acceptable Risk Level), and ROC curves for prediction of fault proneness. Then, the generated results were compared with the performance of 2 clustering-based models and 4 ML models. They used 12 public datasets, where these datasets belonged to the PROMISE Repository and Eclipse project. Results depicted that models using ROC curves outgrow both ML and clustering-based models.

Wang et al. [17] proposed a representation learning algorithm using the deep belief network (DBN) which helps in learning semantic program representation directly from source code. They worked on 10 open-source projects and showed that directly learned semantic features considerably improve both within and cross-project defect prediction (WPDP) (CPDP). On an average, WPDP was improved by 14.2% in F1, 11.5% in recall, and 14.7% in precision. And, the CPDP approach beats TCA+ having traditional features by 8.9% in F1. Erturk and Akcapinar Sezer [18] proposed a novel software fault prediction methodology, which was based on fuzzy inference system (FIS) and artificial neural network (ANN). The methodology was developed as Eclipse plugin. Their investigation demonstrated that the hybrid approach used in the proposed methodology gave favorable results to use SFP in everyday routine of software development phases. Miholca et al. [19] proposed HyGRAR, a non-linear hybrid supervised classification method for software fault prediction. HyGRAR combined relational association rule mining and artificial neural networks (ANN) to distinguish between faulty and nonfaulty software objects. For experimental purpose, they used 10 open-source datasets and validated the outstanding performance of the HYGRAR classifier.

Samir et al. [20] built a software defect prediction model using deep neural network technique and compared its performance with ML techniques (random forests (RF), decision trees (DT), and naive Bayesian networks (NB)). Results showed that deep neural network technique outperformed ML techniques in most of the cases. For the experimental purpose, they used NASA datasets and datasets from TERA-PROMISE repository. Turabieh et al. [21] focused in developing an effective defect prediction classifier using L-RNN, an iterated feature selection algorithm. They experimented on 19 open-source datasets and found out that defect prediction models are best fit for modules with faulty classes having lesser values than the threshold value. Li et al. [22] proposed a framework called Defect Prediction via Convolutional Neural Network (DP-CNN) that used deep learning in order to effectively generate features. On the bases of program's Abstract Syntax Trees (ASTs), they initially extracted token vectors and then encoded them as numerical vectors with the help of the process of word embedding and word mapping. Then, these numerical

vectors were fed into the convolutional neural network that automatically learnt structural and semantic program features. Then after, for perfect software fault prediction, they combined traditional hand-crafted features with the learnt features. The experiment was conducted on 7 open-source project data. The measurement was done on the bases of *F*-measure. The final results showed that, DP-CNN improves the state-of-the-art method by 12%.

Yucalar et al. [23], in their study, aimed at empirical demonstration of performance of fault prediction of 10 ensemble predictors with baseline predictor. The experiment was conducted on 15 open-source project datasets from PROMISE repository. The performance was tested on the bases of Area under the Receiver Operating Characteristics (ROC) Curve (AUC) and *F*-measure. They concluded that ensemble predictors may improve performance of fault detection to some degree. Duddu et al. [24], in their work, considered the trade-off between adversarial robustness, fault tolerance, and privacy. Two adversarial settings were also considered under the security and privacy threat model. They studied the effect of training the model with gradient noise (differential privacy) and input noise (adversarial robustness) on neural network's fault tolerance. It was observed that due to increased overfitting, the adversarial robustness drops fault tolerance and also $(\in \mathrm{d}p, \delta\mathrm{d}p)$-differentially private models boost the fault tolerance.

Lyu and Jiang [25] established a method by using a combination of the artificial neural network and gray neural network with fuzzy recognition to understand the fault prediction of the avionics system. In this method, they first created a network model using a combination of the artificial neural network and gray neural network with fuzzy recognition, and experimental analysis was conducted. Then, the weight update strategy of the gray neural network was improved by using the additional learning rate (LR) method. This improved combination improved prediction accuracy and time series prediction which is an effective technical method for avionics system fault prediction.

In practice, software defect prediction models often suffer from highly imbalanced data, which makes classifiers difficult to identify defective instances. Recently, many techniques were proposed to tackle this problem; oversampling technique is one of the most well-known methods to address the class imbalance problem. This technique balances the number of defective and nondefective instances by generating new defective instances. However, these approaches would generate nondiverse synthetic instances and many unnecessary noise instances at the same time. Motived by this, Gong et al. [26] proposed a cluster-based oversampling with noise filtering (KMFOS) approach to tackle the class imbalance problem in software defect prediction. KMFOS first divides defective instances into K clusters, and new defective instances are generated by interpolation between instances of every pair of two clusters. Experimental results indicate that the KMFOS can obtain better Recall and bal values than other oversampling methods and other compared class-imbalance methods. Hence, KMFOS is an efficient approach to generate balanced data for software defect prediction and improve the performance of predicting models.

Huda et al. [27] proposed two hybrid SDP models by using wrapper and filter techniques. The wrapper approach included ANN and SVM and a maximum filter approach which helped in finding significant metrics. The experiment showed that the hybrid approach produced high prediction accuracy as compared to the traditional filter or wrapper approach. Proposed framework's performance was validated using a statistical multivariate quality control process using multivariate exponentially weighted moving average. Bishnu and Bhattacherjee [28] applied a quad tree-based K-means algorithm for defect prediction in program modules. It is a cluster-based technique. Initially, cluster centers were found out using quad tree, which became input to the K-mean algorithm. Clustering gain was used to determine the quality of generated clusters for evaluation. The clusters generated by the quad tree-based algorithm had maximum gain values. Then, this quad tree-based algorithm was applied for defect prediction in modules. The error rate of this algorithm was compared to other algorithms, and it was observed to perform better in most of the cases.

Pandey et al. [29] proposed a rudimentary classification-based framework Bug Prediction using Deep representation and Ensemble learning (BPDET) techniques for the software bug prediction (SBP) model. Staked de-noising auto-encoder (SDA) was used for the deep representation of software metrics. Their proposed model was divided into deep learning stage and two layers of EL stage (TEL). The experiment was performed on NASA (12) datasets, to calculate the efficiency of deep representation (DR), SDA, and TEL. The performance was evaluated in terms of Mathew correlation coefficient (MCC), the area under the curve (AUC), precision-recall area (PRC), *F*-measure, and Time. BPDET was tested using the Wilcoxon rank sum test which rejects the null hypothesis at $\alpha = 0.025$. They also tested the stability of the model over 5-, 8-, 10-, 12-, and 15-fold cross-validation and got similar results. Finally, conclusion was that BPDET is stable and outperformed on most of the datasets compared with EL and other state-of-the-art techniques.

Lei et al. [30] reviewed applications of machine learning to machine fault diagnosis, which they divided into 3 periods. They also pictured and systematically presented the development of intelligent fault diagnosis (IFD) to show potential research trends. Also, challenges of IFD were also discussed. Zhang et al. [31] proposed a novel deep CNN method which was based on knowledge transferring from shallow models for rotating machinery fault diagnosis with scarce labeled samples. In their work, they first applied short-time Fourier transform (STFT) to extract integral features. Then, they trained the SVM model with scarce labeled samples and made predictions on unlabelled samples which were in turn used to train a deep CNN model of better discriminative ability. Experimental results demonstrated the effectiveness of their proposed method over the SVM model and original deep CNN model trained with only scarce labeled samples.

Bashiri and Farshbaf Geranmayeh [32] studied 3 ANN performance measuring criteria and 3 factors which affect the selected criteria. To design experiments, the central composite design was used, and then, network behavior was

analysed according to identified parameters. Then, to find the optimal parameter status, a genetic algorithm was proposed. The results show that the designed ANN, according to the proposed procedure, had a better performance than other networks by random selected parameters and also parameters which are selected by the Taguchi method. In general, the proposed approach could be used for tuning neural network parameters in solving other problems. Lee et al. [33] proposed a method to improve CNN performance by hyperparameter tuning in the feature extraction step of CNN. In their proposed method, the hyperparameter was adjusted using a parameter-setting-free harmony search (PSF-HS) algorithm. In the PSF-HS algorithm, the hyperparameter that was to be adjusted was set as harmony, and harmony memory was generated after generating the harmony. Harmony memory got updated based on the loss of a CNN. Two simulations using CNN architecture on the LeNet-5 and MNIST and CifarNet and Cifar-10 dataset were performed. It was observed that, by two simulations, it was possible to improve the performance by tuning the hyperparameters in CNN architectures.

