Journal on Artificial Intelligence
DOI:10.32604/jai.2020.014829
Article

Tech Science Press

A Learning Framework for Intelligent Selection of Software Verification Algorithms

Weipeng Cao[1], Zhongwu Xie[1], Xiaofei Zhou[2], Zhiwu Xu[1], Cong Zhou[1], Georgios Theodoropoulos[3] and Qiang Wang[3,*]

[1]Shenzhen University, Shenzhen, China
[2]Hangzhou Dianzi University, Hangzhou, China
[3]Southern University of Science and Technology, Shenzhen, China
[*]Corresponding Author: Qiang Wang. Email: wangq8@sustech.edu.cn
Received: 20 October 2020; Accepted: 29 November 2020

Abstract: Software verification is a key technique to ensure the correctness of software. Although numerous verification algorithms and tools have been developed in the past decades, it is still a great challenge for engineers to accurately and quickly choose the appropriate verification techniques for the software at hand. In this work, we propose a general learning framework for the intelligent selection of software verification algorithms, and instantiate the framework with two state-of-the-art learning algorithms: Broad learning (BL) and deep learning (DL). The experimental evaluation shows that the training efficiency of the BL-based model is much higher than the DL-based models and the support vector machine (SVM)-based models, while the prediction accuracy of the DL-based model is much higher than other models.

Keywords: Software verification; algorithm selection; broad learning; deep learning

1 Introduction

Software verification is a widely used technique to ensure the correctness of software. In the past decades, many software verification algorithms and tools have been developed, e.g., UAutomizer [1], CBMC [2], UFO [3], CPAChecker [4], in order to scale up the performance of software verification. We refer to [5,6] for a detailed overview of the progress. When applying verification algorithms or tools to the real-world software, it is preferable for the engineers to know which algorithm or tool is the most suitable one for the software at hand. However, most existing software verification algorithms are designed or optimized for a specific type of software codes. Knowing the weakness or strengths of each algorithm or tool requires a deep understanding of the underlying verification theory, which is usually a difficult task for engineers.

Recently, some researchers have attempted to use machine learning techniques to achieve the automatic selection and fusion of software verification algorithms, e.g., [7–10]. However, their work suffers from two limitations: (1) They have not formed a unified solution for the intelligent selection of software verification algorithms. Most of them are only suitable for specific competitions or applications and cannot be directly applied to the source code verification of complex systems. Besides, their work mainly focuses on predicting the ranking of available tools in terms of the verification performance, instead of selecting the most suitable tool. (2) The machine learning techniques involved in these works are relatively simple, and most of them have only tried SVM [11], which has several notorious weaknesses: (a) Compared with deep neural networks, the complexity of the SVM model is relatively low; (b) SVM cannot be directly applied to deal with multi-classification problems; (c) It is difficult to train the SVM model when the number of training samples is very large.

In this work, we propose a general learning framework for the intelligent selection of software verification algorithms. The framework clearly describes the mapping from software source codes to the corresponding verification algorithms. Specifically, one should first collect the massive source codes and existing software verification algorithms, and then describe these data by designing a set of appropriate meta-features and constructing a corresponding meta-dataset. After that, one can apply various machine learning techniques to mine the relationship between the source codes and the performance of the verification algorithms, which can be used to infer the final selection model. Given a source code segment or task to be verified, the model can predict the most appropriate verification algorithm according to the meta-features of the source code or task, thereby greatly improving the efficiency of software verification in practical engineering.

To verify the effectiveness of the proposed framework, we have collected a related data set from [8] and used static analysis techniques to design and extract meta-features from the source codes. The features include the variable role usage, control flow metrics and loop patterns, etc. We chose two most representative neural networks (i.e., BL [12] and DL [13]) to train the automatic selection model. BL represents a type of neural network that uses the non-iterative training mechanism, while DL represents a type of traditional neural network that is trained based on the gradient descent and its variants. Both of them can overcome the above-mentioned shortcomings of SVM. Therefore, the intelligent selection models trained with them are expected to have higher feasibility in practice. The contributions of this work can be summarized as follows:

(1) A general learning framework for the intelligent selection of software verification algorithms is proposed and two representative neural network based learning algorithms (i.e., BL and DL) are implemented in the framework, which gives a new perspective for improving the efficiency of software verification in practice;

(2) The performance of BL and DL on algorithm selection for software verification has been comprehensively evaluated for the first time. We have found that under the proposed learning framework, the DL-based and BL-based models can respectively achieve much higher prediction accuracy and faster training speed than the most commonly used SVM-based model, which provides valuable guidelines for engineers in selecting neural networks to model the problem.

