Received February 17, 2020, accepted March 15, 2020, date of publication March 19, 2020, date of current version March 30, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2981869*

# An Adversarial Discriminative Convolutional Neural Network for Cross-Project Defect Prediction

**LEI SHENG, LU LU, AND JUNHAO LIN**
School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China

Corresponding author: Lu Lu (lul@scut.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61370103, in part by the Guangzhou Produce and Research Fund under Grant 201902020004, and in part by the Fundamental Research Funds for the Central Universities.

**ABSTRACT** Cross-project defect prediction (CPDP) is a promising approach to help to allocate testing efforts efficiently and guarantee software reliability in the early software lifecycle. A CPDP method usually trains a software defect classifier based on labeled data sets. Then the trained classifier can predict new projects without labeled data. Most previous CPDP techniques focused on manually designing handcrafted features. However, these handcrafted features ignore the programs' semantic information. Moreover, some other existing defect prediction approaches learned semantic features from source code to build classifiers directly. However, they did not consider the distribution divergence between source and target projects. To address these limitations, we put forward a new method called Adversarial Discriminative Convolutional Neural Network (ADCNN). It can extract the transferable semantic features from source code for CPDP tasks. Specifically, we first parse source files into token vectors and then map them to integer vectors via word embedding. Second, we combine adversarial learning with discriminative feature learning to train the ADCNN model. The key of the ADCNN model is to learn the discriminative mapping of the target project to the source feature space by deceiving a domain discriminator. A domain discriminator tries to distinguish the target project files from the source project files. Finally, we use the extracted transferable semantic features to build a classifier for CPDP tasks. We evaluate our method on ten benchmark projects in terms of F-measure, AUC, and PofB20 (an effort-aware evaluation metric). The experimental results demonstrate that our ADCNN method performs better compared with other related CPDP methods.

**INDEX TERMS** Cross-project defect prediction, transfer learning, adversarial learning, deep learning, convolutional neural network.

## I. INTRODUCTION

With the increase of software scale and complexity, software reliability assurance becomes more difficult and vital. Software testing is an essential means of reliability assurance. However, it is impractical for testers to test all code units. Software defect prediction (SDP) could help to find the defect-prone modules or files by analyzing the characteristics of static code. With the help of SDP, the software testing team could allocate resources more efficiently [25].

SDP usually uses machine learning to train prediction models [5], [21] based on historical data (e.g., source code edit logs [27]). Depending on whether the source project and the target project are from the same project, SDP can be divided into Within-Project Defect Prediction (WPDP) [19], [34], [35], [44] and Cross-Project Defect Prediction (CPDP) [32]. In the early stages of a project, it is difficult for WPDP to build a feasible predictive model due to the lack of labeled file information. Unlike WPDP, CPDP uses sufficient labeled file information in mature projects to train the defect predictors, which can predict whether the new project files are defective. The CPDP method is a dominant application of transfer learning [30]. In this paper, we focus on researching CPDP.

The CPDP methods usually contain two phases: extracting features from source files and building machine learning classifiers. Some prior researches have focused on manually designing new or different features and combining

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

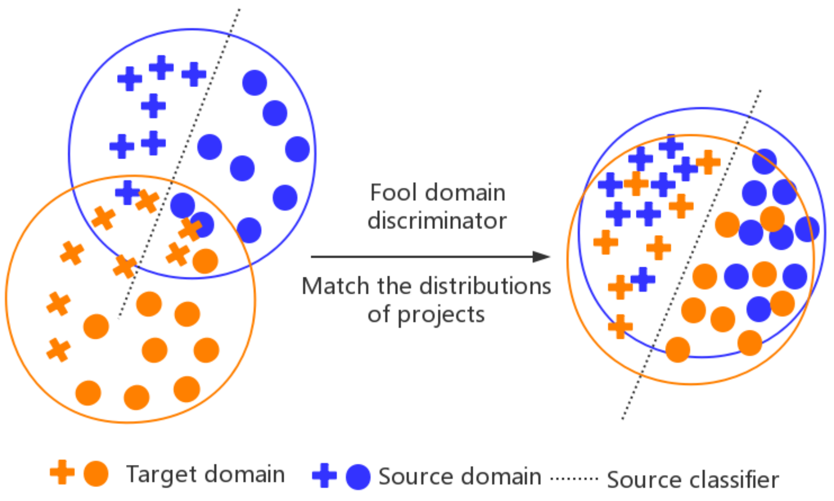

Fool domain discriminator
Match the distributions of projects
Target domain
Source domain
Source classifier

**FIGURE 1. Left: Different projects have different distributions. The classifiers trained by source projects could not necessarily be adapted to target projects. Right: When matched the distributions, the cross-project defect predictors would be more transferable to other projects.**

certain features, such as Halstead features [23], MaCabe features [24], and CK features [7]. Using these features, researchers built feasible models to distinguish between defective files and non-defective files effectively. Nevertheless, the manually extracted features ignore the rich semantic features of the program. Hence, in recent years, many studies [8], [20], [33], [41] have proposed to extract the abstract features hidden in semantics through deep learning (DL). Furthermore, they demonstrated that the DL-based methods could improve prediction performance. Generally speaking, the DL-based methods extract features from the program's abstract syntax tree (AST) by using deep learning networks. Inspired by this, we utilize convolutional neural networks as the feature extractors. However, because different projects have different code characteristics (e.g., code scales, coding styles, etc.), different projects have significantly different distributions [13]. As shown in Figure 1 (left), the classifiers trained by source projects could not be adapted to target projects because of the different distributions. After matching the distributions of projects via a transfer learning technique, the transferable data representations would be applied to train a transferable predictor across projects (Figure 1(right)).

In this paper, we propose a new model called Adversarial Discriminative Convolutional Neural Network (ADCNN). We aim to eliminate the evil influence of the differences in the distribution of semantic features between the source and target projects and learn the extracted transferable features for CPDP missions. First, we parse the source code of each file into a numerical vector as the input of our model. Second, we introduce adversarial discriminative learning to minimize the distance between the source mapping distribution and the target mapping distribution through two independent training stages. In the first stage, we only train the source encoder and source classifier using the labeled source data. Next, we train the target encoder to make the target data representation similar to the source data representation, by fooling the discriminator. By reducing distribution differences between projects, our ADCNN can learn the transferable semantic features. Finally, we feed the features generated from both source and target projects into a Logistic Regression (LR) classifier to conduct CPDP. The following are the main contributions of our paper.

- In this paper, considering the data distribution divergence between projects and adopting the domain adaptation method, we put forward a new CPDP model, named ADCNN. The key is that ADCNN uses adversarial discriminative learning to learn an asymmetric mapping, where it changes the target representation distribution to match the source distribution.
- We evaluate our ADCNN on ten benchmark projects in terms of F-measure, AUC, and PofB20 (an effort-aware evaluation metric). The experimental results prove that our ADCNN method could improve prediction performance compared with other related CPDP methods. [20], [29], [31], [38], [41].

We organize the remaining parts of the paper as follows. We introduce the related work of software defect prediction in Section II. We elaborate our proposed ADCNN in Section III. We show our settings of experiments and evaluated metrics in Section IV. Then we present our results in Section V. Next, we discuss the performance of our model under different parameter setting and threats to validity in Section VI. In Section VII, we summarize this paper and discuss possible future works.

## II. RELATED WORK

SDP has attracted the attention of a large number of researchers [2], [16], [19], [35], [36], [43]. In this section, we mainly introduce related work from three aspects: SDP using handcrafted features, SDP using DL-based semantic features, and adversarial learning.