Yang and Shami [34] studied the optimization of the hyperparameters of common machine learning models. They introduced, discussed, and applied several state-of-the art optimization techniques. Experiments were applied on benchmark datasets so as to see the clear comparison of performance between different optimization methods. Out of all the hyperparameter optimization (HPO), they summarized Bayesian Optimization HyperBand (BOHB) as the recommended choice for optimizing a ML model; Bayesian optimization (BO) models were given preference for small hyperparameter configuration space, while particle swarm optimization (PSO) was the best choice for large configuration space. Cho et al. [35], for DNN hyperparameter optimization, analysed 4 basic strategies for enhancing Bayesian Optimization (BO). Investigation for diversification, early termination, parallelization, and cost function transformation was carried out. An algorithm named DEEP-BO (Diversified, Early-termination Enabled, and Parallel Bayesian Optimization) was proposed by the authors. Experiments were conducted on six DNN benchmarks. Their proposed algorithm out performed well-known solutions including GP-Hedge and BOHB. In general, DEEP-BO exhibited a robust performance, and it also displayed high performance particularly for the challenging targets under the use of multiple processors. Moolayil [36] discussed L1, L2, dropout regularization, and hyperparameter tuning which included discussion about the number of neurons in a layer, number of layers, number of epochs, weight initialization, batch size, learning rate, activation function, and optimization. They also discovered different strategies one could use to tune the hyperparameters and obtain a better quality model. Also a few principles were addressed which are needed, while deploying a model. At the end, they also looked into a small architecture for deploying the model using Flask.

Akl et al. [37], in their work, studied the effect of altering a hyperparameter within the deep learning model architecture. An architectural position optimization (ArchPosOpt) method was proposed for model architectural hyperparameter optimization. This architecture extended three different hyperparameter optimization techniques, namely, grid search (GS), random search (RS), and Tree-structured Parzen Estimator (TPE), so as to gain a new aspect of the hyperparameter optimization problem—the hyperparameter position. With the help of a set of experiments (experiments of image classification for two datasets; binary classification and multiclass classification), they showed that the position of the hyperparameters does matter for both model performance as well as the hyperparameter values. The ArchPosOpt method was found to have higher accuracy as compared to original tools. Bal and Kumar [38] explored an effective machine learning technique, i.e., extreme learning machine (ELM) for estimation of the number of software faults. And, also a new variation of ELM was proposed, named weighted regularization ELM (WR-ELM). It generalized the imbalanced data to balanced data. The proposed model was validated through the use of 26 open-source PROMISE software fault datasets. The use of three prediction scenarios named intrarelease, interrelease, and cross project was done for experimentation. The proposed WR-ELM model was able to characterize minority (faulty) modules and performed better as compared to other traditional ML algorithms. It was also able to handle the imbalanced software defect data by including the information of imbalanced class distribution.

Manjula [39] presented an approach for software fault prediction. In this approach, the genetic algorithm optimization process for feature subspace reduction was linked with the deep belief network for pattern learning. Then, the deep belief networks were further enhanced by applying the $L1$-regularization scheme which resulted in better learning process which reduced the overfitting errors. This linked model was executed on the SPIE lab software defect database. A broad experimental study was carried out which showed that the proposed approach achieved higher accuracy when compared with other state-of-the-art software fault prediction techniques. Qu et al. [40] conducted an in-depth analysis to check the impact on the performance of cross-project defect prediction (CPDP) by using hyperparameter optimization. Based on diverse classification methods, they selected 5 different instance selection-based CPDP methods. For empirical studies, 8 projects in AEEEM and Relink datasets were chosen. AUC was used as a model performance measure. The results showed that the impact of hyperparameter optimization for 4 methods is non-negligible, and among the 11 hyperparameters considered by these 5 classification methods (K-nearest neighbor (IBK), J48, NB, RF, and SVM), the impact of 8 hyperparameters is nonnegligible, and these hyperparameters are mostly dispersed in SVM and IBK classification methods. Kudjo et al. [41] presented an approach to characterize and predict vulnerable software components grounded on a concept take from the field of fault prediction. Their study inspects the degree to which parameter optimization affects the performance of vulnerability prediction models. The evaluation of the approach was conducted by applying it on three open-source vulnerability datasets i.e., Drupal, Moodle, and

PHPMyAdmin using five ML algorithms, namely, random forest (RF), *K*-nearest neighbor, support vector machine (SVM), J48 decision tree, and multilayer perceptron. The effect of parameter tuning on vulnerability prediction models (VPMs) was also examined. The finding showed a significant increase in precision and accuracy against the benchmark study.

2.1. Theoretical Background. This section discusses about brief overview of a generalized software fault prediction process, deep neural networks, parameter tuning process, *L2* regularization, and dropout regularization.

2.1.1. A Generalized Software Fault Prediction Process Based on Machine Learning. For the process of software fault prediction, the data that is faulty should be collected for training a prediction model. Figure 1 explains the following process.

(1) Firstly, extract instances (data items) from software repository/archives.

(2) Then, feature extraction takes place which mean extracting required metrics from instances.

(3) Then, preprocessing is applied on metrics as the real world data is in raw format, and it cannot be passed through a model directly.

(4) Now the processed data is split into training and testing instances. Usually, to separate the training and testing instances, 10-fold cross-validation is used.

(5) From the training instance, the prediction model is built.

(6) The model built obtains a new instance and can also classify labels, i.e., faulty (defect) or nonfaulty (no defect).

2.1.2. Brief Overview of Deep Neural Network. A DNN is a series of fully connected hidden layers which transform an input vector x into a probability distribution to estimate the output class y [42]. The DNN thus acts as a mapping for the distribution $p\ (y|x)$. A DNN maps this function using l hidden layers followed by an output layer. The nodes in each layer are connected to all the nodes in the subsequent layer with weighted edges. DNN architecture is shown in Figure 2. These weights can be thought of as a weight matrix W. Each layer also has a bias vector b. Compute vector h^i of the i^{th} layer using the activations of the previous layer of the DNN as

$$h^{(i)}(x) = \sigma\left(W^{(i)}h^{(i-1)} + b^{(i)}\right). \tag{1}$$

In all hidden layers, a nonlinear function as part of the hidden layer computation is applied. This activation function is attached to each neuron in the neural network. The activation function normalizes each neuron's output to a range between −1 and 1 or between 1 and 0. In the most of previous works, typically a sigmoidal function would be used as the activation function. However, in our work, rectified linear units are used which were recently shown to give a better performance in many DNN classification tasks.

Here, the Rectified Linear Unit (ReLU) function is used because it looks like a linear function, but is indeed a nonlinear function which allows complex relationships in the data to be learned. For the input values that are negative, the neurons stay deactivated and result is 0, and for positive inputs, the output is equal to the input. Figure 3 displays the ReLU activation function graph.

The mathematical expression for Rectified Linear Unit (ReLU) activation function is

$$R(x) = \max(0, x). \tag{2}$$

To produce values from the output layer, the Softmax activation function is used which is also a type of sigmoid function. Softmax normalizes each neuron's output to a range of 1 and 0. It is nonlinear in nature. It is usually used when trying to handle multiple classes.

The mathematical expression for the Softmax activation function is

$$\sigma(z)_i = \frac{e^{zi}}{\sum_{j=1}^{K} e^{zj}}. \tag{3}$$

Also, the Adam optimizer, as an optimization function, is used in order to update the weight of the network after every single iteration.