We organize the reminder of this paper as follows. In Section 2, we briefly review the related work on the automatic selection of software verification algorithms. We present the details of the proposed learning framework in Section 3. In Section 4, we introduce the experimental setting and the corresponding results. We conclude this paper in Section 5.

2 Related Work

The application of algorithm selection [14] into software model checking is relatively new, and has not yet been systematically investigated. The first effort to our knowledge was [10], where the authors proposed a technique called MUX that is able to construct a strategy selector for a set of features of the input program and a given number of strategies. The underlying learning technique is based on SVM. In [7, 8], the authors presented a sophisticated set of empirical software metrics consisting of variable role usage, control flow metrics and loop patterns. The prime goal was to explain the performance of different model checkers in SV-COMP using the above metrics. They further presented a SVM-based portfolio solver for software model checking based on these metrics. Their experiments show that the portfolio solver would be the overall winner of SV-COMP in three consecutive years (i.e., 2014-2016). In [15], the authors look at the ranking prediction problem of software model checkers. A ranking of the candidate model checkers could help users choose the appropriate one for the program at hand.

More recently, the authors in [16] present a specific strategy selector, in order to leverage on the numerous model checking techniques integrated in the tool CPAChecker[1]. The selector also takes as input

[1] https://cpachecker.sosy-lab.org/

a set of strategies and the selection model that represents some information about the program and its property specification, and returns as output the strategy that is predicted to be useful. However, their strategy selector only works for CAPChecker, since the strategies are mere different parameter specifications of CPAChecker. The selection model is explicitly defined by the developers. No machine learning techniques are applied. In [9], the authors presented a tool PeSCo that can predict a (likely best) sequential combination of model checkers (i.e., different configurations of CPAChecker) on a given model checking task. The approach is based on SVM, and can predict rankings of model checkers on tasks.

3 A Learning Framework for Intelligent Selection of Software Verification Algorithm

In Fig. 1, we show the proposed learning framework for the intelligent selection of software verification algorithms. The core module of the framework includes three parts: the training module (shown by the red arrows in the figure), the prediction module (shown by the blue arrows), and the verification module (shown by the green arrows). The success of the last two modules depends on the completion quality of the first module. In this paper we focus on how to complete the first module with high quality.

Specifically, three problems need to be solved before using the machine learning technique to train a model for the intelligent selection of software verification algorithms: (1) How to formalize the software verification algorithm selection problem as a machine learning problem? (2) How to get suitable training data? (3) How to choose an appropriate machine learning algorithm?

Note that one can decompose a software into multiple code fragments and use the trained algorithm selection model to predict the appropriate verification algorithms and tools for them, and then complete the verification of the original software system.

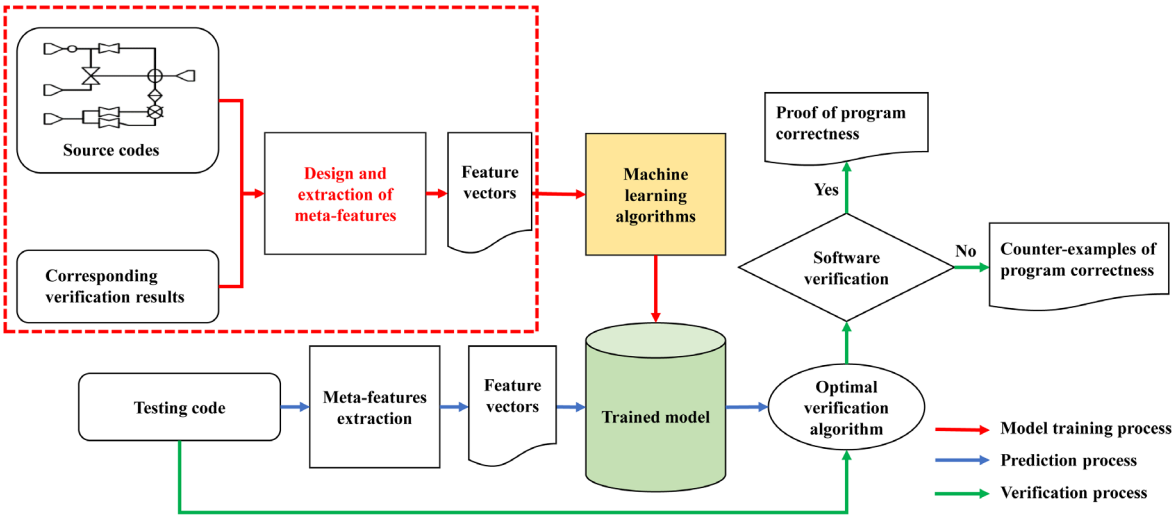

Figure 1: A learning framework for the intelligent selection of software verification algorithms

3.1 Representing Software Verification Tasks

Here we show how to encode the algorithm selection problem for software verification tasks into the machine learning problem and construct the corresponding data sets for model training.

