Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction

Jinyin Chen[1], Keke Hu[1], Yue Yu[2], Zhuangzhi Chen[1], Qi Xuan[3], Yi Liu[4,*], Vladimir Filkov[5]
chenjinyin@zjut.edu.cn,2111703352@zjut.edu.cn,yuyue@nudt.edu.cn,2111703088@zjut.edu.cn,xuanqi@zjut.edu.cn,
yliuzju@zjut.edu.cn,filkov@cs.ucdavis.edu
[1]College of Information Engineering, Zhejiang University of Technology, Hangzhou 310023, China
[2]College of Computer, National University of Defense Technology, Hefei 230000, China
[3]Institute of Cyberspace Security, Zhejiang University of Technology, Hangzhou 310023, China
[4]Institute of Process Equipment and Control Engineering, Zhejiang University of Technology, Hangzhou 310023, China
[5]Department of Computer Science, University of California, Davis, CA 95616, USA

ABSTRACT

Software defect prediction aims to automatically locate defective code modules to better focus testing resources and human effort. Typically, software defect prediction pipelines are comprised of two parts: the first extracts program features, like abstract syntax trees, by using external tools, and the second applies machine learning-based classification models to those features in order to predict defective modules. Since such approaches depend on specific feature extraction tools, machine learning classifiers have to be custom-tailored to effectively build most accurate models.

To bridge the gap between deep learning and defect prediction, we propose an end-to-end framework which can directly get prediction results for programs without utilizing feature-extraction tools. To that end, we first visualize programs as images, apply the self-attention mechanism to extract image features, use transfer learning to reduce the difference in sample distributions between projects, and finally feed the image files into a pre-trained, deep learning model for defect prediction. Experiments with 10 open source projects from the PROMISE dataset show that our method can improve cross-project and within-project defect prediction. Our code and data pointers are available at https://zenodo.org/record/3373409#.XV0Oy5Mza35.

KEYWORDS

Cross-project defect prediction, within-project defect prediction, deep transfer learning, self-attention, software visualization

ACM Reference Format:
Jinyin Chen[1], Keke Hu[1], Yue Yu[2], Zhuangzhi Chen[1], Qi Xuan[3], Yi Liu[4,*], Vladimir Filkov[5]. 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. In *42th International Conference on Software Engineering, May 23-29, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380389

*Corresponding author: Yi Liu
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380389

1 INTRODUCTION

Software defect prediction techniques can help software developers locate defective code modules automatically, to save human effort and material resources. Most prediction methods build prediction models based on modules in the source code and historical development data at different levels of modeling, e.g., commit changes, methods, and files [1]. In practice, based on whether the historical training data comes from the same project or not, we distinguish between within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) [2].

A large number of manually designed features, including static code features and process features [3], have been adopted to predict whether a module is defective or not. To improve on those, current software defect prediction studies mainly focus on two main directions: new feature extraction methods and classification methods learned from large-scale datasets. However, defect prediction techniques which feed manually designed features into machine learning algorithms for classification, have some limitations [4]. The required feature engineering is time consuming and requires that special tools be used upstream, such as code complexity analysis, submission log mining, and code structure analysis tools. Consequently, many features can be difficult to capture in some projects. For example, semantic code information, such as the features hidden in abstract syntax trees (ASTs), may not be effectively represented by existing traditional features. In addition to the inconvenience of feature engineering for traditional features, as described by Wan et al. in a recent review of defect prediction [5], semantic information can be more capable than syntax information to distinguish one code region from another. Thus, while AST-conveyed features can be useful for defect prediction, such approaches are indirect, requiring additional tools in order to build and mine the ASTs. Moreover, in such approaches, the source code is most frequently not used once the ASTs are extracted.