### A. SDP USING HANDCRAFTED FEATURES

Based on manually extracted features (e.g., Halstead [23], McCabe [24], CK [7] features), many researchers have constructed predictive models for SDP, including WPDP and CPDP. WPDP has adopted many machine learning methods (e.g., Naive Bayes (NB) [2], Dictionary Learning [16], Support Vector Machine (SVM) [9], Decision Tree [40]). However, for a new project, it is challenging to obtain labeled historical data to build a useful WPDP model.

To address this limitation in WPDP, researchers employed the abundant labeled historical data in open source projects (e.g., PROMISE, AEEEM, and NASA) and proposed some CPDP methods [12], [18], [28], [38], [45], [46]. Zimmermann *et al.* [46] evaluated the performance of CPDP predictors on 12 open projects and the 622 combinations of them. The experimental results revealed that it was still a challenge for the defect prediction models to achieve cross-project prediction well. He *et al.* [12] provided lots of experiments on 34 datasets and proposed a CPDP method based on features selection with comparisons to WPDP approaches. They proved that it was feasible to find the best CPDP model on special projects. To enhance the CPDP performance, Turhan *et al.* [38] explored the feasibility of utilizing cross-project handcrafted features data to construct local defect predictors. They employed the NN filter to select cross-project data and then built CPDP predictors. However, the NN filter might remove some critical information for cross-project data during training. Transfer Component Analysis (TCA), which could discover a potential correlation of the source data and target data. Inspired by this, Nam *et al.* [28] proposed a state-of-the-art CPDP method, called TCA+, by optimizing TCA in the normalization process. Xia *et al.* [45] proposed HYDRA, which constructed a genetic algorithm (GA) classifiers with some target domain data firstly, and then leveraged ensemble learning (EL) to weigh each classifier. As a result, HYDRA could alleviate the effect of distribution differences between different projects. HYDRA was also a promising technique for CPDP. However, HYDRA required 5% of labeled test data. In practice, it was expensive for the developers to gain labeled test data by inspecting software projects manually.

### B. SDP USING DL-BASED SEMANTIC FEATURES

The approaches mentioned in Section 2.1 only use handcrafted features, which ignore the rich semantic features of the source code. To make full use of the potential semantic information in ASTs, DL has been adopted to generate features automatically.

For both WPDP and CPDP, Wang *et al.* [42] leveraged Deep Belief Network (DBN) to extract token vectors from programs' ASTs and then learn semantic and structural features automatically. It is worth mentioning that their evaluation of 10 source projects demonstrated that learned semantic features could significantly improve performance compared with handcrafted features. DL can also be applied to improve the performance of SDP. Li *et al.* [20] built a framework called Defect Prediction through Convolutional Neural Network (DP-CNN), which exploited the word embedding method and CNN to generate useful features. Besides, they combined generated features and existing traditional features to improve defect prediction further. Dam *et al.* [8] developed a prediction model, which used tree-structured LSTM (Tree-LSTM) to learn semantic and structural representations from source code. The experimental results on Samsung projects and the public PROMISE repository confirmed the effectiveness.

However, the above DL-based methods did not consider the features' distribution gap between the source and target domains. This common shortcoming would hinder the predictive performance.

### C. ADVERSARIAL LEARNING

Adversarial adaptation techniques have recently been used as a general tool to minimize the divergence between domains' distribution. The original Generative Adversarial Network (GAN) [10] simultaneously trains a generative network $G$ and a discriminative network $D$. $G$ produces a fake data distribution, and $D$ learns to represent the probability of an example coming from the real data distribution rather than the generator's distribution. During training, the networks simultaneously minimize the loss of $G$ and maximize the loss of $D$ (fooling the discriminator). The key is to ensure that the model cannot distinguish the distribution of training and test samples.

Based on the original GAN, many studies [1], [4], [22], [39] have designed different generative adversarial algorithms for different scenarios. Ajakan *et al.* [1] and Bousmalis *et al.* [4] applied adversarial learning to domain adaptation on the natural language process (NLP). They aimed at transferring the source domain's knowledge to the target domain. For domain adaptation, Liu *et al.* [22] proposed coupled generative adversarial networks (CoGANs), which trained two GANs to generate two domains' images, respectively. The framework could obtain a domain invariant feature space in both domains by binding the high-level parameters of two GANs. Tzeng *et al.* [39] attempted to build a general framework for cross-domain classification, called Adversarial Discriminative Domain Adaptation (ADDA). The approach was composed of untied weight sharing, discriminative model, and a GAN loss.

In this paper, we employ the adversarial training strategy to train our model. Therefore, our model can minimize the distribution differences between the source and target projects and extract the transferable structural and semantic features from the source code as effectively as possible.

## III. METHODOLOGY

In this section, we further extend the procedure proposed by [20] to learn deep semantic features from source code.

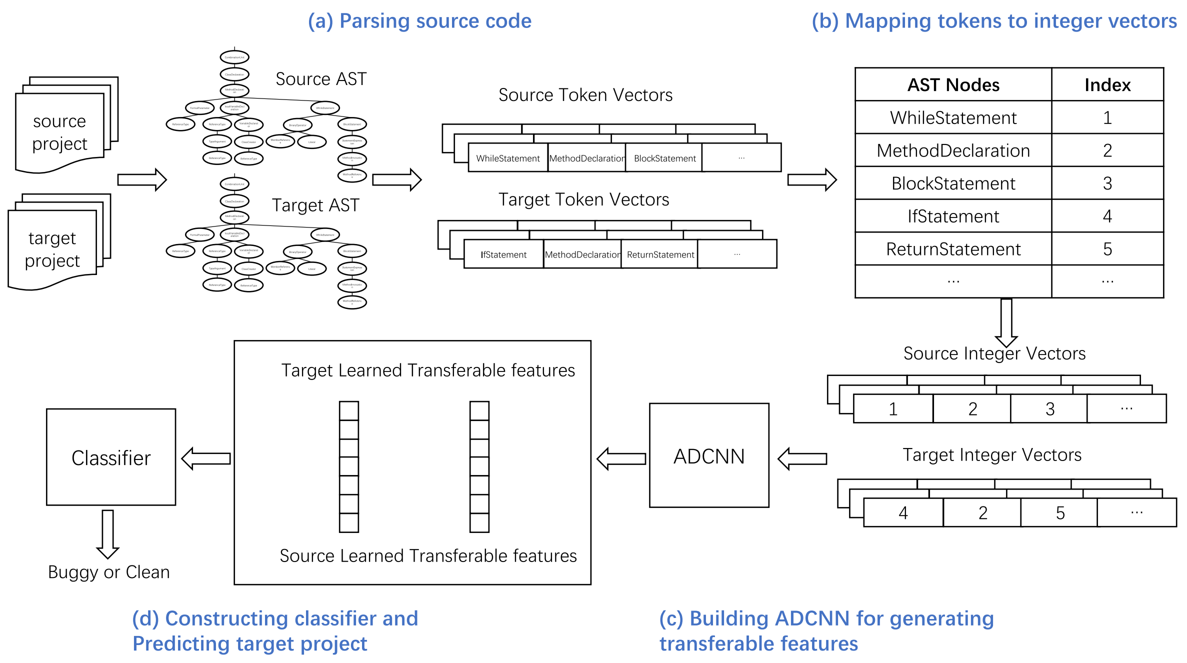


**FIGURE 2.** The overall framework of our proposed ADCNN.

**TABLE 1.** The selected AST node categories and types.

| Node Category | Node Type |
|---|---|
| Nodes of method invocations and instance creations | MethodInvocator, SuperMethodInvocator, ClassCreator |
| Declaration nodes | PackageDeclarator, InterfaceDeclarator, ClassDeclarator, ConstructorDeclaration, MethodDeclarator, VariableDeclarator, FormalParameter |
| Control-flow nodes | ForControl, IfStmt, ForStmt, WhileStmt, EnhancedForControl, DoStmt, AssertStmt, BreakStmt, ContinueStmt, ReturnStmt, ThrowStmt TryStmt, SynchronizedStmt, SwitchStmt, BlockStmt, CatchClauseParameter, TryResource, CatchClause, SwitchStatementCase |

Figure 2 illustrates the overall workflow of our proposed CPDP approach.