Definition 1. *A software verification task* sv *is denoted by a triple* $sv = \{f, p, type\}$*, where* f *is the source file,* p *is the property to be verified, and* $type$ *is the property type. We denote by* SV *a set of tasks.*

For each task $sv = \{f, p, type\} \in SV$, we define the feature vector as $\times(sv) = (\boldsymbol{m}_{vrole}, \boldsymbol{m}_{cfg}, \boldsymbol{m}_{loop}, type)$, where $\boldsymbol{m}_{vrole}$ is the vector of variable role based metrics, $\boldsymbol{m}_{cfg}$ is that of control flow based metrics, $\boldsymbol{m}_{loop}$ is that of loop pattern based metrics, and $type \in \{0, 1, 2, 3\}$ encodes whether the property is related to reachability, memory safety, overflow or termination. These features can be extracted by using static analysis techniques as reported in [8]. The expected verification output (i.e.,

whether property p holds on the source file f) is regardless of the tool being used and defined as the function $ExpAns: SV \rightarrow \{true, false\}$. This function gives the ground truth of each task. Given a tool t and a task $sv = \{f, p, type\}$, its verification result is denoted by $ans_{t,sv} \in \{true, false, unknown\}$, where $unknown$ indicates the tool is unable to verify if the property holds or not.

Definition 2. *Given a verification task sv and a tool t, the labeling function $L_t(sv)$ is defined in the following manner: (1) $L_t(sv) = 1$ if tool t gives the correct answer on sv, (2) $L_t(sv) = 2$ if tool t outputs $unknown$, and (3) $L_t(sv) = 3$ if tool t gives the incorrect answer (i.e., $ans_{t,sv} \neq ExpAns(sv)$).*

3.2 Data Preparation

In this study, the experimental data were collected from [8]. We used the same method as [7, 8] to design the meta-features of the source codes, including the variable role usage, control flow metrics and loop patterns, etc. They can be obtained through static analysis techniques. After the above data preprocessing operation, we get a data-set containing 31371 samples of source codes and 35 verification tools. Each sample has 46 attributes, which contain the meta-features of source codes and the information of verification algorithms. There are three classes (i.e., true, false, and unknown) in the data-set, which indicate the feasibility of a specific verification algorithm on a specific code fragment. We divided the data set into a training set and a testing set by 8:2 and kept the relative proportion of each class in them to be consistent.

3.3 Machine Learning Based Algorithms Selection

From Section 2, one can observe that the most commonly used machine learning technique in modeling the verification algorithms selection problem is SVM. Although the existing SVM-based models demonstrate the feasibility of the data-driven approach in solving the algorithm selection problem, it cannot be ignored that the modeling with SVM suffers from the following disadvantages:

(1) SVM can be regarded as a feed-forward neural network with a single hidden layer, and its activation function is the kernel function, which is responsible for mapping data features from the original space into a high-dimensional space. It is difficult for such a training mechanism to allow full fusion and transformation between data features, resulting in related models facing great challenges in dealing with problems with complex features.

(2) The efficiency of SVM is limited when handling large data sets. This is because SVM uses the quadratic programming technique to calculate its model parameters, which is sensitive to the number of samples.

(3) SVM cannot be directly used to solve multi-classification problems. Although one can transform a multi-classification problem into multiple binary classification problems and use SVM to train the model, this strategy is inefficient.

Based on the above considerations, we choose neural networks as the machine learning algorithm module, which can avoid the above-mentioned problems. One of the advantages of neural networks is that they have powerful feature transformation and extraction ability. In the past decade, related algorithms have made breakthroughs in many fields [17]. Existing representative neural networks include multilayer perceptron (MLP) [13], convolutional neural networks (CNN) [18], neural networks with random weights (NNRW) [17], broad learning system (BLS) [12], etc. From the perspective of training mechanism, these algorithms can be divided into two categories: traditional neural networks with the iterative training mechanism (e.g., MLP and CNN) and randomized neural networks with the non-iterative training mechanism (e.g., NNRW and BLS).

Considering that the data set used in this study does not have an obvious local feature structure, we chose MLP with a fully connected structure as the training algorithm instead of CNN. Moreover, to completely evaluate the effectiveness of neural networks under the proposed framework, we also chose the BLS to compare with MLP. To clearly distinguish the difference between these two neural networks, we use DL to represent MLP and BL to represent BLS.