A number of machine learning methods, e.g., support vector machines (SVMs) [6], naive Bayes (NB) [7], decision trees (DTs) [8], and neural networks (NNs) [9], have been applied to defective module prediction. In particular, recent research has been conclusive that deep learning networks are highly effective in image classification, feature extraction, and knowledge representation in many areas [10–15]. In defect prediction specifically, to better generate semantic features, a state-of-the-art method [16] leveraged deep belief network (DBN) for learning features from token vectors extracted from programs' ASTs. On this basis, Li et al. [17] and Dam

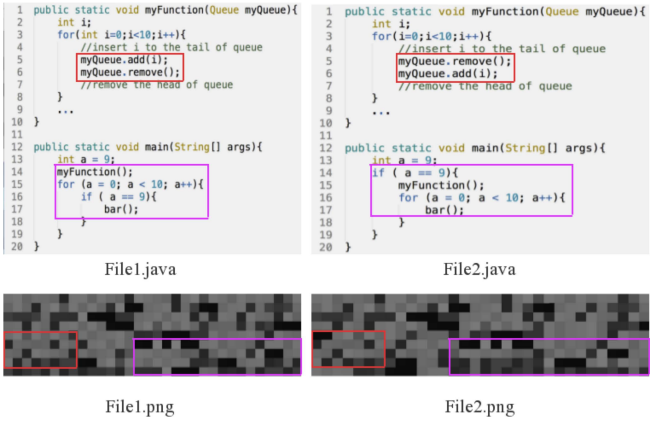


Figure 1: A motivating example

et al. [18] used the structural information of programs and the semantic relations between keywords to improve defect prediction with convolutional neural network (CNN) and long-short-term-memory networks (LSTMs). In those papers, the required feature engineering was significant and required specific tools be used upstream.

In this work we are motivated by the possibility of improving defect prediction by avoiding intermediary representations, e.g. ASTs, and instead obtaining code semantic information directly. To that end, inspired by the power of existing deep learning platforms for image classification, in this paper we propose a more direct way to use programs' semantic information to predict defects: by representing source code as images and training image classification models on those images.

Figure 1 shows a motivating example. The two Java files on top, *File1.java* and *File2.java*, are similar, both containing 1 *if* statement, 2 *for* statements, and 4 function calls. Nevertheless, the files' semantic and structural features are different. We wondered if a visualization can help to tell that those two programs are different. To do so, we turned code characters into pixels, their colors based on the characters' ASCII decimal value, and then arranged those pixels in a matrix, thus obtaining code images. By comparing those images, we were able to visually recognize significant differences between the corresponding programs, as shown in the bottom of Figure 1[1]. Both semantic and structural differences can be recognized visually in those images. That leads us to our driving thesis:

> Semantic and structural similarities between two programs can be effectively identified by visually comparing program images.

To test that thesis, we start from a well known data set from the PROMISE repository, of 10 projects with known defective files. Once the files are converted to images, we automatically extract features and build a classification model with the popular AlexNet platform [19]. In addition, we use deep transfer learning [20] and self-attention mechanism [21] (the details are in Figure 4 and Section 3.2) to further improve the defect prediction, especially CPDP. Finally, we propose an end-to-end framework, i.e., deep transfer learning for defect prediction (DTL-DP), to automatically predict whether a program file contains defects or not.

This paper makes the following contributions:

- We propose a novel approach for defect prediction based on visualizing program files as images, in order to capture their semantic and structural information.
- We propose an end-to-end deep-learning framework, DTL-DP, comprised of a deep transfer learning model combined with a self-attention mechanism, which takes program file images as input, and outputs labels, either 'defective' or 'not defective'.
- Our experiments, on 10 OS Java projects, show that our approach improves both WPDP and CPDP. The advantages of DTL-DP in CPDP are much more obvious than in WPDP. We show that self-attention has the greatest contribution to WPDP, and transfer learning contributes most to CPDP.

The remainder of this paper is organized as follows. First, we review related work in Section 2. Then we describe DTL-DP and present our experimental setup in Sections 3 and 4, respectively. After that, we discuss the performance of DTL-DP in Section 5, and the threats to validity in Section 6. Finally, we conclude the work and provide several potential future directions in Section 7.

2 RELATED WORK

Defect prediction (DP) As one of the primary areas of interest in software engineering research, DP has been receiving significant attention [22–25]. A number of studies have focused on manually designing features or producing new combinations of existing features from labeled historical defects. Those features are typically fed into a machine learning-based classifier to determine if a file is defective. Commonly used features can be divided into static, e.g., code size and code complexity (e.g. Halstead features [26], McCabe features [27], CK features [28]), and process features, like the behavioral differences of developers in the software development process. Many studies have demonstrated that process features can predict software quality [29]. Moser et al. [24] used authors, past fixes, the number of revisions and ages of files as features to predict defects. Nagappan et al. [30] indicated that code churn was effective for DP. Hassan et al. [31] used entropy of changes to predict defects. Other process features are helpful too, including individual developer characteristics [22, 32] and their collaborations [33–35].

