Semi-supervised Multi-task Learning Using Convolutional AutoEncoder for Faulty Code Detection with Limited Data

Anh Phan Viet*, Khanh Nguyen Duy Tung, Lam Bui Thu

Le Quy Don Technical University, 236 Hoang Quoc Viet St., Hanoi, Vietnam

Abstract

This paper proposes a semi-supervised multitask learning framework to overcome the limited data problem for faulty code prediction. The framework has two parts including 1) a convolutional autoencoder performing the auxiliary task that is learning the latent representations of programs and 2) a sequence-based convolution for the main task of faulty code prediction. The keys to reduce the data requirement and remain the prediction accuracy are 1) latent representations can support to generate good features for the classification task and 2) self-transfer learning for the joining part according to unsupervised manner and then fine-tuning with labeled data instead of training from scratch. To evaluate the benefits of pretraining and latent representations, we tried to feed the model with different amount of labelled data. Our experiments on four real datasets of software fault prediction have shown that pretraining the autoencoder achieves better performance and multitask learning makes the model able to learn with limited labelled data. Specifically, the model was trained with 50% or 75% can reach the performance as trained with 100% of data.

Keywords: Faulty Code Detection, Semi-supervised Learning, Multi-task Learning, Self-supervised learning, Convolutional AutoEncoder

*Corresponding author

Email address: anhpv@lqdtu.edu.vn (Anh Phan Viet)

1. Introduction

Detecting faulty modules before releasing projects is crucial to software industry. Faulty code contains specific bugs, e.g. division by zero, infinite loops and buffer overflow that make programs work incorrectly such as out of memory, execution interruption, or generating the unexpected output. It is infeasible to prevent all bugs in software components due to various causes including programmer's skill, miscommunication, buggy third-party tools and so on. According to annual reports, software bugs have led to many devastating consequences in terms of the loss of finance, company reputation and customer satisfaction [1]. The later the bugs are detected in the development process, the more serious the consequences are. For above reasons, faulty code detection (FCD) is a hot topic for both academia and industry in the field of software engineering.

There are two major approaches to build machine learning models for detecting faulty modules. The traditional methods design software metrics, e.g. function points and the complexity to estimate the characteristics of programs and then train detectors based on such features [2, 3, 4]. The main obstacles are extracting good features that are highly relevant to software bugs. Software bugs depend on specific scenarios and only appear when satisfying certain conditions. Thus, the surface features are difficult to capture them. For example, given two C statements `scanf("%s",string)` and `scanf("%10s",string)` where `string` is a character array, the first one potentially contains a buffer overflow bug due to not verifying the input sequence length, while the second one is safe. Recently, investigating deep neural networks to automatically generate sophisticated features from different program representations has made many breakthroughs in program analysis problems. Learning syntactic structures on abstract syntax trees (ASTs) is beneficial to program classification [5, 6]. For faulty code prediction, current studies have focused on both representations of ASTs and assembly instructions to develop deep models [7, 8]. According to several reports, deep models on assembly instructions can achieve higher performance because they are closed to machine code and show the execution process of programs [7, 9].

However, both the approaches have been suffered from the imbalance and lack of labelled data. Indeed, collecting large amount of historical source code from software projects is not trivial due to confidentiality. Moreover, the number of clean modules normally dominates that of defective modules while detecting the defective ones is more preferable. As a proof, the software metrics datasets in PROMISE repository [1] [10] just contain hundreds to several thousands instances including the duplication and the defective ratios range from 7% to 33%. Similarly, open source projects like camel (enterprise integration framework) and jEdit (text editor designed for programmers) were utilized in several researches have only hundred files [8].

Many efforts have been made to overcome the data shortage problem in the software engineering field. Some works leverage the data from previous versions in building models to detect faulty modules in the new version[11]. These methods result in the over-fitting problem since many instances may be available in both training and test sets. Another selection is the transfer learning technique that uses the data from other projects to pre-train and then fine-tune the model with the current project data. Reported by many works, transfer learning is an efficient solution to tackle the limited data problem. This is one of the keys to bring deep neural networks to the real world applications.

This paper proposes a new deep neural network that is able to learn with smaller amount of labelled data to overcome the data shortage problem[2]. The model has two sub-networks performing different tasks including an autoencoder to generate program latent representations and a classifier for prediction. The sub-networks share two some first layers of convolution that undertakes encoding programs for the autoencoder and extracting features for the classifier. The motivation for building the joint model comes from the advantages of the

[1] `http://promise.site.uottawa.ca/SERepository/`

[2] Our implementation is publicly available at `https://github.com/pvanh/ae_cnn_multitask`

autoencoder. In the reduction side, the latent representations contains semantic features such that the decoder side can reconstruct input programs. These semantic features are beneficial to distinguish programs. We also propose a learning strategy using semi-supervised learning to increase the performance of the model. The autoencoder branch is trained by unsupervised manner. Then, we continue training the whole model with labelled data instead of from scratch. This is feasible in practice because unlabelled data can be collected from a large amount of open source files in the internet.

In summary, this paper makes the following contributions:

- Designing an autoencoder-based multitask network for faulty code detection problem.
- Formulation of training the network by self-transfer and semi-supervised manner to reduce the labelled data need.
- Comprehensively assessing the ability to learn with limited data according to various criteria such as classification performance (Accuracy, F1, and AUC), the ability to distinguish classes, the training process, and the quality of generated features.