3.4 Broad Learning

Broad learning (BL) is a feed-forward neural network with a non-iterative training mechanism. A typical network structure of BL is shown in Fig. 2, where $\boldsymbol{X}$ refers to the input data, $\boldsymbol{\omega_1}$ refers to the input weights that connects the input layer and the feature layer, $\boldsymbol{Z}$ is the output of the feature layer, $\boldsymbol{\omega_2}$ refers to the weights between the feature layer and the enhancement layer, $\boldsymbol{H}$ is the output of the enhancement layer, $\boldsymbol{b_1}$ and $\boldsymbol{b_2}$ refers to the hidden biases of feature nodes and enhancement nodes respectively, $\boldsymbol{\beta}$ denotes the output weights connecting the feature layer and the enhancement layer to the output layer, and $\boldsymbol{Y}$ is the output of the model.

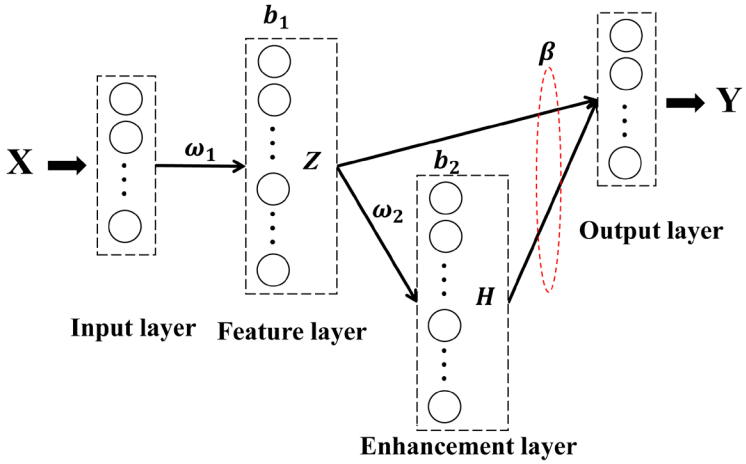

Figure 2: The network structure of BL

Given a training data set $\{\boldsymbol{X}, \boldsymbol{T}\} \in R^{(d+c)*N}$, where d refers to the dimension of input data, c denotes the number of classes, and N is the number of samples. Suppose that there are n feature nodes and m enhancement nodes in a BL, and its transformation function of the feature layer and the activation function of enhancement layer are denoted as $\phi(\cdot)$ and $g(\cdot)$, respectively. The output of the i_{th} feature node can be represented as $\boldsymbol{Z_i} = \phi(\omega_i \cdot \boldsymbol{X} + b_i)$, and then the output of the feature layer can be expressed as $\boldsymbol{Z} = [\boldsymbol{Z_1}, \boldsymbol{Z_2}, \cdots, \boldsymbol{Z_n}]$. The output of the enhancement layer can be expressed as $\boldsymbol{H} = g(\omega_2 \cdot \boldsymbol{Z} + b_2)$ and the BL model can be represented as

$$\boldsymbol{Y} = [\boldsymbol{Z}|\boldsymbol{H}] \cdot \boldsymbol{\beta} \tag{1}$$

Let $\boldsymbol{A} = [\boldsymbol{Z}|\boldsymbol{H}]$, one can re-write (1) as $\boldsymbol{Y} = \boldsymbol{A} \cdot \boldsymbol{\beta}$. The optimization objective of the BL model is as follows:

$$Minimize: \frac{1}{2}||\beta||^2 + \frac{\lambda}{2}||\boldsymbol{Y} - \boldsymbol{T}||^2 \tag{2}$$

where λ refers to the regularization factor.

Different from traditional neural networks, BL does not solve (2) through iterative optimization methods such as the gradient descent but uses a non-iterative training mechanism. Specifically, $\boldsymbol{\omega_1}$, $\boldsymbol{b_1}$, $\boldsymbol{\omega_2}$, and $\boldsymbol{b_2}$ are assigned randomly and kept fixed throughout the training process. The output weights $\boldsymbol{\beta}$ are obtained analytically as follows:

If $N > (n + m)$, then

$$\boldsymbol{\beta} = (\frac{\boldsymbol{I}}{\lambda} + \boldsymbol{A^T A})^{-1} \boldsymbol{H^T T} \tag{3}$$

else

$$\boldsymbol{\beta} = \boldsymbol{H^T} (\frac{\boldsymbol{I}}{\lambda} + \boldsymbol{A A^T})^{-1} \boldsymbol{T} \tag{4}$$

where $\boldsymbol{I}$ refers to the identity matrix.

3.5 Deep Learning

Deep learning (DL) has made breakthroughs in many fields in recent years [18]. Although there are many existing DL algorithms, most of them share the same training mechanism, that is, iteratively calculating the model parameters based on the gradient descent method or its variants.