Based on these features, many machine learning models, including SVM [6], NB [7], DT [8], NN [9], etc., have been built for the two different DP tasks: within-project, WPDP, and cross-project, CPDP. For WPDP, the training set and test set come from the same project, while for CPDP, they come from different projects. While WPDP can give better results, it is of limited use in practice as it is often difficult to obtain enough training data for a new project. Some studies have instead used related projects to build prediction models with sufficient historical data, and then used them to predict defects in the target project [36–39]. Panichella et al. [40] proposed an approach named CODEP, which uses a classification model to combine the results of 6 classification algorithms (including logistic regression, radial basis function network, and multi-layer perceptrons) for CPDP. Turhan et al. [41] and Peters et al. [42] used different strategies to select appropriate instances in a source project, based on

[1]There were a number of technical details and choices we had to make when converting programs into images; we discuss those in the Methods section.

nearest neighbor methods, to train the prediction models. Additionally, the application of transfer learning is receiving more attention. Xia et al. [43] proposed an approach named HYDRA to build classifiers using genetic algorithm and ensemble learning. Ma et al. [44] proposed a transfer naive Bayes (TNB) method to assign weights to training instances, and then used them to construct a prediction model. Nam et al. [45] proposed TCA+, which uses TCA [46] and optimizes the normalization to improve CPDP.

Recently, some studies have used deep learning to directly get features from source code for DP. Wang et al. [16] deployed deep belief networks (DBNs) to learn semantic features from token vectors extracted from ASTs automatically, and leveraged the learned semantic features to build machine learning models for DP. Li et al. [17] used convolutional neural networks (CNNs) to generate features from source code and combined CNN learned features with traditional features to further improve upon the prediction. Similarly, Dam et al. [18] proposed a prediction model that takes ASTs representing the source file as input. It used an LSTM architecture to capture long-term dependencies which often exist between code elements.

Our proposed approach differs from the above in that we do not use feature engineering followed by machine learning-based classifiers. We also do not need ASTs to bridge the gap between defect prediction and deep learning. Instead, our approach is based on feeding software visualizations into an automatic, image based, feature discovery pipeline.

Software visualization (SV) for DP SV has long history in software engineering research and practice [47, 48]. It has been used for visualizing code structure and features [49], code execution [50], and evolution [51], of large codebases in particular [52]. SV has also been applied to bug repositories, where it has been helpful in correlating bugs with code structure [53]. In fact, the way code was visualized in a recent study on malware code visualization [54] has, in part, inspired us in this paper. However, to the best of our knowledge, there is a dearth of applications of SV to software defect prediction in the research literature, perhaps because prior to DL approaches no connection was seen between the two areas.

Deep transfer learning (DTL) Transfer learning (TL) is an important tool that can help when there is insufficient training data in machine learning. It works by transferring information from a source domain to a target domain. The key is to relax the assumption that the training and test data must be independent and identically distributed [55]. This is helpful in areas where it is difficult to get enough training data. In the field of DP, studies have shown that CPDP can achieve better performance with transfer learning [56].

Most TL studies are based on traditional machine learning methods. Recently, deep learning-based transfer learning studies have emerged, called *deep transfer learning, DTL*. Based on the techniques used in them they can be divided into four categories: network-based, instance-based, mapping-based, and adversarial-based [57].

Instance-based DTL is implemented through a specific weight adjustment strategy. The instances selected from the source domain are assigned weights that complement the training data in the target domain [58–61]. Network-based DTL refers to the use of a partially trained network in the source domain for the deep neural network of the target domain, including its network structure and connecting parameters [62, 63]. Adversarial-based DTL refers to introducing adversarial technology inspired by generative adversarial nets (GAN) to find a transferable representation that is applicable to the source and target domains [64, 65]. Mapping-based DTL refers to mapping instances of the source and target domains to a new data space, where the distributions of the two domains are similar. TCA [46] and TCA-based methods have been widely used in applications of traditional transfer learning [66]. Tzeng et al. [67] used maximum mean discrepancy (MMD) to measure the sample distribution after deep neural network processing and learned domain invariant representations by introducing an adaptation layer and additional domain confusion loss. Long et al. [68] improved previous work by replacing MMD with multiple kernel variant MMD (MK-MMD), originally by [69], and proposed a method called deep adaptation networks (DAN). Long et al. [70] proposed joint maximum mean discrepancy to promote the transfer learning ability of neural networks to adapt to the data distribution of different domains. The Wasserstein's distance, proposed by Arjovsky et al. [71], has been used as a new distance measure to find a better mapping between domains.