2.1.3. Parameter Tuning Process. The machine learning models largely work in an empirical manner, with the researcher tuning her models as per the application domain and the data available. However, in this research work, the major focus is on tuning the parameters such as number of hidden layers and number of nodes in each layer and working on training details such as learning rate and regularization methods. These shall be discussed briefly now.

For any dataset, the training starts with small number of hidden layers and small number of nodes in each layer. If train accuracy is not good, more layers and nodes are added. The number of epochs is also increased. This strategy of bigger network and longer training continues until the train data fits fairly well or at least up to the accuracy obtained by other classifiers. After this, the validation set performance is checked. If the performance is not good, this is because there is a high variance problem, and the network has overfitted the training data, but unable to generalize. To overcome this, regularizing of the network is considered.

(1) *L2 Regularization*. A regularization parameter λ is set which is used as in the loss (or cost) function J as follows:

$$J(W, b) = \left[\frac{1}{m}\sum_{n=1}^{m}\left(y_n\ \log\widehat{y}_n + (1 - y_n)\ \log\left(1 - \widehat{y}_n\right)\right)\right] + \frac{\lambda}{2m}\left\|W\right\|_{2^2}. \tag{4}$$

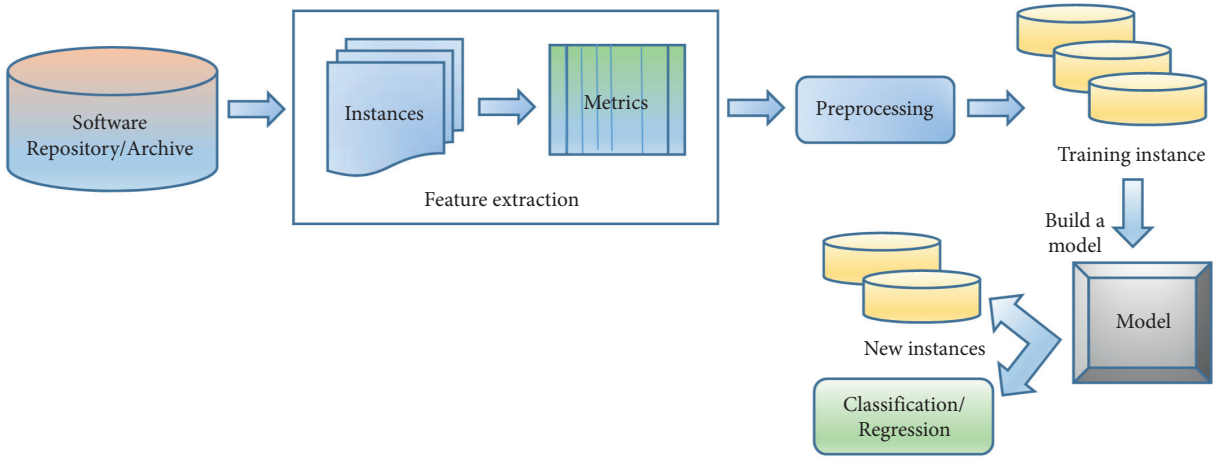

Figure 1: A generalized software fault prediction process based on machine learning.

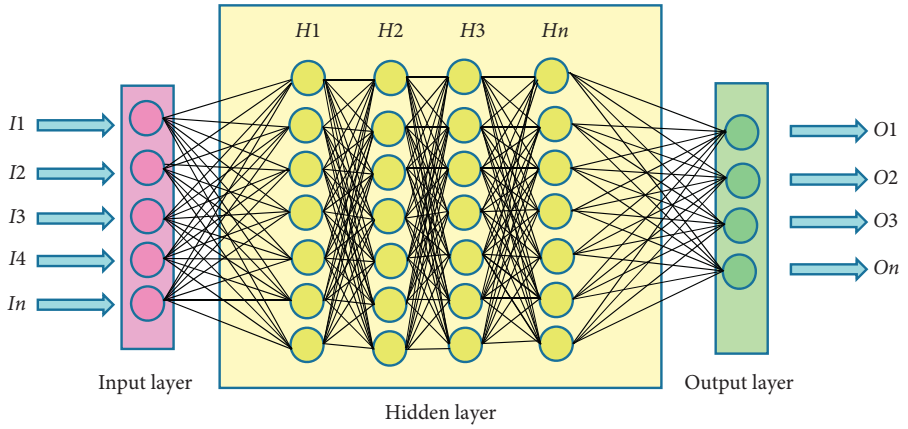

Figure 2: Deep neural network.

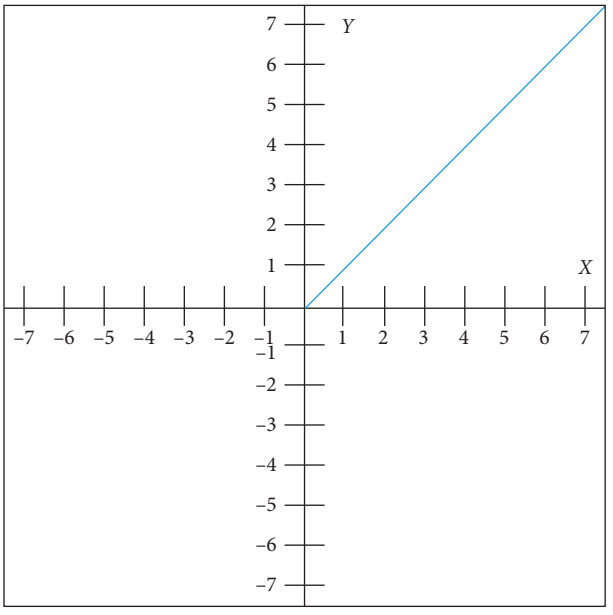

Figure 3: ReLU activation function.

The first term on the right-hand side of equation (4), cross-entropy loss function, evaluates the performance of a classification model whose output is a probability value between 0 and 1. In this, y_n is an actual value and $\widehat{y}_n$ is a predicted value. The second term on the right-hand side of equation (4) is the $L2$ regularization term, which has the squared norm (also called the Frobenius norm) of the weight matrix. Here, "m" is the number of samples in the dataset. To minimize the loss function J, it is required that both the terms on the right-hand side be minimized. By setting a high value of λ, the weights are forced to become smaller (to minimize J). A network with smaller weights is simple and cannot learn complex functions. By penalizing square values of the weights in the cost function, all weights are driven to smaller values since the cost would be high with higher weights. In effect, what happens is some neurons become dormant or left out of the model, making it a simple one. The $L2$ regularization is also sometimes called weight decay regularization. In the experiments, the λ values used are between 0.05 and 0.7.

(2) *Dropout Regularization*. Dropout is a widely used regularization technique that is specific to deep learning. It randomly shuts down some neurons in each iteration. It simply means randomly selected neurons are "dropped out" randomly. When some neurons are shut down, in every iteration, we are actually training a different model that uses a subset of neurons. Thus, the neurons in the model learn features independently without being specifically dependent on other neurons. This means those dropped-out neurons are temporally removed on the forward pass and no weight updation will applied to them on the backward pass. Regularization hurts training set performance because it limits the ability of the network to overfit to the training set. Usually, the place of dropout is in the fully connected layers as it is the one with the larger number of parameters and thus more probable to excessively co-adapt themselves causing overfitting. A DNN with some dropout nodes is shown in Figure 4.

Consider a particular node x in layer l and nodes u_1, u_2, u_3, and u_4 in layer $l-1$ connected to x. What dropout actually does is to spread out the weights. Instead of assigning weight to any one node, it spreads out among all the nodes. The following illustration will demonstrate this. Let the weights of the connections between node x and u_1, u_2, u_3, and u_4 be w_{1x}, w_{2x}, w_{3x}, and w_{4x}.

Squared norm $\|w\|^2$ for this layer is

$$\|w\|^2 = w_{1x}^2 + w_{2x}^2 + w_{3x}^2 + w_{4x}^2. \tag{5}$$

Let the sum of the weights be equal to k, i.e., $\sum w_{ix} = k$.