As shown in Figure 2, our approach contains four main steps: (1) parse source code into token vectors using ASTs, (2) map tokens to numerical vectors as the input of ADCNN, (3) build and train ADCNN, then generate the transferable semantic features, and (4) construct a LR classifier based on the learned features of both source and target projects for cross-project prediction.

## A. PARSING SOURCE CODE

In the first step, our model should parse source code to integer vectors. Previous works [20], [41] have demonstrated that DL networks can extract semantic features from ASTs. ASTs contain enough code structure information. Therefore, in our method, we utilize Javalang[1] to parse source code files from both source project (labeled) and target project (unlabeled) into ASTs and then generate token vectors. Similar to the previous method [20], we choose the three categories and types of AST nodes shown in Table 1: (1) nodes of method invocations and class instance creations. The methods' and classes' names in different projects usually are project-specific, which could not transfer to different projects. Different from [20], we use their node types to represent all methods and classes (e.g., MethodDeclarations and ClassInvocation). (2) declaration nodes, including declarations of type, method, and enum. (3) control-flow nodes (e.g., IfStmts, ForStmts, TryStmts, etc.). To highlight the importance of the above nodes, we exclude some other class-specific or method-specific node types, such as assignment. By doing this, we convert each file in projects into a corresponding token vector.

[1]https://github.com/c2nes/javalang

## B. MAPPING TOKENS AND HANDLING IMBALANCE

The input of our ADCNN model requires same-length numeric vectors. Firstly, we should establish a uniform mapping between node types and numbers and convert token vectors into integer vectors. The same number represents the same token, and the number ranges from 1 to the total number of node types. Specially, we filter out tokens less than three times, which might be file-specific. Therefore, the numeric vectors can preserve some structure information of files. Secondly, we pad all input vectors with zeros, making their length the same as the most extended vector. Finally, we employ a word embedding [26] layer to learn context-based feature vectors in the encoding phase. Tokens occurring in a similar context should be expressed as similar vectors and have closer distance in the representation space.

Besides, there are often more clean files than defective files in real-world software projects. The existence of class distribution skews will hinder the performance of our model. Generally speaking, undersampling and oversampling are two feasible sampling methods to solve the class imbalance problem. Undersampling balances the training datasets by removing some majority class instances (i.e., the clean instances). This kind of method might lose important information. Oversampling adds some minority class instances (i.e., the buggy instances) to balance the overall training datasets. After a comprehensive comparison, we utilize SMOTE [6] to handle imbalance. For a fair comparison, on the source projects, we will apply this process to both our proposed model and compared models.

### C. BUILDING ADCNN

The original GAN [10] is composed of two networks: a generative model $G$ and a discriminative model $D$, which are against each other. However, in ADCNN, the generative model is not necessary for the reason that the ultimate goal of our ADCNN is to learn discriminative representations. Moreover, in the field of CPDP, there is a natural generative and discriminative relationship between the source and target projects. Therefore, the implementation of our ADCNN is different from the origin GAN. As illustrated in Figure 3, building the ADCNN model includes two crucial procedures. Firstly, we train source encoder and classifier based on labeled source project datasets. Secondly, we train the target encoder and discriminator by using adversarial learning. The key is to minimize the approximate domain difference distance through an adversarial objective concerning a domain discriminator.

Referring to [20], CNN benefits from two characteristics: sparse connectivity and weight sharing. Consequently, CNN can catch the local structural and semantic information of a program to generate features. We use CNNs as our source and target encoders. In particular, the network architecture of our CNN encoder is the same as [20], including an embedding layer, a convolutional layer, a max-pooling layer, and a fully-connected hidden layer. Furthermore, our source classifier is composed of a fully-connected hidden layer and a single unit's output layer. Besides, the output of the fully-connected hidden layer in the CNN encoder is treated as the extracted features. The output layer employs the sigmoid as the activation function, while others apply ReLU as the activation function.

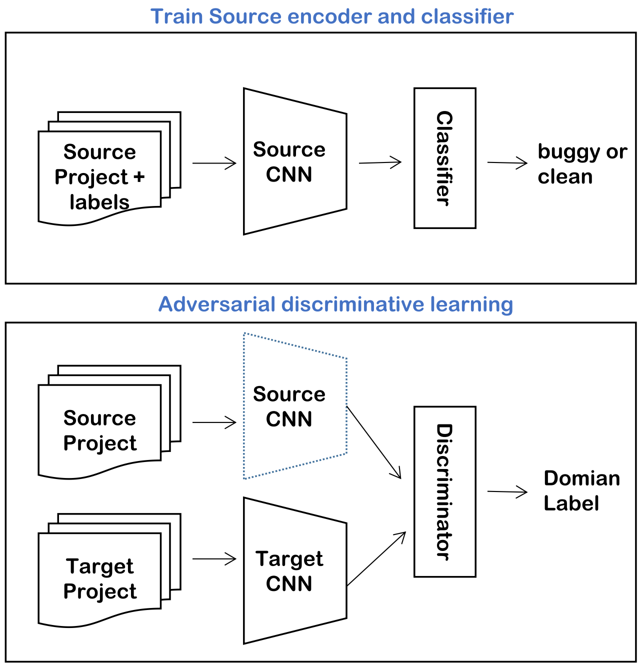


**FIGURE 3. An overview of the ADCNN training process. Firstly, we train the parameters of source encoder and source classifier based on source data (labeled). Secondly, we train the target encoder adversarially so that a discriminator cannot reliably predict the source and target instances to the right domain. In particular, we fix the parameters of source encoder during adversarial discriminative training. Dashed lines indicate frozen network parameters.**

In the CPDP field, we assume that the source project distribution $p_s(x, y)$ has both the source files $X_s$ and labels $Y_s$. In contrast, the target project distribution $p_t(x, y)$ only contains target files with no labels. So, we train the source encoder and source classifier $C$ by source files and labels. The source encoder learns a source representation mapping $M_s$, and the source classifier classifies a file is clean or buggy. During the training process, we utilize a minibatch stochastic gradient descent (SGD) algorithm [3] with Adam optimizer [17] to optimize the following objective optimization function.

$$\min_{M_s, C} L_c(\mathbf{X}_s, Y_s) = -\mathbb{E}_{(\mathbf{x}_s, y_s)\sim(\mathbf{X}_s, Y_s)} \sum_{k=1}^{K} y_s \log C(M_s(\mathbf{x}_s)) \tag{1}$$

Next, we aim at learning a target representation mapping $M_t$ and minimizing the distance between the source mapping distribution $M_t(X_t)$ and the target mapping distribution $M_s(X_s)$. However, the target project has no labeled data. When without suitable initial parameters, the target encoder may learn the wrong solution quickly during training. Hence, we copy the source encoder as the initialization of the target encoder. Also, we freeze the source encoder during adversarial training. We optimize the discriminator $D$ and the target encoder by two independent objective functions. The two objective optimization functions are formulated below:

$$\min_{D} L_{adv_D}(X_s, X_t, M_s, M_t) = -\mathbb{E}_{x_s \sim X_s}[\log D(M_s(x_s))] - \mathbb{E}_{x_t \sim X_t}[\log(1 - D(M_t(x_t)))] \tag{2}$$

$$\min_{M_s, M_t} L_{adv_M}(X_s, X_t, D) = -\mathbb{E}_{x_t \sim X_t}[\log D(M_t(x_t))] \tag{3}$$

By doing this, the model can effectively learn an asymmetric mapping, where it changes the target representation distribution so as to match the source distribution. Thus, the transferable semantic features can be extracted by the source encoder and the target encoder for CPDP.