To the best of our knowledge, DTL methods have not yet been used in defect prediction. In this paper, we propose a novel, mapping-based DTL method using a self-attention mechanism for defect prediction.

3 APPROACH

The overall framework of our approach *deep transfer learning for defect prediction*, DTL-DP, is shown in Figure 2. It is comprised of two stages, (1) source code visualization and (2) modeling with DTL. In the first stage we use a visualization method to convert program files into images. In the second stage, we build a DTL model based on the AlexNet network structure with transfer learning and a self-attention mechanism to construct an end-to-end defect prediction framework. We use that framework to predict if a new instance file is defective or not.

In a nutshell, our approach takes the raw program files of a training and test sets directly as input and generates images from them, which are then used to build the evaluation model for defect prediction. Specifically, since the input data of the network model based on the CNN structure should be in the form of images, we build a mapping to convert the files into images. Then we use the first 5 layers of AlexNet as *Feature-Net* to generate features from the images. The shallow CNN layers correlate the presence and absence of defects to the overall code structure, gradually deepening the granularity from function/loop bodies to function names, identifiers, etc. These features are then fed into the *Attention Layer* where they are used in assigning weights, and highlighting features, which are more helpful in the classification. The re-weighted features of the training and test sets are used to calculate MK-MMD, which is used to measure the difference between their distributions, as the MMD loss. After that, the re-weighted features of the training set are entered into the fully connected layers to calculate the cross entropy as the classification loss. The weighted sum of the MMD loss and classification loss is fed back to train the whole network, including *Feature-Net*, *Attention Layer* and the fully connected layers. Finally, based on the source code visualization method and the DTL model,