The remainder is organized as follows: Section 2 surveys studies related to faulty code prediction and the solutions to deal with data shortage and imbalance problems. The proposed model architecture and semi-supervised training strategy are described in Section 3. Section 4 presents the models' settings for conducting experiments. Section 5 analyzes and discusses the experimental results. Section 6 concludes the new findings in this work.

2. Related Work

2.1. Faulty code detection

Faulty code defection is a significant research field in software engineering [12]. The defect prediction literature can be divided into two approaches: First, by using Software Metrics, the measurable properties of the software system and second, by using fault data from a similar code snippet. In the first approach, authors focus on designing new discriminative features which can distinguish between types of faulty code. Authors in [13] proposed churn metrics and combined it with software dependencies for defect prediction. The efficiency of change metrics and static code attributes for defect prediction are analysed comprehensively in [14]. To select appropriate features, authors in [15] proposed to use Naive Bayesian method to filter redundant ones. These approaches is strongly depended on the designing metrics which may be redundant or not be highly correlated with class labels. Additionally, hand-crafted metrics cannot make full use of code context information to mine the syntactic structure and semantic information of code snippets. To overcome this obstacle, AST and CFG can be used to represent code snippets to preserve the syntactic structure and semantic information of code. Recently, machine learning techniques are being used in software fault prediction to assist testing and maintainability. In the literature, a machine learning model have been constructed in [16] which focused on Least square support vector machine with linear, polynomial and radial basis kernel functions. In [17], DBN is employed to generate hidden features, which contains syntaxes and semantics of programs and a classifiers is used to predict the faulty code by training on these hidden feature. Long short-term memory (LSTM) network is used in [18] to learn the representation of programs' ASTs to discover faulty code. [8] proposed a hybrid model between software metrics and the features learned by convolutional neural network (CNN). A novel graph convolutional network is designed in [7] to learn the semantic features of source code by learning on the program' CFGs. Applying deep learning models can automatically extract the discriminative features however, its drawback is that it requires a large amount of data while it is difficult in FCD problem. In the literature, few studies have focused on solving this issue.

2.2. Data limitation and Imbalance data

Deep artificial neural networks require a large corpus of training data in order to effectively learn. Limited training data results in a poor approximation. An over-constrained model will underfit the small training dataset, whereas an under-constrained model, in turn, will likely overfit the training data, both resulting in poor performance. Transfer learning and data augmentation are two commonly used techniques for dealing with data limitation. Transfer learning is used to improve a classifier from one domain by transferring information from a related domain [19]. There are many fields that transfer learning has been successfully applied to including computer vision [20], [21] and nature language processing with BERT [22], ULMFiT [23], ElMo [24] and GPT [25]. Data augmentation help to increase the amount of training data using information only in training data. In [26], authors present a general data augmentation strategy using Perlin noise, applying it to pixel-by-pixel image classification and quantification of various kinds of image patterns of diffuse interstitial lung disease (DILD). Authors in [27] propose a method to allow a neural net to learn augmentations that best improve the classifier, which authors call neural augmentation which proven the success on various datasets. In FCD problem, authors in [28] and [29] proposed to use transfer learning technique to deal with the data limitation issue.

Our proposed deep learning network differs from the aforementioned faulty code detection methods. It focus on dealing with data limitation in FCD problem by using a novel model architecture and new training strategy.

3. The Proposed Approach

This section presents our proposed method for faulty code prediction with limited data. The model accepts the inputs as assembly instruction sequences obtained by compiling the source files (written in C/C++ programming language). Fig. 2 illustrates an example of a C code and a snippet of its corresponding assembly instructions. The network architecture as shown in Fig. 1

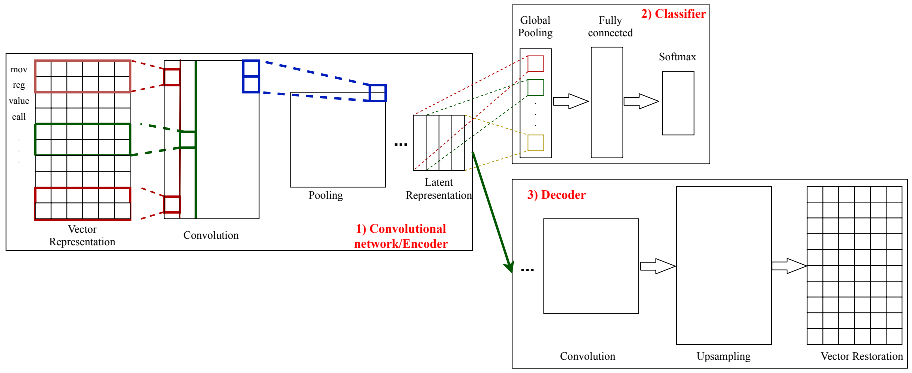


Figure 1: The model architecture.

consisting of three main parts (1) a convolutional network, (2) a classifier, and (3) a decoder. Assembled parts of (1)−(2) and (1)−(3) create two sub-networks to undertake different tasks including classification and an autoencoder for input reconstruction. The rest of this section will describe model details.