### D. CROSS-PROJECT PREDICTION CONSTRUCTING

We process each file in both source and target projects to extract related transferable features by the above steps. Besides, we use the LR classifier as the base classifier and train the LR classifier by the extracted features of training files and their labels. Then, fixed the parameters of both ADCNN and LR classifier, we predict the buggy probability of each file in the target project.

## IV. EXPERIMENTS

In this section, we elaborate on the setup of the evaluation experiments. Moreover, we evaluate our approach compared with other recent related methods on ten open source projects.

### A. DATASETS

To examine the feasibility and effectiveness of our ADCNN model, we chose ten representative projects from the PROMISE repository as our experimental datasets. For SDP research, the PROMISE repository, a widely used public repository, provides the project version numbers, each file's class name, and corresponding defect label. According to the project names and versions, we could find the projects' source code from GitHub and then exploited them to build both our model and compared models. The selected projects have different sizes and overall buggy rates. Specifically, the maximum and the minimum number of source code files in the ten selected projects are 1077 and 32. Besides, the maximum and minimum overall buggy rate of projects is 6.3% and 98.8%, respectively. Table 2 shows the detailed descriptions of selected projects. Moreover, the PROMISE repository provides 20 traditional manual features. The traditional CPDP methods used the 20 traditional manual features to train models. Besides, other DL-based CPDP methods join them together with extracted semantic features to improve prediction performance. The specific meanings of the 20 traditional features are described in detail in [45].

To compose the CPDP tasks, we selected a project as the training set and another project from the rest projects as the test set, respectively. For example, we use forrest-0.8 as the training set and use poi-3.0 as the test set. Therefore, we could obtain 90 different pairs.

**TABLE 2.** The 10 selected projects.

| Project Name | Project Version | numbers of Instances | Buggy Rate |
|---|---|---|---|
| Forrest | 0.8 | 32 | 6.3% |
| Camel | 1.6 | 965 | 19.5% |
| Log4j | 1.2 | 205 | 92.2% |
| Ivy | 2.0 | 352 | 11.4% |
| Poi | 3.0 | 1077 | 63.6% |
| Lucene | 2.4 | 340 | 59.7% |
| Velocity | 1.6.1 | 229 | 34.1% |
| Synapse | 1.2 | 256 | 33.6% |
| Xerces | 1.4.4 | 588 | 74.3% |
| Xalan | 2.7 | 909 | 98.8% |

### B. EVALUATION METRICS

Regarding the selection of evaluation metrics, we considered the two following aspects: (1) non-effort-aware scenario and (2) effort-aware scenario.

#### 1) NON-EFFORT-AWARE EVALUATION METRICS

For non-effort-aware evaluation, we used the F-measure [11] and AUC. The two evaluation metrics have been used in many defect prediction approaches [8], [20], [41]. The following is the brief introduction of F-measure, AUC, recall, and prediction.

In this paper, the buggy files and clean files are defined as the positive instances and the negative instances, respectively. For the two-class classifier, we can get four different results: predicting a defective file successfully (real positive, RP); predicting a defective file unsuccessfully (fake negative, FN); predicting a clean file successfully (real negative, RF); predicting a clean file unsuccessfully (fake positive, FP). Table 3 shows the definition of confusion matrix.

**TABLE 3.** Confusion Matrix.

| | Real Positive | Real Negative |
|---|---|---|
| Predictive Positive | RP | FP |
| Predictive Negative | FN | RN |

With the help of confusion matrix, we define the following evaluation metrics.

$$\text{Precision} = \frac{RP}{RP + FP} \tag{4}$$

$$\text{Recall} = \frac{RP}{RP + FN} \tag{5}$$

To a certain extent, precision and recall can measure the quality of a defect prediction model. However, they perform poorly when the data set is not balanced. Besides, the SDP data sets are usually imbalance. F-measure and AUC are not influenced by the distribution of data sets.

The **F-measure** indicator combines the results of precision and recall. The F-measure is calculated as follows.

$$F1 = \frac{2 \cdot \text{ Precision } \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6}$$

According to the above formula, we know the result of F-measure falls in the interval of 0 to 1. The improvement

**TABLE 4.** The traditional compared SDP methods.

| Traditional SDP method name | Brief Introduction |
|---|---|
| LR | Linear Regression. Building a classifier only using handcrafted features |
| TCA [29] | A classic transferable domain adaptation method which tries to apply maximum mean discrepancy to reproduce kernel Hilbert space and then learns transferable components. |
| TCA+ [28] | The combination of TCA and customized normalization rules. |
| DG [31] | The concept of data gravitation is used to construct this prediction model. |
| NNFilter [38] | The key of this method is to apply principles of analogy-based learning to training set and make training set and test set similar. |

**TABLE 5.** The compared deep learning SDP methods.

| Deep learning SDP methods | Brief Introduction |
|---|---|
| DBN [42] | A standard DBN model with 10 hidden layers and 100 nodes in each hidden layer. |
| CNN [20] | A method using CNN as the semantic feature extractor. Same as [20], the batch size is 32, the epoch number is 15, the embedding dimension is 30, the number of hidden nodes is 100, the number of filters is 10, and the filter length is 5. |
| DPDBN [20] | A method using the concatenation of learned semantic features and handcrafted features as the input. |
| DPCNN [20] | A improved CNN method concatenates learned semantic features with handcrafted features. |

of F-measure value means the enhancement of SDP model performance. In other words, an SDP model with a higher F-measure value can better predict whether an unlabeled file is defective or not.

**AUC** (the area under ROC curve), which is a non-parameter method to evaluate model performance. The higher AUC, the better model performance. ROC is a 2-D curve with plots the Fake Positive Rate (FPR) on the x-axis and Real Positive Rate (RPR) on the y-axis. FPR and RPR are calculated as follows.

$$\mathrm{FPR} = \frac{FP}{FP + RN} \tag{7}$$

$$\mathrm{RPR} = \frac{RP}{RP + FP} \tag{8}$$

### 2) EFFORT-AWARE EVALUATION METRICS

In some scenarios where testing resources are limited or deadlines are approaching, we want to inspect a certain amount of code to find as many bugs as possible. At this time, we need to use another effort-aware evaluation metric to evaluate the sound quality of SDP methods. PofB20 [15] refers to the proportion of discovered bugs after reviewing the first 20% lines of code (LOC) of a whole engineering project. Besides, PofB20 has been employed by some SDP methods [41].

PofB20 can be calculated by the following steps. Firstly, we sort all test files in descending order according to the probability of being predicted as the defective. Then, we select a part of the files according to the above order. Their total LOC accounts for 20% of the total number of LOC in the test project. Finally, we accumulate the number of actual bugs (provided by the data set) of the above-selected files and calculate the percentage of bugs. The higher the PofB20 score, the more bugs can be detected by reviewing an equal number of LOC.

## C. BASELINE METHODS

To evaluate the performance of our ADCNN model, we compared with other defect prediction methods, including traditional SDP methods and DL-based SDP methods. In particular, we would solve three research questions (RQ) as follow:

- ***RQ1:*** *Do ADCNN outperform the traditional CPDP methods that use handcrafted features as the input?*
- ***RQ2:*** *Comparing with other state-of-the-art deep learning methods, do ADCNN perform better?*
- ***RQ3:*** *Comparing with other defect prediction methods, can ADCNN get better results at an acceptable time?*

Table 4 describes traditional CPDP methods using 20 static code features. Table 5 introduces DL-based CPDP methods using generated semantic features.