A typical DL model with a fully connected network structure is shown in Fig. 3, where $\{\boldsymbol{H_1}, \boldsymbol{H_2} \ldots., \boldsymbol{H_m}\}$ refers to the outputs of the $\{1_{st},\ 2_{nd}, \ldots,\ m_{th}\}$ hidden layers, respectively. $(\boldsymbol{\omega_i}, \boldsymbol{b_i})$ refers to the weights and hidden biases of the i_{th} hidden layer. Other symbols have the same meaning as BL.

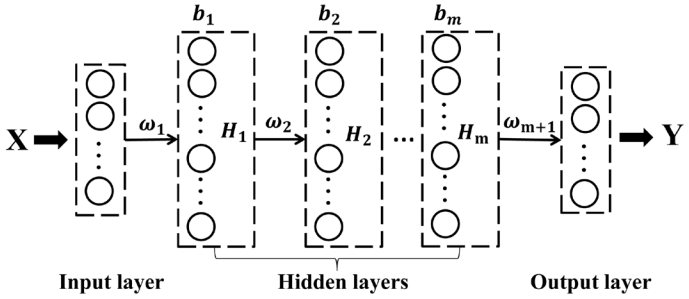

Figure 3: The network structure of the deep learning model

Given a training data set $\{\boldsymbol{X}, \boldsymbol{T}\} \in R^{(d+c)*N}$, the optimization objective of the DL model is usually expressed as follows:

$$Minimize: \boldsymbol{\mathcal{L}} = \frac{1}{N}\sum_{i=1}^{N} l(y_i, t_i) \tag{5}$$

where y_i and t_i refer to the predictive label and the actual label of the i_{th} sample. For regression problems, $\boldsymbol{\mathcal{L}}$ is usually set to the mean square error; for classification problems, $\boldsymbol{\mathcal{L}}$ is usually set to the cross-entropy loss.

Most of the existing DL models use the gradient descent method or its variants to achieve (5). A simple schematic diagram of the gradient descent method is shown in Fig. 4.

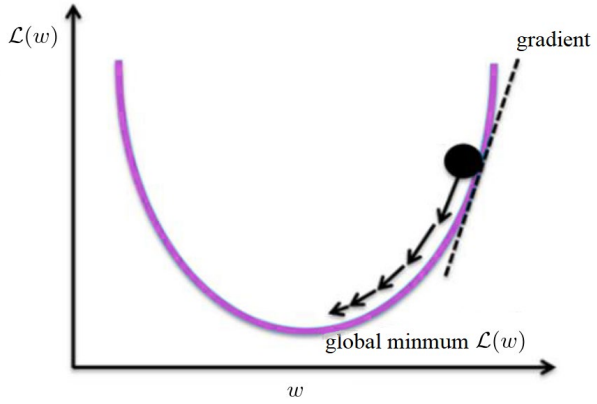

Figure 4: A simple schematic diagram of the gradient descent method

During the process of model training, one can update the model parameters according to the negative gradients to achieve the goal of reducing the global loss of the model (as shown in Fig. 4). Specifically, first randomly initializing the parameters of the DL model and then input $\boldsymbol{X}$ into it. The model will output the predictive labels $\boldsymbol{Y}$ corresponding to all input samples. According to (5), one can get the current loss $\boldsymbol{\mathcal{L}}$. Then calculating the partial derivative of the weights and bias of $\boldsymbol{\mathcal{L}}$ for each layer based the chain-derivative rule, that is, $\frac{\partial \mathcal{L}}{\partial \omega_i}$ and $\frac{\partial \mathcal{L}}{\partial b_i}$, where $\boldsymbol{\omega_i}$ and $\boldsymbol{b_i}$ refer to the weights and bias of the i_{th} hidden layer. After that,

updating the current $\boldsymbol{\omega_i}$ and $\boldsymbol{b_i}$:

$$\boldsymbol{\omega_i} = \boldsymbol{\omega_i} - \boldsymbol{\alpha}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\omega_i}}, \boldsymbol{b_i} = \boldsymbol{b_i} - \boldsymbol{\alpha}\frac{\partial \mathcal{L}}{\partial \boldsymbol{b_i}} \quad (6)$$

where α is the learning rate, which is used to control the pace of updates.

The above process is performed iteratively until $\mathcal{L}$ reaches a threshold or the number of iterations reaches the preset maximum value.