3.1. The convolutional network

Assembly code is selected as the input of the model. Based on previous studies, assembly instructions are more suitable for software fault prediction than other representations like software metrics and abstract syntax trees (ASTs) [9]. Basically, software metrics simply capture surface features of programs, and ASTs just represent the program syntactic structures. While semantic bugs are more relevant to program behaviors than the structures [29]. For this criterion, assembly code should be a selection because assembly instructions are translated into the machine code one by one. To feed into the network, each instruction is encoded as a real-valued vector with thirty dimensions. To learn the vector representations, we collected assembly instruction sequences from all experimental data and trained by the GloVe algorithm [30].

Vector representation is the first layer of the networks. After obtaining instruction vectors, we form the lookup table $E = e_1 \oplus e_2 \oplus ... \oplus e_N \in \mathbb{R}^{N \times d}$,

```c
#include <stdio.h>
int gcd(int a , int b)
{
 if(b == 0)
  return a;
 return gcd(b , a%b);
}

int main(void) {
 int t;
 scanf("%d",&t);
 while(t--)
 {
  int a , b;
  scanf("%d %d", &a ,&b);
  int  g = gcd(a,b);
  int l = (a*b)/g;
  printf("%d %d\n",g,l);
 }
 return 0;
}
```

```
main:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    leaq    -12(%rbp), %rax
    movq    %rax, %rsi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    scanf
    jmp .L5
.L6:
    leaq    -20(%rbp), %rdx
    leaq    -16(%rbp), %rax
    movq    %rax, %rsi
    movl    $.LC1, %edi
    movl    $0, %eax
    call    scanf
    movl    -20(%rbp), %edx
```

Figure 2: A C code example and a snippet of its assembly instructions generated by GCC compiler.

where, N is the number of unique assembly instructions and d is the vector size; e_i is the vector of i^{th} instruction and $\oplus$ is the concatenation operation. Given an input program with instruction sequences $a_1, a_2, ..., a_L$, the embedding matrix is $E_p = e_{idx(a_1)} \oplus e_{idx(a_2)} \oplus ... \oplus e_{idx(a_L)} \in \mathbb{R}^{L \times d}$, in which $idx(a_i)$ is a mapping function returning the index of the assembly instruction a_i in the lookup table.

Convolutional layers undertake the most important role in the network. These layers apply a set of filters sliding over sequences to learn local features. Specifically, the filters with sizes of 2 and 3 will capture the relations among 2 and 3 instructions, respectively. Stacking multiple convolutional layers make the network able to learn high-level abstract features and expand the local regions for feature extraction, but more difficult to train. Formally, given a input program with the embedding matrix $E_p = e_1^p \oplus e_2^p \oplus ... \oplus e_L^p \in \mathbb{R}^{L \times d}$, a convolution

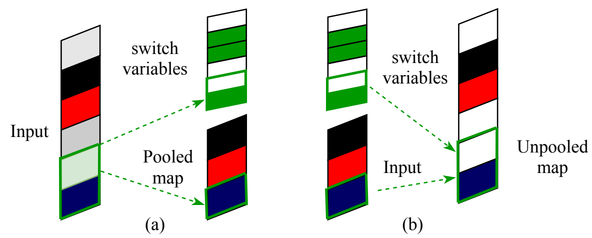


Figure 3: Illustration of pooling (a) and unpooling (b) processes.

will transform into a feature map with the same length $C^f = \{c^f_1, c^f_2, ..., c^f_L\}$:

$$c^f_i = f(W^f \cdot e^p{}_{i:i+h-1} + b_f) \tag{1}$$

where h is the filter size, $e^p_{i:i+h-1} = e^p_i \oplus e^p_{i+1} \oplus ... \oplus e^p_{i+h-1}$, $W_f \in \mathbb{R}^{h \times d}$, f is an non-linear activation function, and b_f is a bias. Normally, many filters are used in a convolutional layer to extract features according to different relation aspects.

A pooling layer is commonly stacked on each convolutional layer to perform dimension reduction. For high dimensional data, downsampling helps to reduce model parameters, and hence to speed up computational time and control overfitting. In the experimental datasets, a program may have up to 3,000 instructions resulting in a $3,000$ embedding matrix. In addition, the convolution just transforms the input data without downsampling. Therefore, applying pooling layers is essential.

In pooling layers, each feature map is resized independently. Normally, a pooling layer splits a feature map into non-overlapping regions and then applies the pooling operation for such sub-regions. Two commonly used operations are *max* and *average*. For example, a max pooling with the filter size of 2 and the stride of 2, the feature map column C^f will be separated into two-value regions, and the greater one of each pair is selected. This results in the pool feature map with size of $L/2$. In the training process, the backward pass simply routes the gradient to the highest value in the forward pass.

3.2. The classifier

For predicting faulty code, the classifier is built based on the features extracted by the convolutional network. This part includes a global pooling and some fully connected layers.

Global pooling. The last pooling layer in the part (1) is also a high dimensional features. To select the most sophisticated features, a global pooling is applied to pool each feature map into a single value.