To exclude extra interference factors, we used LR to implement the base classifiers in all models. For consistency and simplicity, we used the implementation of *LR* in *sklearn.linear_model* (a machine learning library of python) with default parameter settings. Besides, we used the same parsed numeric vectors as the input of the DL-based CPDP methods. To avoid randomness, we ran ten times and recorded the average results (F-measure, AUC, and PofB20).

## V. RESULTS

In this section, we will answer the first two research questions raised in the previous section. Given a target project,

**TABLE 6.** The F-measure results of our ADCNN method and other compared methods (i.e., traditional CPDP methods using handcrafted features and DL-based CPDP methods). The bold values mean the better F-measures. The numbers in brackets are the standard deviation. The next-to-last line stands for the win/tie/loss numbers of ADCNN versus other methods and the last line represents the total average F-measure.

| Target Project | LR | NNFilter | DG | TCA | TCA+ | DBN | CNN | DPDBN | DPCNN | ADCNN |
|---|---|---|---|---|---|---|---|---|---|---|
| camel | 0.327 | 0.327 | 0.345 | 0.327 | 0.325 | 0.313 | 0.327 | 0.331 | 0.327 | **0.346(0.016)** |
| forrest | 0.183 | 0.161 | 0.123 | 0.114 | 0.127 | 0.127 | 0.136 | 0.162 | 0.157 | **0.187(0.050)** |
| ivy | **0.289** | 0.259 | 0.267 | 0.253 | 0.246 | 0.222 | 0.26 | 0.251 | 0.264 | 0.251(0.014) |
| log4j | 0.651 | 0.644 | 0.654 | 0.67 | 0.678 | 0.653 | 0.674 | 0.645 | 0.662 | **0.745(0.042)** |
| lucene | 0.597 | 0.592 | 0.634 | 0.614 | 0.543 | 0.577 | 0.633 | 0.604 | 0.632 | **0.65(0.022)** |
| poi | 0.636 | 0.621 | **0.69** | 0.598 | 0.6 | 0.604 | 0.658 | 0.624 | 0.65 | 0.669(0.035) |
| synapse | 0.51 | 0.51 | 0.537 | 0.516 | **0.542** | 0.435 | 0.491 | 0.487 | 0.49 | 0.504(0.025) |
| velocity | 0.513 | 0.492 | **0.519** | 0.486 | 0.304 | 0.45 | 0.506 | 0.475 | 0.507 | 0.511(0.024) |
| xalan | 0.61 | 0.609 | 0.653 | 0.651 | 0.671 | 0.656 | 0.668 | 0.642 | 0.668 | **0.743(0.030)** |
| xerces | 0.65 | 0.63 | 0.616 | 0.619 | 0.582 | 0.57 | 0.618 | 0.614 | 0.636 | **0.667(0.025)** |
| ADCNN: W/T/L | **7**/0/3 | **8**/0/2 | **6**/0/4 | **8**/0/2 | **9**/0/1 | **10**/0/0 | **9**/0/1 | **9**/1/0 | **9**/0/1 | |
| AVG | 0.4973 | 0.4845 | 0.5038 | 0.4848 | 0.4618 | 0.4607 | 0.4971 | 0.4835 | 0.4993 | **0.5273(0.029)** |

**TABLE 7.** The AUC results of our ADCNN method and other compared methods (i.e., traditional CPDP methods using handcrafted features and DL-based CPDP methods). The bold values mean the better AUC. The next-to-last line stands for the win/tie/loss numbers of ADCNN versus other methods and the last line represents the total average AUC.

| Target Project | LR | NNFilter | DG | TCA | TCA+ | DBN | CNN | DPDBN | DPCNN | ADCNN |
|---|---|---|---|---|---|---|---|---|---|---|
| camel | 0.543 | 0.542 | 0.563 | 0.539 | 0.538 | 0.527 | 0.542 | 0.547 | 0.543 | **0.589** |
| forrest | 0.68 | 0.617 | 0.53 | 0.5 | 0.561 | 0.544 | 0.564 | 0.618 | 0.618 | **0.691** |
| ivy | **0.64** | 0.596 | 0.621 | 0.609 | 0.607 | 0.554 | 0.61 | 0.595 | 0.614 | 0.617 |
| log4j | 0.506 | 0.517 | 0.493 | 0.436 | 0.441 | 0.454 | 0.438 | 0.489 | 0.442 | **0.597** |
| lucene | 0.565 | 0.554 | 0.6 | 0.565 | 0.55 | 0.623 | 0.575 | 0.559 | 0.58 | **0.615** |
| poi | 0.586 | 0.578 | **0.655** | 0.55 | 0.543 | 0.545 | 0.582 | 0.563 | 0.582 | 0.6 |
| synapse | 0.6 | 0.6 | 0.626 | 0.599 | **0.628** | 0.525 | 0.573 | 0.575 | 0.575 | 0.609 |
| velocity | 0.603 | 0.578 | 0.61 | 0.569 | 0.492 | 0.538 | 0.584 | 0.564 | 0.588 | **0.614** |
| xalan | 0.665 | 0.665 | **0.744** | 0.632 | 0.644 | 0.601 | 0.675 | 0.638 | 0.674 | 0.66 |
| xerces | 0.614 | 0.597 | 0.613 | 0.585 | 0.564 | 0.497 | 0.548 | 0.569 | 0.58 | **0.635** |
| ADCNN: W/T/L | **8**/0/2 | **9**/0/1 | **6**/0/4 | **10**/0/0 | **9**/0/1 | **9**/0/1 | **9**/0/1 | **10**/0/0 | **9**/0/1 | |
| AVG | 0.6034 | 0.5844 | 0.607 | 0.5584 | 0.5568 | 0.5408 | 0.5691 | 0.5717 | 0.5796 | **0.618** |

**TABLE 8.** The PofB20 results of our ADCNN method and other compared methods (i.e., traditional CPDP methods using handcrafted features and DL-based CPDP methods). The bold values mean the better PofB20. The next-to-last line stands for the win/tie/loss numbers of ADCNN versus other methods and the last line represents the total average PofB20.

| Target Project | LR | NNFilter | DG | TCA | TCA+ | DBN | CNN | DPDBN | DPCNN | ADCNN |
|---|---|---|---|---|---|---|---|---|---|---|
| camel | 20.6 | 20.78 | **22.22** | 17.38 | 17.64 | 18.1 | 17.77 | 20.5 | 20.4 | 21.63 |
| forrest | 18.52 | 11.11 | 7.41 | 25.93 | 25.93 | 23.7 | 17.78 | 20 | 19.26 | **37.41** |
| ivy | 25 | 25.2 | 19.44 | 22.62 | 20.83 | 23.29 | 21.15 | 23.77 | 22.74 | **23.81** |
| log4j | 8.59 | 8.46 | 9.06 | 11.85 | 10.41 | 14.05 | 11.2 | 8.89 | 10.97 | **15.94** |
| lucene | 17.56 | 17.74 | 16 | 20.02 | **21.08** | 16.79 | 14.77 | 16.07 | 15.52 | 19.6 |
| poi | 16.82 | 17.53 | 13.64 | 17.56 | 13.49 | 18.36 | 13.91 | 16.31 | 16.62 | **19.49** |
| synapse | 26.44 | **26.67** | 20.61 | 22.45 | 23.29 | 20.51 | 18.01 | 24.14 | 21.78 | 23.41 |
| velocity | 11.99 | 12.46 | 9.88 | 20.47 | **31.87** | 19.89 | 16.37 | 13.6 | 15.89 | 19.67 |
| xalan | 14.5 | 14.41 | 12.42 | 18.9 | 15.9 | 14.78 | 9.96 | 14.68 | 12.58 | **16.38** |
| xerces | 14.24 | 14.19 | 13.39 | 15.31 | 14.5 | 16.66 | 13.44 | 14.5 | 14.75 | **19.06** |
| ADCNN: W/T/L | **9**/0/1 | **8**/0/2 | **9**/0/1 | **8**/0/2 | **8**/0/2 | **9**/0/1 | **10**/0/0 | **9**/0/1 | **10**/0/0 | |
| AVG | 17.426 | 16.855 | 14.407 | 19.249 | 19.494 | 18.613 | 15.436 | 17.246 | 17.051 | **21.64** |