Case 1. When the entire weight is with one connection, u_1 to x,

$$\begin{aligned} w_{1x} &= k, \\ w_{2x} &= w_{3x} = w_{4x} = 0, \\ \|w\|^2 &= k^2. \end{aligned} \tag{6}$$

Case 2. When the weight is equally distributed among two connections, u_1 to x and u_2 to x,

$$\begin{aligned} w_{1x} &= \frac{k}{2}; \\ w_{2x} &= \frac{k}{2}; \\ w_{3x} &= 0; \\ w_{4x} &= 0, \\ \|w\|^2 &= \frac{k^2}{4} + \frac{k^2}{4} = \frac{k^2}{2}. \end{aligned} \tag{7}$$

Case 3. When the weight is equally distributed among all four connections,

$$\begin{aligned} w_{1x} = w_{2x} = w_{3x} = w_{4x} &= \frac{k}{4}, \\ \|w\|^2 &= \frac{k^2}{4}. \end{aligned} \tag{8}$$

In each of the cases, the squared norm of weights decreases when the weights are distributed, as in Cases 2 and 3, rather than when it is concentrated with one connection as in Case 1. To choose the value of dropout probability, for layers with large number of nodes, dropout should be high, and for those with small nodes, dropout should be low, maybe 0.

So it can be summarized by saying that the $L2$ regularization method reduces overfitting by modifying the cost function. But on the contrary, the dropout method reduces overfitting by modifying the network itself.

3. Experimental Setup

This section details about datasets, experimental environment, environment deployment, and evaluation parameters.

3.1. Datasets. There are a number of open-source datasets available online for the analysis of defect prediction models. For the study, 4 NASA system datasets (KC1, PC1, PC2, and KC3) are selected from PROMISE repository [43], which is freely available as public datasets. The selected datasets are of

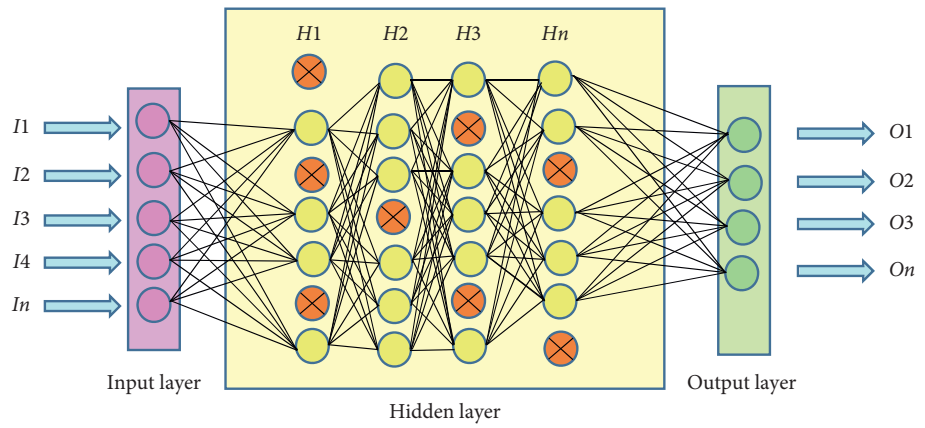

Figure 4: Deep neural network with some dropout nodes.

different sizes and different number of set of metrics i.e., KC1 has 22 attributes with 2109 instances, PC1 has 22 attributes with 1109 instances, PC2 has 37 attributes with 745 instances, and KC3 has 40 attributes with 194 instances. These datasets comprise of software metrics such as Halstead and McCabe metrics and a Boolean variable that indicates defect or no-defect proneness of a module. Table 1 displays characteristics of the NASA dataset (PC1, PC2, KC1, and KC3).

The WEKA (Waikato Environment for Knowledge Analysis) tool was used for the statistical output processing of datasets. WEKA is open-source software that gives the user the power of preprocessing, implementation of well-known machine learning algorithms, and visualization of their data so that one can develop machine learning techniques and apply them to real-world data problems. The data was analysed i.e., the accuracy of different datasets was calculated using various classifiers, namely, random forest, decision tree, and naïve Bayes. The results of these classifiers were then compared with the results generated by the neural network.

3.2. Experimental Environment. For building the deep neural network (DNN), the network parameters such as the total number of hidden layers and the number of neurons in each corresponding layer were configured. Four datasets were selected to conduct the experiment with different configuration settings i.e., by varying the number of hidden layers, number of neurons in each corresponding layer, epochs, learning rate, and with and without dropout.

The network setting for the datasets that gave us desired results while experimenting is as follows: PC1 had 5 hidden layers with 5, 5, 5, 10, and 20 neurons in each layer, respectively, and *L2* regularization with the value 0.7 in the last layer. KC1 had 6 hidden layers with 20, 20, 20, 20, 50, and 50 neurons in each layer, respectively, and *L2* regularization with the value 0.05 in the last layer. KC3 had 4 hidden layers with 5, 5, 10, and 10 neurons in each layer, respectively, and *L2* regularization with the value 0.2 in the last layer. PC2 had 5 hidden layers with 80, 80, 80, 80, and 200 neurons in each layer, respectively, and *L2* regularization with the value 0.7 in the last two layers. An overview of our proposed research framework and the pseudocode is shown in Figures 5 and 6.

3.3. Environment Deployment. For the proposed DNN model's modelling, Python 3.7.3 is used. With the help of Keras, which is a neural network library written in Python and which is also capable of running on top of TensorFlow, the DNN-related results were generated. The experiments were executed using the system having 64 bit operating system with 16 GB RAM.

3.4. Evaluation Parameters. In the field of machine learning and, specifically, the problem of statistical classification, a confusion matrix, also known as an error matrix, is used. A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions is summarized with count values and broken down by each class. This is the key to the confusion matrix. The confusion matrix shows the ways in which the classification model is confused when it makes predictions. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made. Figure 7 shows the description regarding the confusion matrix.

Class 1: NO

Class 2: YES

The above terms are defined as

(1) **YES**: observation is positive

(2) **NO**: observation is not positive

(3) **TruePositive (TP)**: observation is positive and is predicted to be positive

(4) **FalseNegative (FN)**: observation is positive, but is predicted negative

(5) **TrueNegative (TN)**: observation is negative and is predicted to be negative

(6) **FalsePositive (FP)**: observation is negative but is predicted positive

Table 1: Characteristics of the NASA dataset (PC1, PC2, KC1, and KC3).

Dataset	Project	Number of attributes	Number of instances	Number of defective entities	Number of nondefective entities
NASA	PC1	22	1109	77 (6.9%)	1032 (93.05%)
NASA	PC2	37	745	16 (2.10%)	729 (97.90%)
NASA	KC1	22	2109	326 (15.45%)	1783 (84.54%)
NASA	KC3	40	194	36 (18.6%)	158 (81.4%)

```
Select the number of layer and parameters
Perform validation
Repeat
    Select the number of layer (1-M: M depends on experiments)
        Select number of nodes of the layer
        Choose dropout [Range 0.1-0.7]
        Select parameter
        Perform training
        Perform validation
    Replace to next layer
Until reached desired threshold
```

Figure 5: Algorithm pseudocode.