Fully connected layers contains neurons that fully connect to all neurons in the previous layer. The activation of the last layer is $softmax$ to convert score values into distribution probabilities.

$$\sigma(\mathbf{z}_c) = \frac{e^{z_c}}{\sum_{i=1}^{K} e^{z_i}} \tag{2}$$

where K is the number of target labels, $c = 1, ..., K$, and $\mathbf{z} = (z_1, ..., z_K)$

We use categorical cross-entropy to evaluate the classification loss. For an instance i, the loss is computed as follows:

$$L(y_i, \hat{y_i}) = -\sum_{c=1}^{K} y_{i,c} log(p_{i,c}) \tag{3}$$

where $y_{i,c}$ is the indicator function that has value of 1 if and only if c is the correct target for the instance i. $p_{i,c}$ is the predicted probability for the class c as in Equation 2.

3.3. The autoencoder

An autoencoder consists of two segments, the encoder and the decoder (parts 1 and 3) that have symmetric structures. The encoder maps input sequences $\mathbf{X}$ by function ϕ to latent representations $\mathbf{F}$. Inversely, the decoder tries to recreate the original sequences X by function ψ. The process can be formulated as follows.

$$\begin{aligned} &\phi : \mathbf{X} \rightarrow \mathbf{F} \\ &\psi : \mathbf{F} \rightarrow \mathbf{X} \\ &\phi, \psi = \underset{\phi,\psi}{\operatorname{argmin}} \left\| \mathbf{X} - (\phi \circ \psi)\mathbf{X} \ \right\|^2 \end{aligned} \tag{4}$$

In the model, both the encoder and the decoder are built up from convolutional layers. A slight difference between the two architectures is that the pooling is used for dimension reduction in the encoder, while the decoder applies an upsampling to expand feature maps.

The objective of training the autoencoder is to minimize the reconstruction errors. The loss function for an instance x_i is as follows.

$$\begin{aligned} L(x_i, x_i{}') &= \|x_i - x_i{}'\|^2 \\ &= \|x_i - \phi(\psi(x_i))\|^2 \end{aligned} \tag{5}$$

where $x_i{}'$ is the reconstruction that has the same shape as x_i.

3.4. Loss function

The two branches shares the first part and are jointly trained. Therefore, the loss function of the whole model is the combination of classification and reconstruction errors. From equations 3 and 5, training process will minimize the following function.

$$\begin{aligned} L_i &= L(y_i, \hat{y_i}) + L(x_i, x_i{}') \\ &= -\sum_{c=1}^{K} y_{i,c} log(p_{i,c}) + \|x_i - \phi(\psi(x_i))\|^2 \end{aligned} \tag{6}$$

4. Experiments

4.1. Datasets

To verify the model in terms of performance and the ability to learn with limited data, we used the datasets as in [7] to conduct experiments. Each dataset contains all source files written in C/C++ for solving a problem on CodeChef[3], a contest site to learn programming languages. The collected problems are 1) SUMTRIAN for computing the sum of the numbers on the longest path in a triangular matrix, 2) FLOW016 for finding the greatest common divisor (GCD)

[3] `https://www.codechef.com/problems/<problem-name>`

Table 1: The statistics on the number of samples for the datasets

Dataset	Total	Class 0	Class 1	Class 2	Class 3	Class 4
FLOW016	10,648	3,472	4,165	231	2,368	412
MNMX	8,745	5,157	3,073	189	113	213
SUBINC	6,484	3,263	2,685	206	98	232
SUMTRIAN	21,187	9,132	6,948	419	2,701	1,987

and the least common multiple (LCM) of two integer numbers, 3) MNMX for finding the minimum cost to convert the array into a single element by several operations, 4) SUBINC for counting the non-decreasing subarrays in an array.

According to the execution result, a program is assigned to one of categories including *0) clean code* if the output meets all the requirements, and defective code that result in *1) time limit exceeded* - the running time is greater than the time limit, *2) wrong answer* - the output does not match with the expected, *3) runtime error* - the execution process was interrupted, and *4) syntax error* - the compilation was failed.

Table 1 shows data statistics on the experimental datasets. As can be seen, all the datasets are unbalanced in which the instance numbers of classes 0 and 1 dominate those the other classes. We used the same data split as in [7] with the training, validation and testing ratio of 3:1:1.

4.2. Baselines and training scenarios

Baselines. We compare our proposed method with a convolutional neural network (CNN) (as the classification branch without the autoencoder), and the CNN with transfer learning. As confirmed in many studies [7, 31, 9], assembly code-based methods reach much higher performance than those based on software metrics and ASTs on these datasets. In addition, transfer learning has proved as a good solution for limited data in various areas such as image, speech, and language processing [19]. Thus, two baselines including the CNN trained from scratch and the CNN with transfer learning are selected to verify the proposed model performance.

Training scenarios. To assess the influence of data amount on the learning ability, we trained the models with 100%, 75%, 50%, and 25% of the training set and ran prediction for the whole validation and test sets.

To judge the contribution of the autoencoder, the proposed model is trained with three scenarios as follows.

- AECNN: We first train the autoencoder and then remain the weights for the part 1 of convolutions to train CNN independently. This can be considered as a type of self-transfer learning because we initialize the weights using the same data but without target label information.
- CNNMul: the autoencoder and CNN are trained simultaneously from scratch.
- AECNNMul: The autoencoder is pretrained and then we continue training the whole model.

For transfer learning, we merge all data of the other problems to pretrain the CNN and then reuse the network weights as the initialization to build the classifier for the current problem.