we selected the other nine projects as source project in turn. In this way, we could generate nine different combinations for a target project. Then we calculated the average F-measure, AUC, and PofB20 of the nine combinations and the win/tie/loss numbers of ADCNN versus other methods. Finally, we recorded all the experimental results. Table 6, Table 7, and Table 8 present the value of F-measure, AUC, and PofB20, respectively.

Besides, we made use of the Scott–Knott ESD test [37] to analyze the experimental results of the CPDP models. The Scott–Knott ESD test can help us to identify whether the prediction results of the CPDP methods are statistically significant. The Scott–Knott ESD test is a variant of the Scott–Knott test [14]. Like the Scott–Knott test, the primary role of the Scott–Knott ESD test is to use hierarchical cluster analysis to divide a series of methods into statistically different groups.

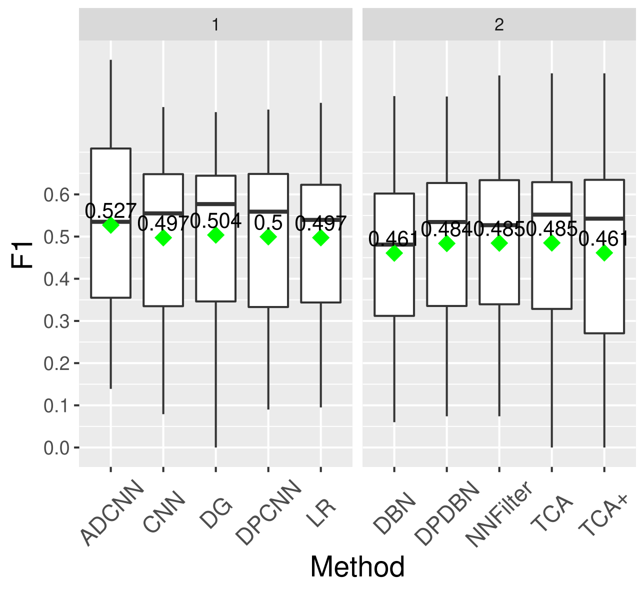


**FIGURE 4.** The Scott–Knott ESD test based on the F-measure results of our ADCNN method and other selected CPDP baseline methods. The average F-measure values are represented by a green diamond.

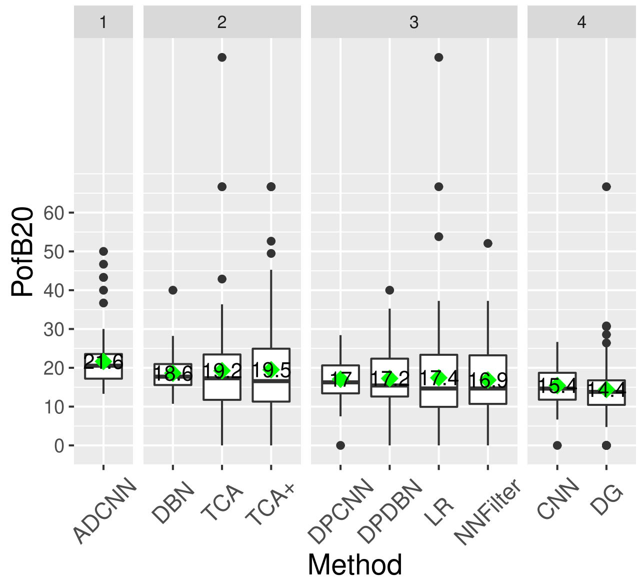


**FIGURE 5.** The Scott–Knott ESD test based on the PofB20 results of our ADCNN method and other selected CPDP baseline methods. The average PofB20 values are represented by a green diamond.

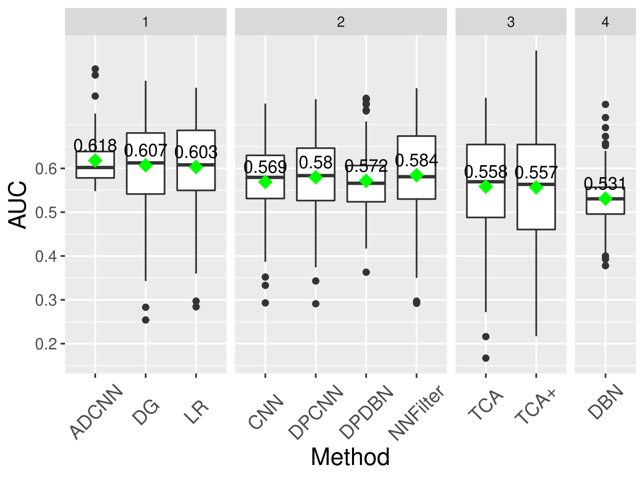


**FIGURE 6.** The Scott–Knott ESD test based on the AUC results of our ADCNN method and other selected CPDP baseline methods. The average AUC values are represented by a green diamond.

However, the Scott–Knott ESD test improved the Scott–Knott test from the following aspects. (1) The input data set is allowed to be non-normally distributed. (2) Any statistically different groups are merged if their effect sizes are trivial.

In our experiments, Figure 4, Figure 6, and Figure 5 show the results and statistical information of the Scott–Knott ESD test according to F-measure, AUC, and PofB20, respectively.

### A. RQ1: DO ADCNN OUTPERFORM THE TRADITIONAL CPDP METHODS THAT USE HANDCRAFTED FEATURES AS THE INPUT?

In this part, we compare our proposed ADCNN method with the traditional handcrafted features-based methods. This comparison aims at answering the research question QR1. In terms of F-measure, AUC, and PofB20, the W/T/L lines shown in Table 6, Table 7, and Table 8 indicate that the ADCNN method beats other methods in most experimental tasks.

Especially, for non-effort-aware scenario, the average F-measure of ADCNN is **0.527**, which outperform DG, LR, NNFilter, TCA, and TCA+ by **4.66%, 6.03%, 8.83%, 8.77%, and 14.18%**, respectively. The average AUC of ADCNN is **0.618**, which outperform DG, LR, NNFilter, TCA, and TCA+ by **1.81%, 2.42%, 5.75%, 10.67%, and 10.99%**, respectively. In terms of F-measure and AUC, Figure 4 and Figure 6 confirm that ADCNN is statistically different from NNFilter, TCA, and TCA+. Besides, the lower limit of DG and LR are lower in our experiments.

Under effort-aware scenario, the average PofB20 of ADCNN is **21.64**, which surpass DG, LR, NNFilter, TCA and TCA+ by **50.2%, 24.2%, 28.4%, 12.4% and 11.0%**, respectively. In terms of PofB20, Figure 5 illustrates that ADCNN is divided into a group, and ADCNN is statistically different from all the other baseline methods in our experiments.

In general, these experimental results indicate that our method performs better than classic CPDP methods under both non-effort-aware and effort-aware scenarios.