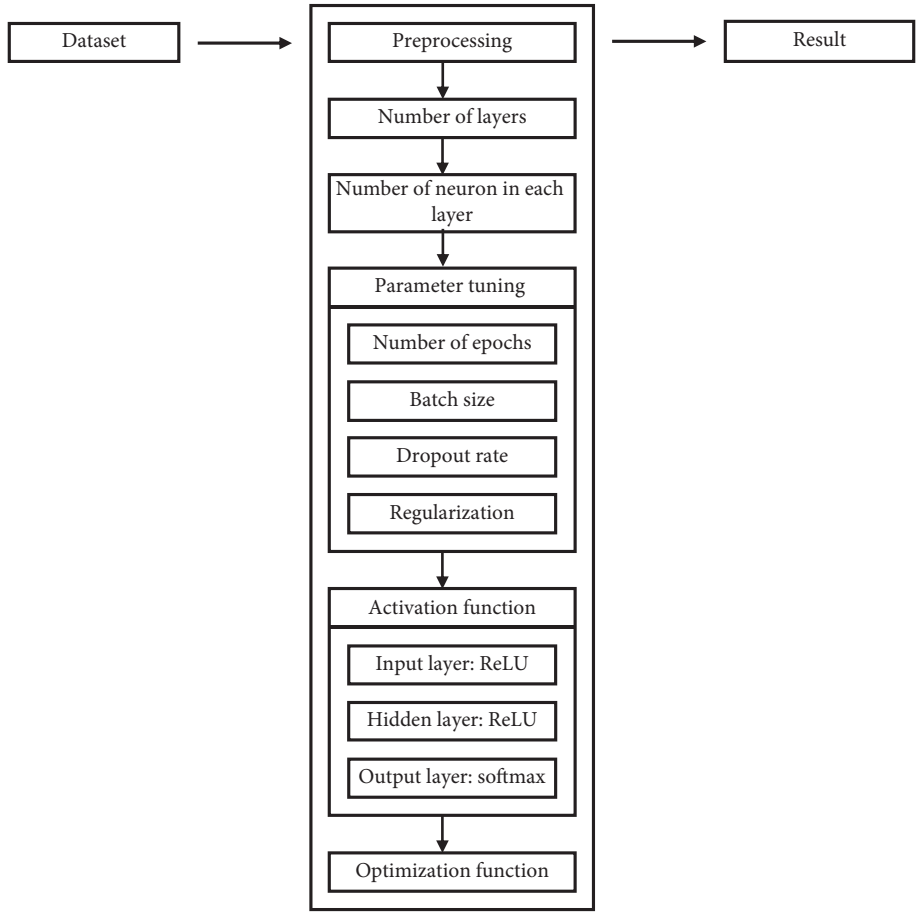

Figure 6: Proposed research framework.

		Predicted	
		NO	YES
Actual	NO	TN	FP
	YES	FN	TP

Figure 7: Confusion matrix.

The evaluation parameters used in this research work are True Positive Rate (TPR), True Negative Rate (TNR), False Negative Rate (FNR), False Positive Rate (FPR), precision, recall, *F*-measure, and accuracy.

True Positive Rate is when it is actually YES; how often does it predict YES?

$$\text{TRP} = \frac{\text{True Positive (TP)}}{\text{True Positive (TP)} + \text{False Negatives (FN)}}. \quad (9)$$

True Negative Rate is when it is actually NO; how often does it predict NO?

$$\text{TNR} = \frac{\text{True Negatives (TN)}}{\text{True Negatives (TN)} + \text{False Positives (FP)}}. \quad (10)$$

False Positive Rate is when it is actually NO; how often does it predict YES?

$$\text{FPR} = \frac{\text{False Positives (FP)}}{\text{False Positives (FP)} + \text{True Negatives (TN)}}. \quad (11)$$

False Negative Rate is the proportion of YES which yields NO test outcomes with the test:

$$\text{FNR} = \frac{\text{False Negatives (FN)}}{\text{False Negatives (FN)} + \text{True Positive (TP)}}. \quad (12)$$

Precision (P) measures the number of positive class predictions that belong to the positive class:

$$(P) = \frac{\text{True Positive (TP)}}{\text{True Positive (TP)} + \text{False Positives (FP)}}. \quad (13)$$

Recall measures the number of positive class predictions made out of all positive examples in the dataset:

$$(R) = \frac{\text{True Positive (TP)}}{\text{True Positive (TP)} + \text{False Negatives (FN)}}. \quad (14)$$

F-measure offers a single score that balances both the concerns of precision and recall in one number:

$$(\text{FM}) = 2^{*}\frac{\text{Recall } (R) * \text{Precision } (P)}{\text{Recall } (R) + \text{Precision } (P)}, \quad (15)$$

whereas accuracy is the total number of correct predictions divided by the total number of predictions made for a dataset:

$$(A) = \frac{\text{True Positive (TP)} + \text{True Negatives (TN)}}{\text{Total Examples (TE)}}. \quad (16)$$

4. Results and Analysis

For the final analysis, the performance measures for all the 4 classification techniques used in the study are computed. The results were based on the values of precision, recall, *F*-measure, and accuracy. Also, accuracy comparison with different dropout rates is also discussed in the later part of this section.

Table 2 presents each classifier's comparative result in terms of precision, recall, *F*-measure, and accuracy for the KC1 dataset. For this dataset, the deep neural network with dropout provides the best result for recall and accuracy i.e., 1 and 92, respectively, whereas the deep neural network without dropout provides the best result in terms of the *F*-measure value. Thus, DNN with dropout outperforms all other classifiers in terms of accuracy. Table 3 shows each classifier's performance statistics for the KC3 dataset. For this dataset, our DNN model with dropout outperforms all other classifiers. The precision, recall, and *F*-measure value is examined to be 0.91, 1, and 0.98, respectively, and the accuracy value is calculated as 97. In Table 4, it is observed that the accuracy value (96) and recall value (1) of DNN with dropout are better than all other classifiers but the value of precision (0.95) is good for the random forest classifier, and *F*-measure (0.97) is good for DNN without dropout. For Table 5, the proposed DNN model with the dropout precision value (0.98), recall value (1), *F*-measure (0.98), and accuracy value (99) is better than the performance values generated from DNN without dropout, RF, NB, and DT.

Figures 8–11 display the graphs showing performance comparison between RF, DT, NB, and DNN with/without dropout for all datasets. Figure 12 displays different accuracies generated by the 4 different classifiers for KC1, KC3, PC1, and PC2 datasets. It is observed that, in each case, the accuracy generated by the proposed deep neural network (DNN) with dropout is the highest as compared to other machine learning techniques.

As it is known, dropout is a method by which model overfitting is prevented. In this method, outgoing edges of hidden neurons are randomly set to zero at each update of the training phase. Here, dropout rates have been taken in between 0.1 and 0.7 for all datasets and are intensively experimented and explored changes in the accuracy. At first, with the increasing dropout rate, loss will decrease and accuracy will gradually increase. But if the dropout is incremented beyond a certain threshold, it results in decrease in accuracy, and hence, the model is not being able to fit properly. It is observed from Table 6 that, with increasing dropout rates from 0.1 to 0.5, accuracy is also increasing. But

Table 2: Comparative results for the KC1 dataset.

KC1				
Algorithm	Precision	Recall	*F*-measure	Accuracy
RF	0.887	0.965	0.925	86.670
DT	0.865	0.974	0.916	84.870
NB	0.888	0.905	0.897	82.360
Without dropout DNN	0.790	0.970	0.940	88.570
With dropout DNN	0.850	1.000	0.920	**92.000**

Table 3: Comparative results for the KC3 dataset.

KC3				
Algorithm	Precision	Recall	*F*-measure	Accuracy
RF	0.832	0.968	0.895	81.440
DT	0.870	0.930	0.899	82.990
NB	0.863	0.880	0.971	78.870
Without dropout DNN	0.890	0.870	0.880	93.000
With dropout DNN	0.910	1.000	0.980	**97.000**

Table 4: Comparative results for the PC1 dataset.

PC1				
Algorithm	Precision	Recall	*F*-measure	Accuracy
RF	0.950	0.984	0.960	93.680
DT	0.937	0.990	0.960	92.870
NB	0.947	0.936	0.942	89.170
Without dropout DNN	0.940	0.990	0.970	93.000
With dropout DNN	0.930	1.000	0.960	**96.000**

Table 5: Comparative results for the PC2 dataset.