4.3. Evaluation Measures

We used three common used measures including accuracy, F1 and AUC (the area under the receiver operating characteristic ROC) to evaluate model performance. The accuracy, and F1 can be computed from the confusion matrix constructed from the prediction results. For binary classification where a sample belongs one of two categories +1 (positive) and -1 (negative), the confusion matrix is as follows.

Classification accuracy:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

The F1 measure:

$$F1 = \frac{2 * Recall * Precision}{Recall + Precision} \tag{8}$$

Table 2: The confusion matrix

		Predicted	
		+1 (Positive)	-1 (Negative)
Actual	+1	TP	FN
	-1	FP	TN

where $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$

AUC is an essential measure to evaluate classifiers, especially in cases of unbalance data. AUC is the area under the ROC curve that is generated by plotting all points of (true positive rate $TPR = \frac{TP}{P}$, the false positive rate $FPR = \frac{FP}{N}$) at different discrimination thresholds. The AUC shows the ability to distinguish between classes.

The above measures were presented for binary classifiers. To extend to multi-class problems, the measures are computed by several ways:

- micro. computing the TPR, and FPR, FP and TP globally.
- macro. computing the metrics for each class and take the unweighted mean.
- weighted. Similar to the macro, but taking the weighted mean according to the number of instances for each class.

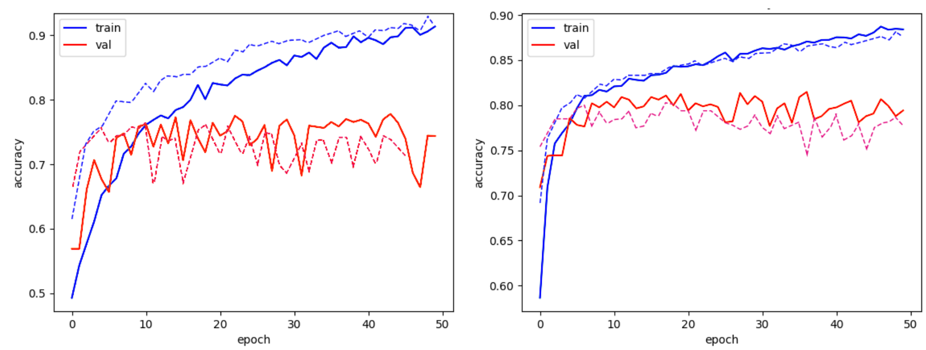
5. Result Analysis and Discussion

From Table 3, it should be confirmed that finding solutions to deal with the data shortage is critical. Indeed, the CNN downgrades significantly when we reduce the amount of training data from 100% to 25%. Specifically, the AUC of CNN decreases 5%, 9%, %6, and 2% on FLOW016, MNMX, SUBINC, and SUMTRIAN, respectively. This issue also happens in the cases of Accuracy and F1. Unlikely, transferable models work more stable with less data. Taking the 25% data as an example, the pretraining the autoencoder in the multitask model (AE_CNN_Mul) achieves the highest performance on all the datasets.

Table 3: The comparison of the methods according to Accuracy, F1, and AUC. Each row block shows the classification performance for the use of 100%, 75%, 50%, and 25% of labelled training data. Each cell is in the form of mean ± standard deviation for 5 times of running.