### B. RQ2: COMPARING WITH OTHER STATE-OF-THE-ART DEEP LEARNING METHODS, DO ADCNN PERFORM BETTER?

In this part, we compare our ADCNN approach with other state-of-the-art DL-based methods (i.e., DPCNN, CNN, DPDBN, and DBN).

For non-effort-aware scenario, the average F-measure of ADCNN is **0.527**, which surpasses DPCNN, CNN, DPDBN and DBN by a significant margin of **5.61%, 6.08%, 9.06%, and 14.46%**, respectively. The average AUC of ADCNN is

**0.618**, which surpasses DPCNN, CNN, DPDBN and DBN by a significant margin of **6.63%, 8.59%, 8.1%, and 14.28%**, respectively. In terms of F-measure and AUC, Figure 4 and Figure 6 show that the performance of ADCNN is superior to other DL-based CPDP methods in our experiments. Under effort-aware scenario, the average PofB20 of ADCNN is **21.64**, which surpasses DPCNN, CNN, DPDBN and DBN by **26.9%, 40.2%, 28.4%, 25.5% and 16.3%**, respectively. As shown in Figure 5, ADCNN and other DL-based methods are divided into statistically distinct groups. In terms of PofB20, the result of ADCNN is statistically different from DPCNN, CNN, DPDBN and DBN.

From the results, we conclude that the semantic features generated by CNN can improve more predictive performance than the features extracted by DBN. The transferable semantic features learned by ADCNN are more effective than neither CNN-extracted features or DBN-extracted features for the SDP domain.

In summary, ADCNN, which combines adversarial learning with CNN, can improve the predictive performance of SDP according to our experiments. We believe that the extracted transferable semantic features generated by ADCNN are better than other features generated by CNN or DBN. ADCNN will help software tester find defect-prone modules or files and allocate test resources effectively and efficiently.

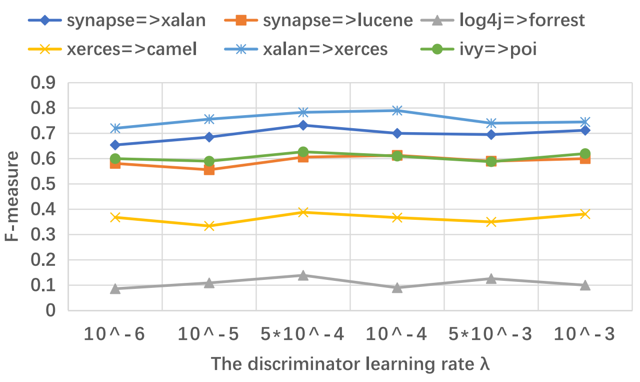


**FIGURE 7.** Different discriminator learning rate $\lambda$.

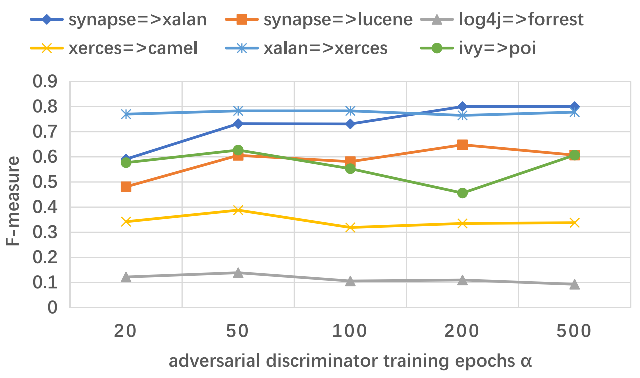


**FIGURE 8.** Different adversarial discriminator training epochs $\alpha$.

## VI. DISCUSSION

### A. WHY DOES ADCNN WORK?

In section 5, the experimental results have proved that the ADCNN method performs better than classic CPDP approaches and other DL-based CPDP approaches. The possible reasons can be summarized below:

(1) ADCNN extracts the potential semantic and structural features from the source code. For the SDP domain, pervious works [20], [42] have confirmed the effectiveness of the semantic features. As introduced in Section 3.3, we use CNNs as our source and target encoders. Consequently, ADCNN performs better than classic CPDP methods.

(2) ADCNN alleviates the data distribution differences between the source and target projects in the CPDP tasks. We use adversarial discriminative learning to learn an asymmetric mapping, where it changes the target representation distribution to match the source representation distribution. Therefore, ADCNN can learn transferable generated semantic features and perform better than other DL-based methods.

### B. THE IMPACTS ON DIFFERENT ADCNN PARAMETER SETTINGS

In our experiments, we should tune some parameters to obtain better predictive performance. These parameters include the discriminator learning rate $\lambda$, the adversarial discriminator training epochs $\alpha$, and the CNNs parameters. Following the previous work [20], we adopted the same CNN parameters. Therefore, we focused on discussing the impact of the different $\lambda$ and $\alpha$ on our model.

To illustrate the impact of $\lambda$, we show the F-measure of the ADCNN with various $\lambda$ (10e-6, 10e-5, 5*10e-5, 10e-4, 5*10e-4, 10e-3) in Figure 7. Due to space constraints, we only randomly selected six pairs of CPDP tasks. From the results in Figure 7, our ADCNN model achieves better results when $\lambda$ is 5*10e-4. So, we set the $\lambda$ value as 5*10e-4 in our experiments.

To avoid the effect of the parameter $\lambda$, we fixed $\lambda$ as 5*10e-4. We chose five different epoch numbers (i.e., 20, 50, 100, 200, 500) to verify the impact of the parameter $\alpha$. Similar to the experiments of $\lambda$, we selected six groups of CPDP tasks. Figure 8 presents the different F-measure values of our model in different parameters. We comprehensively considered the trade-off between the time of training model and performance gains and then chose 50 as $\alpha$.

### C. TIME COST

In this section, we discuss the time cost of all methods in our experiments. Besides, we answer the research question, RQ3. All experiments were run in a Ubuntu server with AMD Ryzen 3.60GHz and 16 RAM and accelerated by NVIDIA GTX1070.

The data preprocessing time of all models was consistent, and the prediction time was almost negligible. Therefore, We only counted the training time of models. Besides,

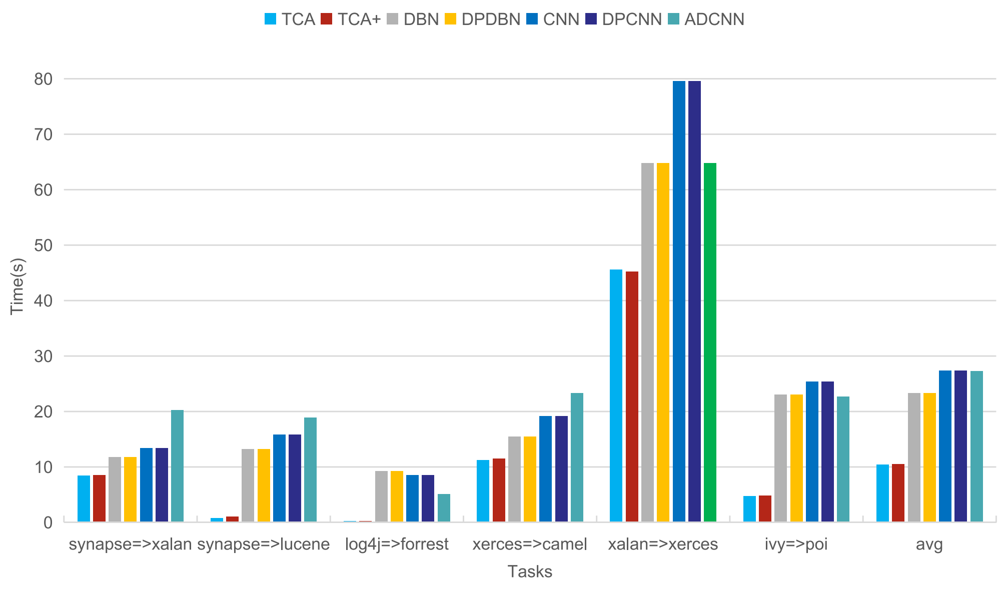


**FIGURE 9. Comparison of training time on 6 selected CPDP tasks and comparison of the total average training time.**

the training time of LR, DG, and NNFilter was trivial. So, we only compared the training time of our ADCNN model with the other models.