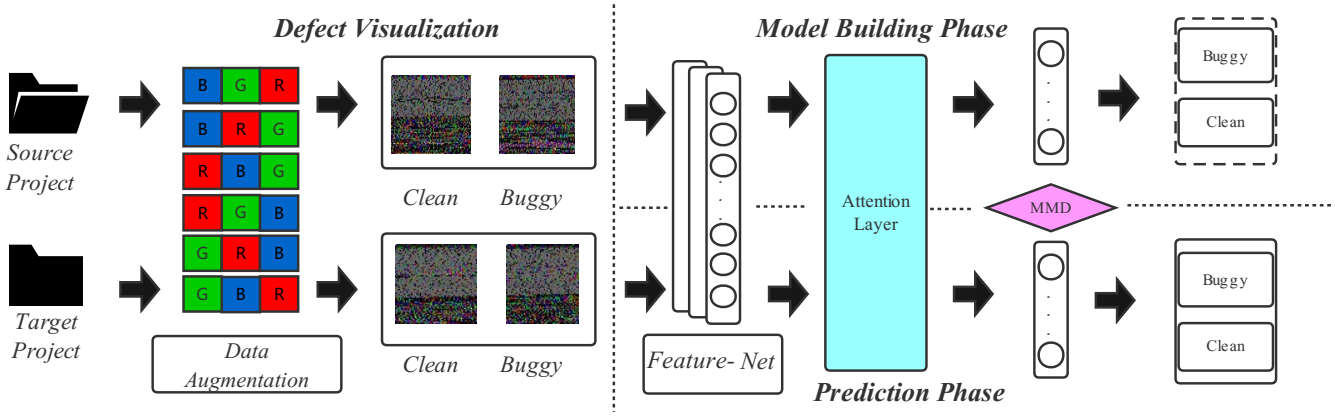


Figure 2: The overall framework of DTL-DP

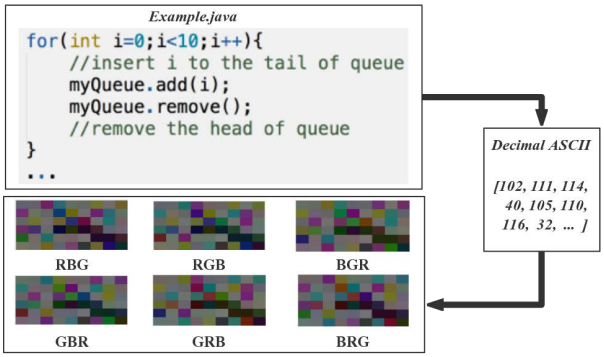


Figure 3: The process of converting code to color images

we build, train, and evaluate a defect prediction model. We give the details in the following.

3.1 Source Code Visualization

Converting code to images retains information and facilitates available deep neural network (DNN) -based methods to improve defect prediction performance. Most existing methods ignore associations between extracted features and the classification task. Our proposed method forms a unified end-to-end framework of feature extraction, modeling, and prediction.

Figure 3 shows how we convert each program file into 6 different images. We call this process *source code visualization* and the images produced *code images*. First, each source file is converted into a vector of 8-bit unsigned integers corresponding to the ASCII decimal values of the characters in the source code, e.g., 'a' is converted to 97, etc. We then generate an image from that vector, by arranging its values in rows and columns and interpreting it as a rectangular image. A straight forward way to visualize the 8-bit values as color intensities would be as levels of gray, e.g., 0=black, and 255=white. However, the relatively small size of our original training set means that if we did that we would end up with a deep model that cannot be sufficiently well trained. Fortunately, an additional benefit to using images for training is that we can augment the original data set to produce a larger data set with the same semantics. Common data augmentation methods for image data sets include flipping (both vertically and horizontally), rotating, cropping, translating (moving along the x or y axis), adding Gaussian noise (distortion of high frequency features), zooming and scaling. Since the sizes of our programs are small, and we want to retain the semantic and structural features in the images, the above image data augmentation methods could result in data loss.

Instead, we designed a novel, color based augmentation method to generate 6 color images from each source code file. Namely, each pixel in a color image can be represented by three primary color components, or channels: red (R), green (G), and blue (B). The letters R, G, B, and thus the color channels, can be ordered in 6 different ways: RBG, RGB, BGR, BRG, GRB and GBR. By adopting a different order every time, as shown in Figs. 2 and 3, we generate six different images for each program file. For example, 'for' is converted to $[102, 111, 114]$, and the 3 values are filled into the R, G, and B channels in 6 different ways to obtain 6 differently colored pixels.

The above method expands our data set six-fold, and because of the nature of the downstream analysis, the generated samples are reasonable since the representation is changed only by the order of the different channels. [2]

Whereas we generated 6 images for each instance in the training set, we randomly selected an RGB permutation for testing. We found that it was not necessary to generate all six images for testing because our experiments on 10 datasets showed that the performance of using the different permutations was quite comparable. We randomly chose one to increase the speed in practice. The data augmentation was used to improve the efficacy of the model.

After the above, for each of the 6 orderings of R, G, B, we obtain a vector of pixels of length one third the original code file size (each source code character is assigned to a color channel, and thus three characters in a row represent a pixel). We then turned those

[2] A CNN model learns the features by convolution. Since we use ImageNet's pre-trained AlexNet model, the initial parameters of the convolution kernel are different for each channel, which means even though byte sequences are the same, the final set of features obtained by sequentially inputting into the model the different channels is also different.