PC2				
Algorithm	Precision	Recall	*F*-measure	Accuracy
RF	0.979	1.000	0.989	97.850
DT	0.979	1.000	0.989	97.850
NB	0.980	0.925	0.951	90.730
Without dropout DNN	0.810	0.900	0.970	98.660
With dropout DNN	0.980	1.000	0.980	**99.000**

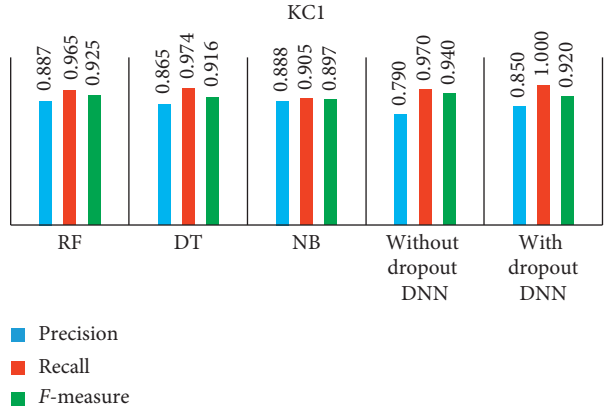

Figure 8: Performance comparison for KC1.

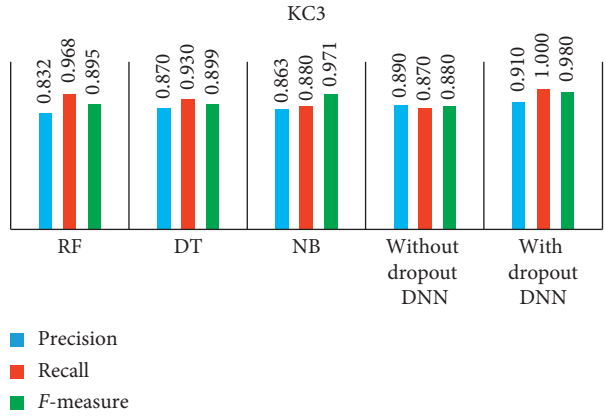

Figure 9: Performance comparison for KC3.

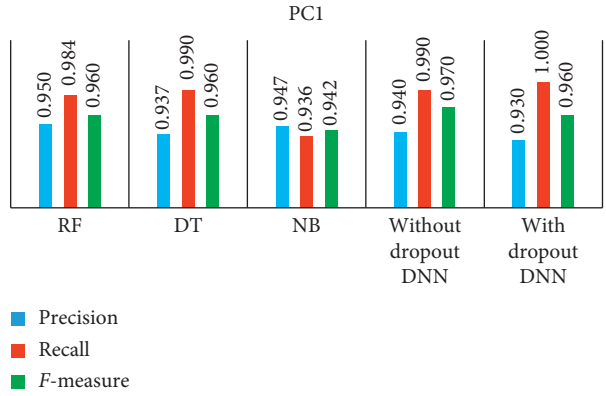

Figure 10: Performance comparison for PC1.

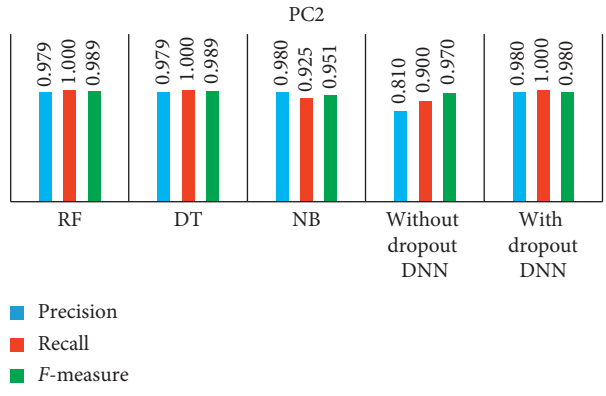

Figure 11: Performance comparison for PC2.

in most of the datasets, the accuracy starts to fall on increasing dropout beyond threshold, say 0.6 onwards.

From Table 7, it is observed that TPR of DNN with dropout is greater than DNN without dropout, RF, DT, and NB for the datasets KC1, KC3, PC1, and PC2. And, in all the cases, accuracy of DNN with dropout is higher than all other analysis which also reflects that proposed DNN with the dropout model outperforms all other classifiers.

From Figure 13, it is observed that the proposed DNN model with dropout in case of KC1 detects 168 faults, whereas the DNN model without dropout, RF, DT, and NB detects 152, 107, 54, and 123 faults, respectively. From Figure 14, it is observed that the DNN model with dropout in case of KC3 detects 37 faults, whereas the DNN model without dropout, RF, DT, and NB detects 32, 5, 14, and 14 faults, respectively. From Figure 15, in case of PC1, it is

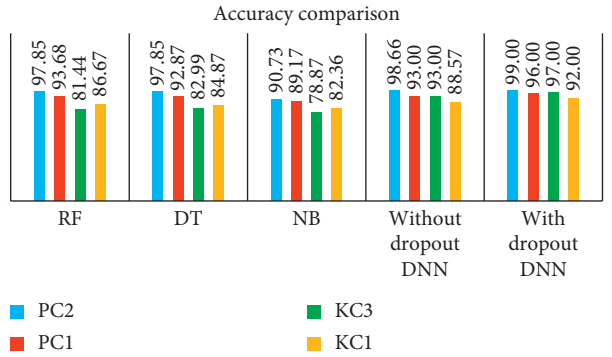

Figure 12: Accuracy comparison between KC1, KC3, PC1, and PC2 datasets.

Table 6: Accuracy comparison with different dropout rates.

Dropout rate	KC1	KC3	PC1	PC2
DR (0.1)	90.89	96.00	89.00	97.10
DR (0.2)	90.00	96.00	93.34	98.19
DR (0.3)	91.60	96.21	92.00	98.19
DR (0.4)	92.00	97.00	95.22	99.00
DR (0.5)	92.00	97.00	96.00	99.00
DR (0.6)	89.66	97.00	94.45	98.11
DR (0.7)	70.23	92.90	94.45	90.00

Table 7: Confusion matrix analysis for the KC1, KC3, PC1, and PC2 datasets (TPR: True Positive Rate, TNR: True Negative Rate, FPR: False Positive Rate, and FNR: False Negative Rate).

Algorithm	KC1				KC3				PC1				PC2			
	TPR	TNR	FPR	FNR	TPR	TNR	FPR	FNR	TPR	TNR	FPR	FNR	TPR	TNR	FPR	FNR
RF	0.330	0.960	0.040	0.670	0.140	0.970	0.030	0.860	0.290	0.980	0.015	0.700	0.000	1.000	0.000	1.000
DT	0.170	0.970	0.030	0.830	0.380	0.910	0.070	0.610	0.290	0.930	0.060	0.700	0.130	0.920	0.070	0.880
NB	0.380	0.900	0.070	0.620	0.380	0.900	0.120	0.610	0.290	0.930	0.060	0.700	0.130	0.920	0.070	0.880
Without dropout DNN	0.470	0.980	0.020	0.530	0.170	0.970	0.030	0.830	0.020	0.990	0.010	0.980	0.020	0.990	0.010	0.980
With dropout DNN	0.520	0.980	0.017	0.470	0.970	0.980	0.012	0.026	0.410	0.980	0.011	0.580	1.000	0.990	0.001	0.000

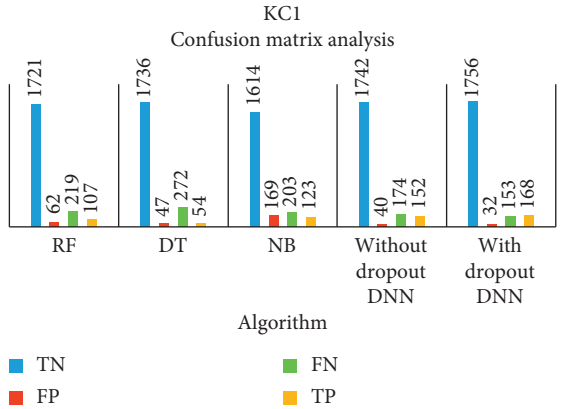

Figure 13: Confusion matrix analysis for KC1.