As shown in Figure 9, we recorded the training time of these seven models on six tasks. Moreover, we calculated the total average training time of 90 CPDP tasks. The average training time of ADCNN was 27 seconds. To sum up, ADCNN requires longer training time than traditional CPDP methods. Nevertheless, ADCNN takes about almost similar training time to other DL-based CPDP methods. We believe that ADCNN can get better results at an acceptable time, compared with other defect prediction methods.

### D. THREATS TO VALIDITY

#### 1) THE SELECTION OF DATASETS

Our selected projects are all written by Java language. They might not represent all software projects. Maybe, our proposed ADCNN method will perform better or worse results in other projects, including Java, Javascript, or Python language projects. Therefore, to make ADCNN more generalizable, we will evaluate our method on more projects in the future.

#### 2) EVALUATION METRICS

In this paper, we used F-measure and AUC as the non-effort-aware evaluation metrics and PofB20 as the effort-aware evaluation metric. However, there are some other evaluation metrics (e.g., G-measure), which are suitable means to evaluate the performance of two-class classifiers.

#### 3) PARAMETER COMBINATION

For better prediction performance, we tried different parameter combinations of the model. However, it is impossible to experiment with all possible combinations of parameters. In Section 6.2, we only assessed two important ADCNN parameters: the discriminator learning rate $\lambda$ and the adversarial discriminator training epochs $\alpha$. According to previous research experience [20], [42], we selected some empirical combinations of parameters. However, there may be a more suitable combination for our method.

## VII. CONCLUSION

In this paper, we proposed a novel ADCNN approach to solve the domain adaptation for CPDP. Our ADCNN method utilized the standard CNNs as the feature extractors. Therefore, ADCNN can learn the semantic and structural features from the source code directly. Furthermore, ADCNN employed adversarial discriminative learning to minimize the distribution divergence between the target project and source project. Hence, ADCNN could extract the transferable semantic features and then build a CPDP classifier. We compared our ADCNN method with nine other related state-of-the-art CPDP approaches on ten open source projects. The experimental results proved that ADCNN improved the predictive performance of both traditional feature-based methods and other DL-based methods.

To make ADCNN more generalizable, we will further investigate several problems as follow. Firstly, we will apply our model to more projects, including other programming language projects. Secondly, we will try to apply other domain adaptation methods to CPDP.

## REFERENCES

[1] H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, and M. Marchand, "Domain-adversarial neural networks," 2014, *arXiv:1412.4446*. [Online]. Available: http://arxiv.org/abs/1412.4446

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

[3] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*. Berlin, Germany: Springer, 2010, pp. 177–186.

[4] K. Bousmalis, G. Trigeorgis, N. Silberman, D. Krishnan, and D. Erhan, "Domain separation networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 343–351.

[5] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Inf. Sci.*, vol. 179, no. 8, pp. 1040–1058, Mar. 2009.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, Jun. 2002.

[7] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[8] H. Khanh Dam, T. Pham, S. Wee Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," 2018, *arXiv:1802.00921*. [Online]. Available: http://arxiv.org/abs/1802.00921

[9] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, May 2008.

[10] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Adv. neural Inf. Process. Syst.*, pp. 2672–2680, 2014.

[11] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[12] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2013, pp. 45–54.

[13] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, Sep. 2018.

[14] E. Jelihovschi, J. C. Faria, and I. B. Allaman, "ScottKnott: A package for performing the Scott-Knott clustering algorithm in R," *TEMA (São Carlos)*, vol. 15, no. 1, pp. 3–17, 2014.

[15] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 279–289.

[16] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 2014, pp. 414–423.

[17] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: http://arxiv.org/abs/1412.6980

[18] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 316–329, May 2007.

[19] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf. Softw. Technol.*, vol. 58, pp. 388–402, Feb. 2015.

[20] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 318–328.

[21] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Softw.*, vol. 12, no. 3, pp. 161–175, Jun. 2018.

[22] M.-Y. Liu and O. Tuzel, "Coupled generative adversarial networks," in *Proc. Adv. Neural Inf. Process. Syst.*, pp. 469–477, 2016.

[23] H. H. Maurice, *Elements of Software Science (Operating and Programming Systems Series)*. 1977.

[24] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[25] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Softw. Eng.*, vol. 17, no. 4, pp. 375–407, Dec. 2010.

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[27] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 2008, pp. 181–190.

[28] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 382–391.

[29] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Trans. Neural Netw.*, vol. 22, no. 2, pp. 199–210, Feb. 2011.

[30] S. Jialin Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.

[31] L. Peng, B. Yang, Y. Chen, and A. Abraham, "Data gravitation based classification," *Inf. Sci.*, vol. 179, no. 6, pp. 809–819, Mar. 2009.

[32] S. Qiu, L. Lu, and S. Jiang, "Multiple-components weights model for cross-project software defect prediction," *IET Softw.*, vol. 12, no. 4, pp. 345–355, Aug. 2018.

[33] S. Qiu, H. Xu, J. Deng, S. Jiang, and L. Lu, "Transfer convolutional neural network for cross-project defect prediction," *Appl. Sci.*, vol. 9, no. 13, p. 2660, 2019.

[34] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.

[35] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1253–1269, Dec. 2019.

[36] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, May 2015, pp. 99–108.

[37] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[38] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, Oct. 2009.

[39] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, "Adversarial discriminative domain adaptation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7167–7176.

[40] J. Wang, B. Shen, and Y. Chen, "Compressed C4.5 models for software defect prediction," in *Proc. 12th Int. Conf. Qual. Softw.*, Aug. 2012, pp. 13–16.

[41] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, early access, Oct. 23, 2018, doi: 10.1109/TSE.2018.2877612.

[42] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 297–308.

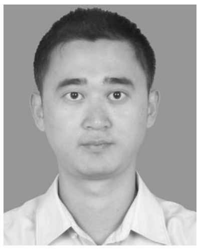
[43] T. Wang, Z. Zhang, X. Jing, and L. Zhang, "Multiple kernel ensemble learning for software defect prediction," *Automated Softw. Eng.*, vol. 23, no. 4, pp. 569–590, Dec. 2016.

[44] F. Wu, X.-Y. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, and Y. Sun, "Cross-project and within-project semisupervised software defect prediction: A unified approach," *IEEE Trans. Rel.*, vol. 67, no. 2, pp. 581–597, Jun. 2018.

[45] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.

[46] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.* New York, NY, USA: ACM, 2009, pp. 91–100.

**LEI SHENG** is currently pursuing the master's degree with the School of Computer Science and Engineering, South China University of Technology. His research interests include software defect prediction and transfer learning.



**LU LU** received the Ph.D. degree from Xi'an Jiaotong University, in 1999. He is currently a Professor with the School of Computer Science and Engineering, South China University of Technology, China. His main research interests include software engineering, software testing, and software architecture design.

**JUNHAO LIN** is currently pursuing the master's degree with the School of Computer Science and Engineering, South China University of Technology. His research interests include software defect prediction and transfer learning.