4 Experimental Evaluation

4.1 Parameter Setting

We chose SVM [11] as the baseline algorithm, which is the most commonly used machine learning technique in the existing related work. For SVM, the regularization parameter C was set to 1.0, the kernel type was set to RBF, and the training strategy was set to one-vs-rest. For BL [12], the numbers of feature nodes and enhancement nodes were set to 100 and 500, respectively. Moreover, its randomized parameters were assigned from [–1, 1] under a uniform distribution, the regularization factor was set to 1, and the transformation function was set to the linear transformation. To fairly compare with BL, we set the number of hidden layers in DL to 2 and the number of nodes in the first hidden layer and the second hidden layer to 100 and 500, respectively. Moreover, the learning rate and the optimization algorithm of the DL model was set to 0.0003 and Adam, respectively. To compare the performance of BL and DL more comprehensively, three non-linear functions were chosen as their activation functions, that is, ReLU, Sigmoid, and Tanh functions. Their mathematical forms are as follows:

$$ReLU: y = max(0, x), Sigmoid: y = \frac{1}{1+e^{-x}}, Tanh: y = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \quad (7)$$

In our experiments, the performance evaluation indexes include the testing accuracy, training accuracy, training time, and the testing standard deviation. Among them, the testing standard deviation (SD) can be obtained by using the following equation:

$$Testing\ SD: \sqrt{\frac{\sum_{j=1}^{S}(e_j-\overline{e})^2}{S-1}} \quad (8)$$

where S is the number of independent experiments for each case, e_j is the prediction error of the model in the j_{th} experiment, and $\overline{e}$ is the average prediction error of S experiments. In this study, S was set to 10, which means that we did each experiment independently ten times.

4.2 Experimental Results and Analysis

4.2.1 BL vs. DL

First, we compare the performance of BL and DL under the same number of nodes (46-100-500-3). The experimental results are shown in Tab. 1 and the best results are in bold. Note that the BL-ReLU means that the activation function of the BL model is ReLU. The naming rules of other models are the same.

It can be observed from Tab. 1 that the training speed of BL is much faster than that of SVM and DL. For example, the training speed of BL-ReLU is 50 times faster (0.6553 s *vs.* 33.0878 s) and 1143 times (0.6553 s *vs.* 749.2075 s) than that of SVM and DL-ReLU, respectively.

The predictive ability of the BL model is comparable to that of the SVM model but significantly weaker than that of the DL model. The DL model using Tanh as the activation function achieves the highest testing accuracy in our experiments. Compared with the SVM-based model, the DL-based model improves the testing accuracy by 21.13%: (0.9211 – 0.7604) / 0.7604 * 100% = 21.13%.

Table 1: The performance comparison of BL, DL, and SVM

Algorithm	Testing accuracy	Training accuracy	Training time (s)	Testing SD
SVM	0.7604	0.7734	33.0878	-
BL-ReLU	0.7459	0.7577	0.6553	0.0045
BL-Sigmoid	0.7480	0.7594	0.8640	0.0027
BL-Tanh	0.7512	0.7680	1.5302	0.0012
DL-ReLU	0.9211	0.9559	749.2075	0.0043
DL-Sigmoid	0.9107	0.9204	956.9280	0.0024
DL-Tanh	0.9203	0.9519	997.9541	0.0032

4.2.2 Sensitivity Analysis of BL and DL Models to the Number of Hidden Nodes

According to the BL theory [12,19], enhancement nodes play a more important role than feature nodes. Therefore, we analyze the sensitivity of the BL model to the number of enhancement nodes. The enhancement layer corresponds to the second hidden layer in DL. To ensure the fairness of the experiment, we chose to analyze the sensitivity of the DL model to the number of its second hidden layer nodes. Specifically, we selected 10 values from [50, 500] with 50 steps as the number of enhancement nodes in BL and the number of the second hidden layer nodes in DL, and then evaluated the testing accuracy of the corresponding models. Since the experimental results show that the experimental phenomena corresponding to different activation functions are similar in this experiment, here we take the BL-Tanh and DL-Tanh models as examples to show our experimental results (as shown in Fig. 5).

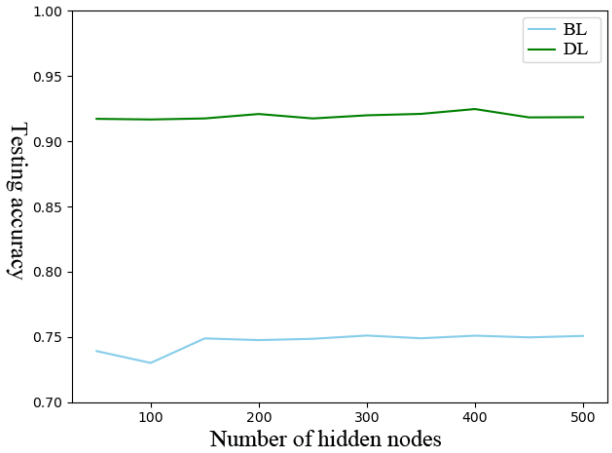

Figure 5: The sensitivity of BL and DL models to the number of hidden nodes

It can be observed from Fig. 5 that in BL, with the increase of the number of enhancement nodes, the test accuracy of the model fluctuates first, but when the number of enhancement nodes exceeds 150, the prediction performance of the model tends to be stable. This phenomenon implies that the prediction performance of the BL model is slightly sensitive to the number of enhancement nodes when the number is relatively small. For DL, as the number of the second hidden layer nodes increases, the test accuracy of the model does not change significantly, which implies that the prediction performance of the DL model is not sensitive to the number.