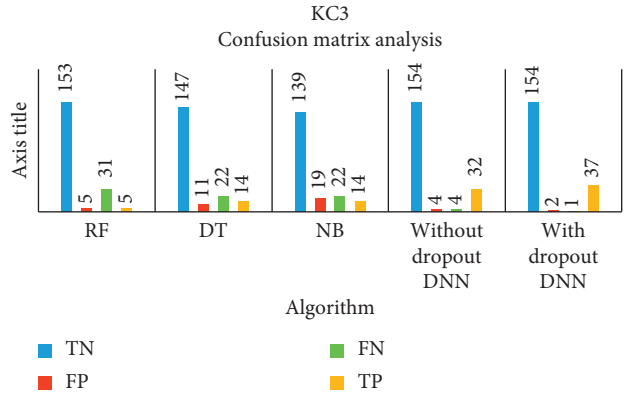

Figure 14: Confusion matrix analysis for KC3.

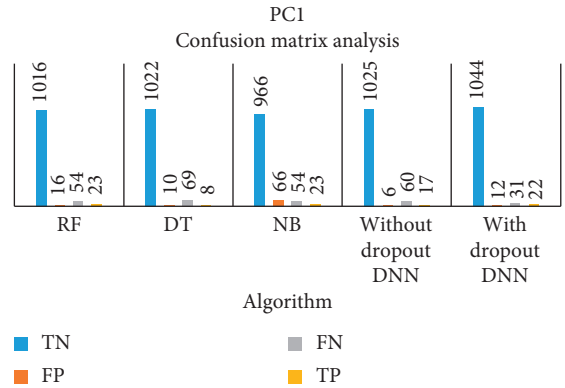

Figure 15: Confusion matrix analysis for PC1.

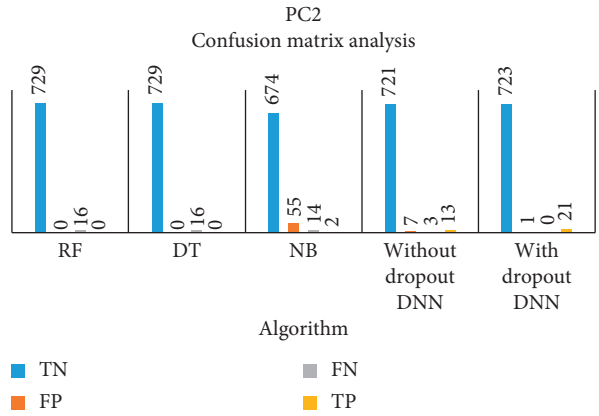

Figure 16: Confusion matrix analysis for PC2.

observed that RF and NB detect 23 faults, DNN model with dropout detects 22 faults, DNN model without dropout detects 17 faults, and DT detects only 8 faults. From Figure 16, it is observed that the DNN model with dropout in case of PC2 detects 21 faults, whereas the DNN model without dropout, RF, DT, and NB detects 13, 0, 0, and 2 faults, respectively.

Unlike model parameters, one cannot learn hyperparameters; they are needed to be tuned with different settings to get enhanced performance and desired results. So, the DNN was made robust by playing around with its width of the network (i.e., the number of neurons in the layer). It was also found that, by just adding up few layers, performance increased marginally. Sometimes, increase in the number of epochs displayed promising results.

For example, experiments with several parameter tunings were performed on all the four datasets. One set of experiment on the PC2 dataset is shown in Tables 8–11. In

Table 8: Experiment performed on the PC2 dataset by tuning the parameter i.e., learning rate = −2, epochs = 100, 200, 300, 500, 1000, and 2000, number of layers = 5, and dropout = 0.2 and 0.5.

No. of layers	5	5	5	5	5	5
No. of neurons with dropout	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80 (0.2), 80, 80, and 200	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)
Lr	$1.00E-02$	$1.00E-02$	$1.00E-02$	$1.00E-02$	$1.00E-02$	$1.00E-02$
Epoch	100	200	300	500	1000	2000
Accuracy	0.92	0.913	0.913	0.94	0.94	0.94
Loss	0.55	0.389	0.42	0.63	0.765	0.223

Table 9: Experiment performed on the PC2 dataset by tuning the parameter i.e., learning rate = -3, epochs = 100, 200, 300, 500, 1000, and 2000, number of layers = 5, and dropout = 0.2 and 0.5.

No. of layers	5	5	5	5	5	5
No. of neurons with dropout	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80 (0.2), 80, 80, and 200	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)
Lr	$1.00E-03$	$1.00E-03$	$1.00E-03$	$1.00E-03$	$1.00E-03$	$1.00E-03$
Epoch	100	200	300	**500**	1000	2000
Accuracy	0.9635	0.9783	0.9819	**0.99**	0.99	0.994
Loss	0.3606	0.3486	0.3611	**0.319**	0.4112	0.3529

Table 10: Experiment performed on the PC2 dataset by tuning the parameter i.e., learning rate = -4, epochs = 100, 200, 300, 500, 1000, and 2000, number of layers = 5, and dropout = 0.2 and 0.5.

No. of layers	5	5	5	5	5	5
No. of neurons with dropout	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80 (0.2), 80, 80, and 200	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)
Lr	$1.00E-04$	$1.00E-04$	$1.00E-04$	$1.00E-04$	$1.00E-04$	$1.00E-04$
Epoch	100	200	300	500	1000	2000
Accuracy	0.972	0.899	0.972	0.972	0.981	0.981
Loss	0.3614	0.3761	0.3649	0.3504	0.435	0.342

Table 11: Experiment performed on the PC2 dataset by tuning the parameter i.e., learning rate = -5, epochs = 100, 200, 300, 500, 1000, and 2000, number of layers = 5, and dropout = 0.2 and 0.5.

No. of layers	5	5	5	5	5	5
No. of neurons with dropout	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80, 80, 80, and 200	80 (0.2), 80 (0.2), 80, 80, and 200	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)	80 (0.5), 80 (0.5), 80 (0.5), 80 (0.5), and 200 (0.5)
Lr	$1.00E-05$	$1.00E-05$	$1.00E-05$	$1.00E-05$	$1.00E-05$	$1.00E-05$
Epoch	100	200	300	500	1000	2000
Accuracy	0.9623	0.9623	0.9623	0.9623	0.9623	0.9623
Loss	0.5104	0.4097	0.3033	0.1055	0.1112	0.2213

this, results are generated by varying the learning rate = 0.01, 0.001, 0.0001, and 0.00001, epochs = 100, 200, 300, 500, 1000, and 2000, and dropout = 0.2 and 0.5.

5. Conclusion and Future Scope

Software fault prediction is typically used to predict faults in software components. Machine learning techniques (e.g., classification) are widely used to tackle this problem. Deep neural network models built by the appropriate design decisions are crucial to obtain the desired classifier performance. This is especially desired when predicting fault proneness of software modules. When correctly identified, this could help in reducing the testing cost by directing the efforts more towards the modules identified to be fault prone. However, there is still a need to improve the prediction accuracy of these models. In this paper, an attempt has been made to build an efficient deep neural network model, based on the parameters such as the number of

hidden layers, number of nodes in each layer, and training details such as learning rate and regularization methods (such as $L2$ regularization and dropout regularization). An attempt has been also made to show the importance of hyperparameter tuning in developing efficient deep neural network models for predicting fault proneness of software module, and to compare the results with other machine learning algorithms. To evaluate the correctness of the proposed model, it is compared against other well-known machine learning models such as the random forest, decision trees, and naive Bayesian networks. The experiments were performed on 4 NASA system datasets (KC1, PC1, PC2, and KC3), selected from PROMISE repository which are freely available as public datasets. From Figure 12, it is observed that, in each case, the accuracy generated by the proposed deep neural network (DNN) with dropout is the highest as compared to other machine learning techniques for all the datasets. And, in most of the cases, the proposed DNN model with dropout detects more faults as compared to other machine learning techniques. It is also seen that DNN with dropout preforms better than DNN without dropout. Thus, it is shown that the proposed DNN model with dropout outperforms the other algorithms in most cases. In terms of future scope, it is intended to use more advanced deep learning techniques and explore more datasets from different resources. In addition, it would be promising to try some of feature generation techniques to generate the features which will help in improving the model's recall, F-measure, and also accuracy.