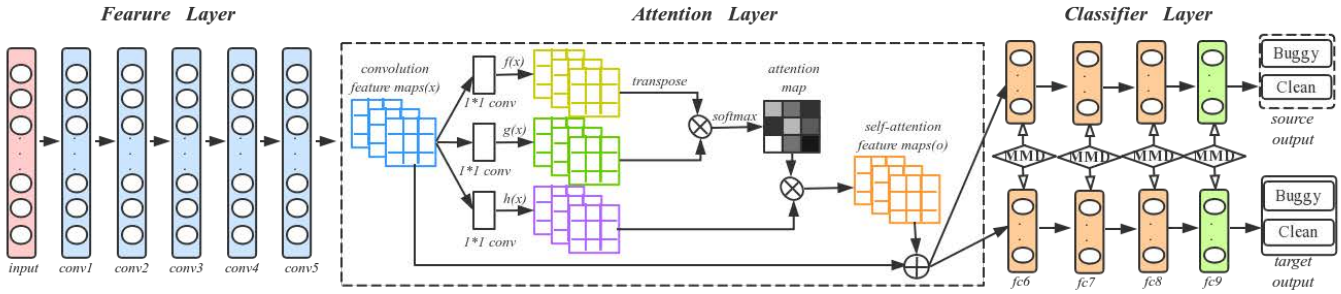


Figure 4: The training and testing process of our approach, DTL-DP, inspired by the original DAN paper [68]

Table 1: Image Width for Various File Sizes

File Size Range	Image Width
<10 kB	32
10 kB - 30 kB	64
30 kB - 60 kB	128
60 kB - 100 kB	256
100 kB - 200 kB	384
200 kB - 500 kB	512
500 kB - 1000 kB	768
>1000kB	1024

6 image vectors into image matrices of width and length such that $width * length = vector\_size/3$. We note that the first convolution kernel of AlexNet is of size 11*11, so if the image is too narrow, the convolution performance will be poor. And the image with a suitable width can be convolved to obtain more efficient semantic and structural features in contexts with a proper sequence length. Table 1 gives some recommended image widths for different file sizes, based on previous work [72, 73]. We adopt those in this work.

3.2 Modeling with Deep Transfer Learning

Next we describe our approach, DTL-DP. The goal is to learn transferable features across projects, and build a classifier $y = \phi(x)$ of file defectiveness, with source code supervision, where x is the representation of the sample and y is the predicted label. In the defect prediction problem, we are given a *source* project $P_s = \{(x_i^s, y_i^s)\}_{i=1}^{n_s}$ with n_s labeled instances, and a *target* project $P_t = \{(x_i^t, y_i^t)\}_{i=1}^{n_t}$ with n_t unlabeled instances. In WPDP, the *source* project is the previous version of the *target* project, while in CPDP, the *source* project is a related project of the *target* project. The sample distributions p and q in the source and target projects are often similar in WPDP, but different in CPDP.

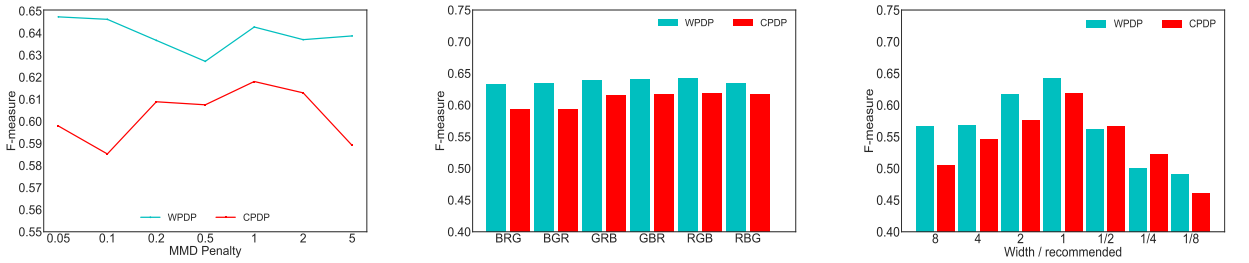
For our deep network architecture we adopt a similar architecture as that of deep adaptation networks (DANs) [68] to capture the semantic and structural information of the source code. To the original DAN model we add an attention layer to further enhance the expressive ability of important features. The overall architecture is illustrated in Figure 4. In particular, our DTL-DP consists of an input layer, five convolution layers ($conv1 - conv5$, from AlexNet), an *Attention-Layer*, and finally four fully-connected hidden layers ($fc6 - fc9$), working as a classifier. The structure and parameters of ($conv1 - conv5$) and ($fc6 - fc8$) are consistent with a DAN. But since defect prediction is a binary classification problem, a fully-connected layer ($fc9$) is added to obtain a binary result at the end.