A speculative explanation for the above phenomenon is that for BL, since its training process is non-iterative, one can infer that the feature extraction quality of its model largely depends on the random mapping quality of the enhancement layer. In this case, when the number of enhancement nodes is relatively small, the feature extraction quality will be sensitive to the number; as the number of enhancement nodes increases, the randomness here is gradually diluted, so the model performance tends to be stable. For DL,

its parameters are calculated iteratively based on the gradient descent method. Under our parameter configuration, this method can always get a relatively stable solution, so the model performance is insensitive to the number of the second hidden layer nodes.

4.2.3 Sensitivity Analysis of the DL Model to the Learning Rate and Optimizer

To analyze the sensitivity of the DL model to the learning rate and optimizer, we evaluated the test accuracy of the model under four learning rates and three optimizers. Note that our experimental results show that the experimental phenomena of models with different activation functions are similar, so here we take the results corresponding to the DL-Tanh model as an example to perform the hyper-parameters sensitivity analysis. The details of the experimental results are shown in Fig. 6.

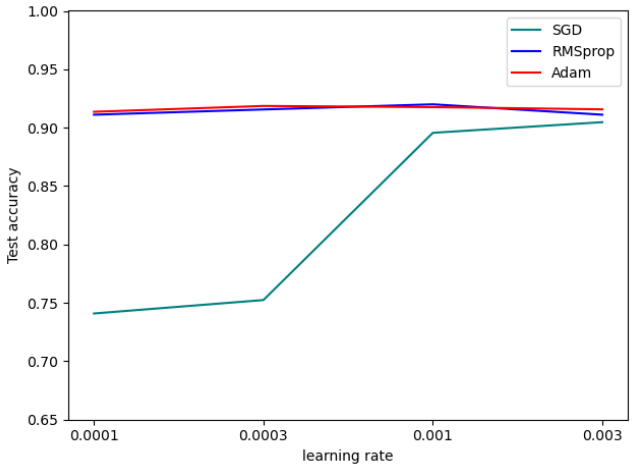

Figure 6: The sensitivity of the DL model to the learning rate and optimizer

It can be observed from Fig. 6 that if SGD [20] were selected as the optimizer of the DL model, the model performance would be sensitive to the value of the learning rate. For example, when the learning rate is 0.003, the test accuracy of the model is much higher than that of the model with other parameters. For the DL models using RMSprop [21] and Adam [22] as the optimizer, the different values of the learning rate will not have a significant impact on the model performance, and the test accuracy of the models is higher than that of the model with the SGD optimizer. From this phenomenon, one can infer that the choice of optimizer plays a more important role than the learning rate.

Remark 1. From the above experiments, it can be inferred that the advantage of the DL model is high prediction accuracy, while its disadvantages include slow training speed, many hyper-parameters, and the model performance is sensitive to the choice of the optimizer. As for BL, its advantages are simple implementation and high training efficiency, while its disadvantage is the relatively weak ability in data feature transformation and extraction, resulting in the prediction ability of the model is not as good as the DL model on some complex problems. Compared with the SVM-based model, both BL and DL models show significant advantages in certain performance indicators, such as the training efficiency and the testing accuracy, which provides a new direction for the research of the automatic selection of verification algorithms.

Remark 2. The non-iterative training mechanism gives BL extremely fast training speed, which is beneficial for applications that have stringent requirements on modeling speed. However, its feature extraction ability is relatively weak. For DL, the iterative training mechanism and various optimization algorithms enable it to better transform and extract data features, but its training process is often time-consuming and requires high hardware computing power. How to combine the training speed advantage of BL and the feature processing power of DL to efficiently solve the complex problems in practical engineering is a problem worth studying in the future.

5 Conclusions

To improve the efficiency of software verification in practical engineering, we proposed a general learning framework for the intelligent selection of software verification algorithms in this paper. Based on the proposed learning framework, we comprehensively evaluated the performance of two representative neural networks (i.e., BL and DL) in the selection of software verification algorithms. The experimental results show that the BL-based model can achieve considerable prediction accuracy as the most commonly used SVM-based model, but its training speed can be 50 times faster than the latter. As for the DL-based model, its prediction accuracy can be 21.13% higher than that of the SVM-based model, but its training efficiency is behind other models due to the iterative training mechanism. These research results provide valuable guidelines for researchers and engineers in the research of automatic software verification.