Data Availability

The datasets are taken from an online public data repository, namely, "PROMISE Software Engineering Repository" (http://promise.site.uottawa.ca/SERepository/datasets-page.html).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering—ICSE 2014*, Hyderabad India, May 2014.

[2] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy, May 2015.

[3] A. G. Liu, E. Musial, and M.-H. Chen, "Progressive reliability forecasting of service-oriented software," in *Proceedings of the 2011 IEEE International Conference on Web Services*, Washington, DC, USA, July 2011.

[4] S. S. Rathore and S. Kumar, "A decision tree logic based recommendation system to select software fault prediction techniques," *Computing*, vol. 99, no. 3, pp. 255–285, 2016.

[5] R. Malhotra and A. Jain, "Fault prediction using statistical and machine learning methods for improving software quality," *Journal of Information Processing Systems*, vol. 8, no. 2, pp. 241–262, 2012.

[6] 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*, vol. 59, pp. 170–190, 2015.

[7] M. O. Elish, A. H. Al-Yafei, and M. Al-Mulhem, "Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: a case study of Eclipse," *Advances in Engineering Software*, vol. 42, no. 10, pp. 852–859, 2011.

[8] Y. Peng, G. Kou, G. Wang, W. Wu, and Y. Shi, "Ensemble of software defect predictors: an ahp-based evaluation method," *International Journal of Information Technology & Decision Making*, vol. 10, no. 1, pp. 187–206, 2011.

[9] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[10] Y. Singh, A. Kaur, and R. Malhotra, "Prediction of fault-prone software modules using statistical and machine learning methods," *International Journal of Computer Applications*, vol. 1, no. 22, pp. 8–15, 2010.

[11] K. Dejaeger, T. Verbraken, and B. Baesens, "Toward comprehensible software fault prediction models using bayesian network classifiers," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 237–257, 2013.

[12] J. Cahill, J. M. Hogan, and R. Thomas, "predicting fault-prone software modules with rank sum classification," in *Proceedings of the 2013 22nd Australian Software Engineering Conference*, Hawthorne, Australia, 2013.

[13] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Applied Soft Computing*, vol. 33, pp. 263–277, 2015.

[14] L. Kumar, S. K. Sripada, A. Sureka, and S. K. Rath, "Effective Fault Prediction model developed using least square Support vector machine (LSSVM)," *Journal of Systems and Software*, vol. 137, pp. 686–712, 2018.

[15] B. Twala, "Predicting software faults in large space systems using machine learning techniques," *Defence Science Journal*, vol. 61, no. 4, pp. 306–316, 2011.

[16] A. Boucher and M. Badri, "Software metrics thresholds calculation techniques to predict fault-proneness: an empirical comparison," *Information and Software Technology*, vol. 96, pp. 38–67, 2018.

[17] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering—ICSE'16*, Austin, TX, USA, May 2016.

[18] E. Erturk and E. Akcapinar Sezer, "Iterative software fault prediction with a hybrid approach," *Applied Soft Computing*, vol. 49, pp. 1020–1033, 2016.

[19] D.-L. Miholca, G. Czibula, and I. G. Czibula, "A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks," *Information Sciences*, vol. 441, pp. 152–170, 2018.

[20] M. Samir, M. El-Ramly, and A. Kamel, "Investigating the use of deep neural networks for software defect prediction," in *Proceedings of the 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, Abu Dhabi, UAE, November 2019.

[21] H. Turabieh, M. Mafarja, and X. Li, "Iterated feature selection algorithms with layered recurrent neural network for software fault prediction," *Expert Systems with Applications*, vol. 122, pp. 27–42, 2019.

[22] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic, July 2017.

[23] F. Yucalar, A. Ozcift, E. Borandag, and D. Kilinc, "Multiple-classifiers in software quality engineering: combining predictors to improve software fault prediction ability," *Engineering Science and Technology, An International Journal*, vol. 23, no. 4, pp. 938–950, 2019.

[24] V. Duddu, N. Rajesh Pillai, D. V. Rao, and V. E. Balas, "Fault tolerance of neural networks in adversarial settings," *Journal of Intelligent & Fuzzy Systems*, vol. 38, no. 5, pp. 5897–5907, 2020.

[25] Y. Lyu and Y. Jiang, "Examination on avionics system fault prediction technology based on ashy neural network and fuzzy recognition," *Journal of Intelligent & Fuzzy Systems*, vol. 38, no. 4, pp. 3939–3947, 2020.

[26] L. Gong, S. Jiang, and L. Jiang, "Tackling class imbalance problem in software defect prediction through cluster-based over-sampling with filtering," *IEEE Access*, vol. 7, pp. 145725–145737, 2019.

[27] S. Huda, S. Alyahya, M. Mohsin Ali et al., "A framework for software defect prediction and metric selection," *IEEE Access*, vol. 6, pp. 2844–2858, 2018.

[28] P. S. Bishnu and V. Bhattacherjee, "software fault prediction using Quad tree-based K-means clustering algorithm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 6, pp. 1146–1150, 2012.

[29] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "BPDET: an effective software bug prediction model using deep representation and ensemble learning techniques," *Expert Systems with Applications*, vol. 144, Article ID 113085, 2020.

[30] Y. Lei, B. Yang, X. Jiang, F. Jia, N. Li, and A. K. Nandi, "Applications of machine learning to machine fault diagnosis: a review and roadmap," *Mechanical Systems and Signal Processing*, vol. 138, Article ID 106587, 2020.

[31] J. Zhang, J. Tian, T. Wen, X. Yang, Y. Rao, and X. Xu, "Deep fault diagnosis for rotating machinery with scarce labeled samples," *Chinese Journal of Electronics*, vol. 29, no. 4, pp. 693–704, 2020.

[32] M. Bashiri and A. Farshbaf Geranmayeh, "Tuning the parameters of an artificial neural network using central composite design and genetic algorithm," *Scientia Iranica*, vol. 18, no. 6, pp. 1600–1608, 2011.

[33] W.-Y. Lee, S.-M. Park, and K.-B. Sim, "Optimal hyperparameter tuning of convolutional neural networks based on the parameter-setting-free harmony search algorithm," *Optik*, vol. 172, pp. 359–367, 2018.

[34] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[35] H. Cho, Y. Kim, E. Lee, D. Choi, Y. Lee, and W. Rhee, "Basic enhancement strategies when using bayesian optimization for hyperparameter tuning of deep neural networks," *IEEE Access*, vol. 8, pp. 52588–52608, 2020.

[36] J. Moolayil, "Tuning and deploying deep neural networks," in *Learn Keras for Deep Neural Networks*, pp. 137–159, Springer, Berlin, Germany, 2018.

[37] A. Akl, I. El-Henawy, A. Salah, and K. Li, "Optimizing deep neural networks hyperparameter positions and values," *Journal of Intelligent & Fuzzy Systems*, vol. 37, no. 5, pp. 6665–6681, 2019.

[38] P. R. Bal and S. Kumar, "WR-ELM: weighted regularization extreme learning machine for imbalance learning in software fault prediction," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1355–1375, 2020.

[39] C. Manjula, "Software defect prediction using deep belief network with L1-regularization based optimization," *International Journal of Advanced Research in Computer Science*, vol. 9, no. 1, pp. 864–870, 2018.

[40] Y. Qu, X. Chen, Y. Zhao, and X. Ju, "Impact of hyper parameter optimization for cross-project software defect prediction," *International Journal of Performability Engineering*, vol. 14, no. 6, pp. 1291–1299, 2018.

[41] P. K. Kudjo, S. B. A formaley, S. Mensah, and J. Chen, "The significant effect of parameter tuning on software vulnerability prediction models," in *Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Sofia, Bulgaria, July 2019.

[42] A. Ng, *Neural Networks and Deep Learning*, Springer, Berlin, Germany, 2019.

[43] http://promise.site.uottawa.ca/SERepository/datasets-page.html.