Approach	FLOW016			MNMX			SUBINC			SUMTRIAN		
	Acc.	F1	AUC	Acc.	F1	AUC	Acc.	F1	AUC	Acc.	F1	AUC
100												
CNN	72.52±0.23	71.34±0.15	0.80±0.08	81.99±0.06	80.03±0.21	0.80±0.02	70.93±0.71	69.94±0.32	0.72±0.11	67.97±0.34	66.35±0.42	0.77±0.19
CNN_Trans	73.05±0.54	71.67±0.43	0.81±0.34	81.53±0.31	79.07±0.55	0.80±0.11	69.08±0.92	68.15±0.81	0.74±0.52	67.12±0.73	66.06±0.56	0.79±0.11
AE_CNN	**74.83±0.25**	72.92±0.21	0.81±0.10	**83.43±0.31**	**81.44±0.37**	**0.81±0.28**	72.59±0.66	**71.62±0.28**	0.73±0.38	**68.09±0.87**	**67.33±0.51**	0.78±0.21
CNN_Mul	73.90±0.32	72.55±0.33	0.80±0.12	82.11±0.29	81.05±0.08	0.80±0.12	71.47±0.82	70.32±0.41	0.73±0.41	66.66±0.63	65.69±0.66	0.78±0.58
AE_CNN_Mul	73.96±0.18	**73.06±0.11**	**0.82±0.07**	83.06±0.23	81.12±0.10	0.80±0.03	**72.63±0.59**	71.40±0.25	**0.74±0.22**	67.53±0.81	66.54±0.47	**0.79±0.17**
75												
CNN	71.95±0.33	70.40±0.31	0.78±0.53	79.96±0.23	78.62±0.73	0.78±0.70	70.62±0.55	69.99±0.52	0.72±0.11	65.99±0.48	64.9±0.33	0.78±0.22
CNN_Trans	72.91±0.41	71.73±0.39	0.81±0.27	79.04±0.62	77.52±0.24	0.76±0.22	67.77±0.67	66.24±0.66	0.70±0.23	66.89±0.24	66.20±0.19	0.78±0.30
AE_CNN	72.82±0.31	71.86±0.21	0.80±0.10	**81.41±0.29**	**78.89±0.58**	0.78±0.08	71.55±0.49	70.14±0.43	**0.74±0.10**	66.07±0.67	64.34±0.41	0.78±0.32
CNN_Mul	72.79±0.41	71.19±0.42	0.79±0.09	80.11±0.58	78.12±0.65	0.78±0.82	71.16±0.50	69.59±0.45	0.72±0.32	65.43±0.42	63.84±0.23	0.78±0.15
AE_CNN_Mul	**73.74±0.21**	**72.78±0.11**	**0.81±0.12**	81.02±0.56	78.80±0.71	**0.79±0.12**	**71.67±0.38**	**70.44±0.25**	0.72±0.26	**67.77±0.72**	**66.18±0.37**	**0.79±0.09**
50												
CNN	70.08±0.92	70.12±0.65	0.77±0.16	78.80±0.11	77.22±0.81	**0.77±0.62**	68.79±0.87	67.17±0.33	**0.70±0.34**	64.32±0.15	62.78±0.32	0.76±0.17
CNN_Trans	72.53±0.82	71.22±0.77	0.79±0.20	77.84±0.79	76.54±0.55	0.76±0.78	65.76±0.71	64.92±0.67	0.68±0.22	65.77±0.81	64.14±0.72	0.77±0.53
AE_CNN	71.98±0.47	71.02±0.31	0.78±0.21	**80.11±0.77**	**78.09±0.68**	**0.77±0.54**	**70.16±0.20**	67.87±0.45	**0.70±0.15**	**66.21±0.46**	**64.75±0.23**	**0.78±0.21**
CNN_Mul	70.29±0.31	70.56±0.50	0.78±0.32	78.99±0.18	76.12±0.82	0.76±0.81	68.02±0.23	66.92±0.24	0.69±0.36	64.18±0.51	61.78±0.40	0.76±0.19
AE_CNN_Mul	**72.69±0.52**	**71.19±1.22**	**0.80±0.01**	79.55±0.61	77.80±0.77	**0.77±0.15**	69.25±0.81	**68.84±0.44**	**0.70±0.39**	64.98±0.74	63.8±0.88	0.77±0.39
25												
CNN	68.98±0.61	67.82±0.33	0.75±0.16	77.56±0.73	76.32±0.78	0.71±0.21	65.54±0.16	63.06±0.46	0.66±0.27	62.12±0.71	60.07±0.66	0.74±0.26
CNN_Trans	69.11±0.22	68.12±0.15	**0.77±0.52**	76.10±.082	75.08±0.66	0.72±0.15	62.07±0.52	59.58±0.46	0.64±0.25	**64.82±0.57**	**62.44±0.33**	**0.76±0.25**
AE_CNN	69.15±0.36	68.02±0.41	0.76±0.06	**79.24±0.81**	**76.59±0.65**	**0.73±0.11**	66.11±0.90	63.71±0.61	0.66±0.61	62.51±0.62	60.27±0.55	0.74±0.31
CNN_Mul	68.92±0.44	67.75±0.53	0.76±0.73	77.01±0.77	76.11±0.81	**0.73±0.72**	65.65±0.86	63.66±0.52	0.66±0.15	61.32±0.79	60.57±0.92	0.75±0.29
AE_CNN_Mul	**69.75±0.38**	**68.21±0.33**	**0.77±0.33**	78.08±0.18	76.12±0.66	**0.73±0.64**	**67.31±0.71**	**64.07±0.44**	**0.68±0.29**	64.75±0.53	62.20±0.51	**0.76±0.39**

It should be noted that self-transfer learning is beneficial to our model. In this technique, the autoencoder is pretrained without use of label information, then we continue training either the classification or both of two tasks. The classifiers with pretrained autoencoder (AE_CNN and AE_CNN_Mul) completely dominate that trained from scratch (CNN_Mul) according to all metrics. Generally, the model with pretrained autoencoder can enhance the performance from 1% to 2%. In terms of dealing with limited data, our proposed model outperforms transfer learning. The AE_CNN_Mul model achieves higher performance measures like Accuracy, F1 and AUC than those of CNN_Trans on almost datasets at any ratio of labelled data. The CNN_Trans just slightly overcomes our model on SUMTRIAN and FLOW016 with 25% of data. Our method worked well because it can addresses the limitations of transfer learning. Transfer learning requires several conditions including 1) trained weights for the initialization in the fine-tuned process should be out of local minima of the loss

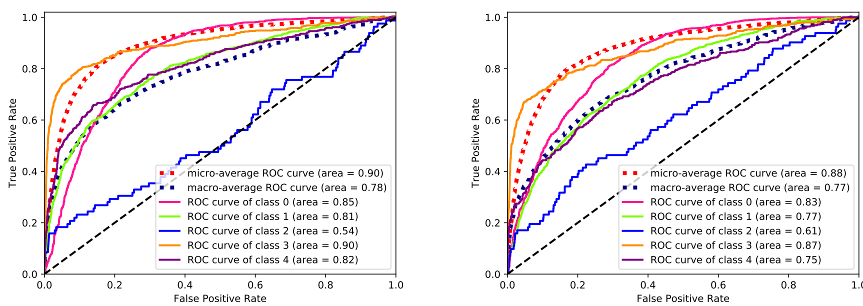


(a) (b)

Figure 4: The training process of AE_CNN_Mul (solid lines) and CNN_Trans (dashed lines) classifiers on MNMX with 25% (4a) and 100% (4b) data.

function [32], 2) data distributions of the source and target should be relevant. These issues have appeared in the experiments. After observing training processes that are illustrated as in Fig. 4, we have found that the transfer learning has a good starting, but it rapidly reaches the top, varies around this point, and tends to over-fitting. Unlikely, our method begins with the lower accuracy, improves by epochs and finally achieves higher accuracy. Two keys lead to the proposed model's success are 1) applying self-transfer learning by pretraining the auto-encoder without label information will avoid the distribution difference between the source and the target, and 2) with multitask learning, the model must balance between two objectives,thus preventing overfitting.