In the future, we will develop a hybrid method that combines the advantages of BL and DL, give full play to the powerful feature extraction capability of DL and the efficient training efficiency of BL, and further improve the generalization ability and learning efficiency of the current algorithm selection model.

Acknowledgement: The authors would like to thank some colleagues from Shenzhen University, Hangzhou Dianzi University and Southern University of Science and Technology for their constructive suggestions.

Funding Statement: This work was supported by the National Natural Science Foundation of China (61836005) and the Opening Project of Shanghai Trusted Industrial Control Platform (TICPSH202003008-ZC).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. Heizmann, Y. F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke *et al.,* "Ultimate automizer and the search for perfect interpolants," in *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Cham, Berlin, Heidelberg, vol. 10806, pp. 447–451, 2018.

[2] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, vol. 8413, pp. 389–391, 2014.

[3] A. Albarghouthi, Y. Li, A. Gurfinkel and M. Chechik, "Ufo: A framework for abstraction-and interpolation-based software verification," in *Int. Conf. on Computer Aided Verification*, Springer, Berlin, Heidelberg, vol. 7358, pp. 672–678, 2012.

[4] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Int. Conf. on Computer Aided Verification*, Springer, Berlin, Heidelberg, vol. 6808, pp. 184–190, 2011.

[5] E. M. Clarke, T. A. Henzinger, H. Veith and R. Bloem, Handbook of Model Checking, Cham: Springer, 2018.

[6] S. P. Miller, M. W. Whalen and D. D. Cofer, "Software model checking takes off," *Communications of the ACM*, vol. 53, no. 2, pp. 58–64, 2010.

[7] Y. Demyanova, T. Pani, H. Veith and F. Zuleger, "Empirical software metrics for benchmarking of verification tools," in *Int. Conf. on Computer Aided Verification*, Springer, Cham, San Francisco, CA, USA, vol. 9206, pp. 561-579, 2015.

[8] Y. Demyanova, T. Pani, H. Veith and F. Zuleger, "Empirical software metrics for benchmarking of verification tools," *Formal Methods in System Design*, vol. 50, no. 2–3, pp. 289–316, 2017.

[9] C. Richter and H. Wehrheim, "Pesco: Predicting sequential combinations of verifiers," in *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Cham, Prague, Czech Republic, vol. 11429, pp. 229–233, 2019.

[10] V. Tulsian, A. Kanade, R. Kumar, A. Lal and A. V. Nori, "Mux: Algorithm selection for software model checkers," in *Proc. of the 11th Working Conf. on Mining Software Repositories (MSR 2014), Association for Computing Machinery,* New York, NY, USA, pp. 132–141, 2014.

[11] C. Cortes and V. N. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[12] C. L. P. Chen, Z. L. Liu and S. Feng, "Universal approximation capability of broad learning system and its

structural variations," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 4, pp. 1191–1204, 2018.

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

[14] J. R. Rice, "The algorithm selection problem," in *Advances in Computers*, Elsevier, vol. 15, pp. 65–118, 1976.

[15] M. Czech, E. Hüllermeier, M. C. Jakobs and H. Wehrheim, "Predicting rankings of software verification tools," in *Proc. of the 3rd ACM SIGSOFT Int. Workshop on Software Analytics*, New York, NY, USA, pp. 23–26, 2017.

[16] D. Beyer and M. Dangl, "Strategy selection for software verification based on boolean features," in *Int. Sym. on Leveraging Applications of Formal Methods*, Springer, Cham, Rhodes, Greece, vol. 11245, pp. 144–159, 2018.

[17] W. P. Cao, X. Z. Wang, Z. Ming and J. Z. Gao, "A review on neural networks with random weights," *Neurocomputing*, vol. 275, pp. 278–287, 2018.

[18] F. J. Xu, V. N. Boddeti and M. Savvides, "Local binary convolutional neural networks," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, Honolulu, HI, pp. 4284–4293, 2017.

[19] S. Feng and C. L. P. Chen, "Fuzzy broad learning system: A novel neuro-fuzzy model for regression and classification," *IEEE Transactions on Cybernetics*, vol. 50, no. 2, pp. 414–424, 2018.

[20] Y. Z. Li and Y. Y. Liang, "Learning overparameterized neural networks via stochastic gradient descent on structured data," in *Advances in Neural Information Processing Systems*, Montréal, Canada, pp. 8157–8166, 2018.

[21] M. C. Mukkamala and M. Hein, "Variants of rmsprop and adagrad with logarithmic regret bounds," arXiv preprint arXiv:1706.05507, 2017.

[22] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint, arXiv:1412.6980, 2014. [Online]. https://arxiv.org/abs/1412.6980