With limited data, our model maintains the ability to distinguish among faulty types that is important for unbalanced data problems. For these problems, detecting minority samples are more important than that of majority samples, e.g. faulty code versus clean code. As can be seen in Table 3, our proposed classifiers obtains higher AUCs than other classifiers on all datasets with different labelled data ratios. Specifically, three our classifiers including AE_CNN, CNN_Mul, and AE_CNN_Mul have similar AUCs, and beats the CNN from 1-2%. We also analyze ROC curves of classifiers to validate the AUC performance.



(a) (b)

Figure 5: The ROC curves on the SUMTRIAN test set of the CNN trained with 100% data (Fig. 5a) and AE_CNN_Mul trained with 25% data (Fig. 5b).

Fig. 5 depicts ROC curves on SUMTRIAN of two classifiers. Interestingly, just the use of 25% labelled data, the AE_CNN_Mul can be competitive with CNN trained on 100% data according average AUCs. In addition, it improves 7% (54% to 61%) of AUC for the minority class (class 2).

The autoencoder can assist in learning sophisticated features. Fig. 6 visualizes the features resulting by the global pooling of the CNN and three training scenarios of our network. In general, the samples of the majority classes (0, 1, 3) are in separable clusters, and minority classes (2, 4) distribute sparsely. Our proposed model with pretrained autoencoder (Figures 6b and 6d) reduce the diversity among classes notably. This outcome is due to the support of the autoencoder task in the feature generation. In the CNN, learning features is towards one objective of classification labels, thus biasing to majorities. Unlikely, our multitask model focuses on two tasks of classification and input reconstruction. In which, the autoencoder facilitates generating sophisticated features, because the reconstruction process guarantees that such features must remain the semantic meanings of the inputs.

Based on above analysis we can see that the multitask learning make the model able to learn with limited data and the autoencoder can support to learn high-quality features. These have been proved by many evaluations such as 1)

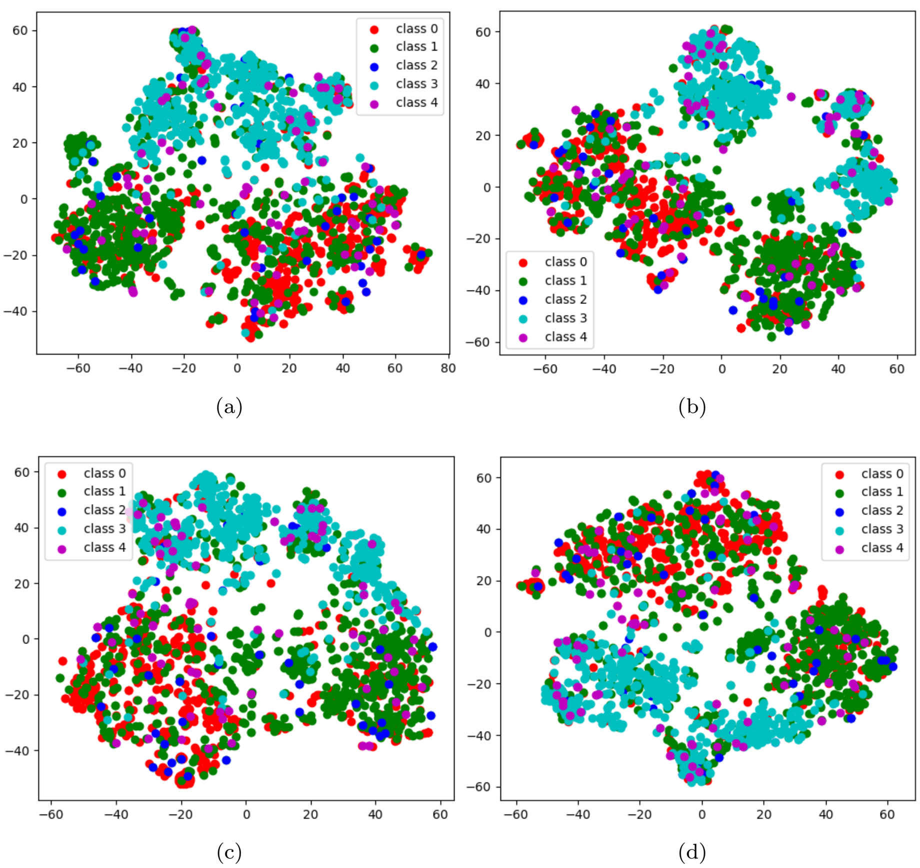


Figure 6: Feature visualization on the FLOW016 test set of methods CNN (6a), AE_CNN (6b), CNN_Mul (6c), and AE_CNN_Mul (6d)

classification metrics of Accuracy, F1, AUC 2) the ability to detect the minor but important classes, 3) the training process, and 4) sample distributions on the feature space.

6. Conclusion

In this paper, we designed a multitask learning neural network for solving the data shortage problem in faulty code prediction. The network includes two branches to undertake different tasks of classification by convolutions and input reconstruction by an autoencoder. We also proposed the self-transfer learning in which the autoencoder is pretrained without label information. The experimental results have shown that our model outperforms the others notably. Moreover, it is able to learn with a small amount of labelled data, but remains the prediction performance.

Acknowledgements

This research is funded by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.05-2018.306.

References

References

[1] I. H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Information and Software Technology 58 (2015) 388–402.

[2] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (6) (1994) 476–493.

[3] M. Lorenz, J. Kidd, Object-oriented software metrics: a practical guide, Prentice-Hall, Inc., 1994.

[4] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, T. Love, Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics, IEEE Transactions on software engineering (2) (1979) 96–104.

[5] A. V. Phan, M. Le Nguyen, L. T. Bui, Sibstcnn and tbcnn+ knn-ted: New models over tree structures for source code classification, in: International Conference on Intelligent Data Engineering and Automated Learning, Springer, 2017, pp. 120–128.

[6] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, in: Thirtieth AAAI Conference on Artificial Intelligence, 2016.

[7] A. V. Phan, M. Le Nguyen, L. T. Bui, Convolutional neural networks over control flow graphs for software defect prediction, in: 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), IEEE, 2017, pp. 45–52.

[8] J. Li, P. He, J. Zhu, M. R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2017, pp. 318–328.

[9] A. V. Phan, M. Le Nguyen, Convolutional neural networks on assembly code for predicting software defects, in: 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), IEEE, 2017, pp. 37–42.

[10] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases., School of Information Technology and Engineering, University of Ottawa, Canada (2005).
URL `http://promise.site.uottawa.ca/SERepository`

[11] F. Wu, X.-Y. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, Y. Sun, Cross-project and within-project semisupervised software defect prediction: A unified approach, IEEE Transactions on Reliability 67 (2) (2018) 581–597.

[12] L. L. Minku, E. Mendes, B. Turhan, Data mining for software engineering and humans in the loop, Progress in Artificial Intelligence 5 (4) (2016) 307–314.

[13] N. Nagappan, T. Ball, Using software dependencies and churn metrics to predict field failures: An empirical case study, in: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE, 2007, pp. 364–373.

[14] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 30th international conference on Software engineering, 2008, pp. 181–190.

[15] Ö. F. Arar, K. Ayan, A feature dependent naive bayes approach and its application to the software defect prediction problem, Applied Soft Computing 59 (2017) 197–209.

[16] L. Kumar, S. K. Sripada, A. Sureka, S. K. Rath, Effective fault prediction model developed using least square support vector machine (lssvm), Journal of Systems and Software 137 (2018) 686–712.

[17] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 297–308.

[18] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, P. Montague, Cross-project transfer representation learning for vulnerable function discovery, IEEE Transactions on Industrial Informatics 14 (7) (2018) 3289–3297.

[19] K. Weiss, T. M. Khoshgoftaar, D. Wang, A survey of transfer learning, Journal of Big data 3 (1) (2016) 9.

[20] C. Wang, S. Mahadevan, Heterogeneous domain adaptation using manifold alignment, in: Twenty-second international joint conference on artificial intelligence, 2011.

[21] Y. Zhu, Y. Chen, Z. Lu, S. J. Pan, G.-R. Xue, Y. Yu, Q. Yang, Heterogeneous transfer learning for image classification, in: Twenty-Fifth AAAI Conference on Artificial Intelligence, 2011.

[22] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805.

[23] J. Howard, S. Ruder, Universal language model fine-tuning for text classification, arXiv preprint arXiv:1801.06146.

[24] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, L. Zettlemoyer, Deep contextualized word representations, arXiv preprint arXiv:1802.05365.

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, OpenAI Blog 1 (8) (2019) 9.

[26] H.-J. Bae, C.-W. Kim, N. Kim, B. Park, N. Kim, J. B. Seo, S. M. Lee, A perlin noise-based augmentation strategy for deep learning with small data samples of hrct images, Scientific reports 8 (1) (2018) 1–7.

[27] L. Perez, J. Wang, The effectiveness of data augmentation in image classification using deep learning, arXiv preprint arXiv:1712.04621.

[28] J. Nam, W. Fu, S. Kim, T. Menzies, L. Tan, Heterogeneous defect prediction, IEEE Transactions on Software Engineering 44 (9) (2017) 874–896.

[29] A. P. Viet, K. D. T. Nguyen, L. V. Pham, Transfer learning for predicting software faults, in: 2019 11th International Conference on Knowledge and Systems Engineering (KSE), IEEE, 2019, pp. 1–6.

[30] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543.
URL http://www.aclweb.org/anthology/D14-1162

[31] A. V. Phan, M. Le Nguyen, Y. L. H. Nguyen, L. T. Bui, Dgcnn: A convolutional neural network over large-scale labeled graphs, Neural Networks 108 (2018) 533–543.

[32] B. Tan, Y. Song, E. Zhong, Q. Yang, Transitive transfer learning, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2015, pp. 1155–1164.