We adopt the defaults for AlexNet, so the input to our DTL-DP must be a 224*224 size image cropped from a three-channel (RGB) image of size 256*256. The code image is placed in the approximate center of the 256*256 image (the determination of the size of each code image is described later, in Sect. 3.3). We note that the code image can be smaller than 224*224 pixels due to the variance in the size of the code files. If this happens the image is padded around with blank (zero valued) pixels. Padding should not negatively effect the qualitative performance of feature detection in the images; in fact for deeper DL architectures like ours, padding has been shown to provide extra contrast to the embedded image for each of the layers as well as buffering against data loss by each layer [74].

Training deep models requires a significant amount of labeled data, which is not available for a new project, so we first pre-trained an AlexNet model on ImageNet 2012. Unlike a DAN, which freezes $conv1 - conv3$ and fine-tunes $conv4 - conv5$, we fine tune all convolution layers $conv1 - conv5$ by taking the parameters of the pre-trained models as initial parameters that we then optimize during the training phase. We do that to minimize the differences between our code images and the actual object images in ImageNet 2012.

In order for DTL-DP to focus on the key features in different defect images, and thus further improve prediction, we employ a self-attention mechanism into our model inspired by the good performance of self-attention in GANs [21]. As shown in the *Attention Layer* in Figure 4, the attention mechanism makes the feature map generated by layer *conv5* be the self-attention feature map input to the next layer, *fc6*. Specifically, at first, it linearly maps the input features x (it is a 1*1 convolution, to compress the number of channels, i.e., $out\_chanels = in\_chanels/8$), and produces $f(x), g(x)$, and $h(x)$, where $f(x) = w_f x, g(x) = w_g x, h(x) = w_h x$. The difference between the three is that the size of $h(x)$ is still the same as x, but the other two are not. Thus, if the width of x is W, the height H and the number of channels C, the size of x is $[C, N]$, where $N = W * H$, the size of *f(x)* and *g(x)* is $[C/8, N]$, but the size of *h(x)* is $[C, N]$. The transposed $f(x)$ and $g(x)$ are matrix-multiplied to obtain the autocorrelation in the features, i.e., the relationship of each pixel to all other pixels, where $S_{ij} = f(x_i)^T g(x_j)$. Then, softmax is applied to the autocorrelation features, S, to get the attention map, comprised of weights with values between 0 and 1:

$$\alpha_{j,i} = \frac{\exp(S_{ij})}{\sum_{i=1}^{N} exp(S_{ij})} \quad . \tag{1}$$



(a) Effect of different MMD Penalty, λ (b) Effect of color channel order (c) Effect of code image width:recommended ratio

Figure 5: Sensitivity studies to aid in choosing the parameters in the model

After that, the output of the *AttentionLayer* is the self-attention feature map $o = (o_1, o_2, ..., o_j, ..., o_N)$, where

$$o_j = \sum_{i=1}^{N} \alpha_{j,i} h(x_i). \tag{2}$$

Then, each fully connected layer learns a nonlinear mapping $h_i^l = f^l(w^l h_i^l + b^l)$, where h_i^l is the lth layer hidden representation of feature x_i, w^l and b^l are the weights and biases of the lth layer, and f^l is the activation function, using ReLU ($f^l(o) = max(0, o)$) for $fc6 - fc8$ and softmax for $fc9$. If we let Θ denote the set of all DTL-DP parameters, the empirical risk of DTL-DP, then, is

$$\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} F(\phi(o_i), y_i) \tag{3}$$

where F is the cross-entropy loss function, n is the number of instances and $\phi(o_i)$ is the conditional probability that the DTL-DP assigns o_i to label y_i. It is used to calculate the final loss in Equation 7.

To make the distribution of the source and target projects similar, the same multi-layer adaptation and multi-kernel MMD strategy as in a DAN [68] are used in our model. The feature mapping function σ is defined as the combination of m positive semi-definite kernels k_u,

$$< \sigma(o^s), \sigma(o^t) >= k(o^s, o^t) \tag{4}$$

$$k = \sum_{u=1}^{m} \beta_u k_u \tag{5}$$

where $\beta \geq 0$ are the weights of the kernels.

The MK-MMD $d_k(p, q)$ of the probability distributions p and q is defined as the reproducing kernel Hilbert space distance between the mean embedding of p and q. The square formula of MK-MMD is defined as

$$d_k^2(p, q) = \| E_p[\sigma(x^s)] - E_q[\sigma(x^t)] \|^2 . \tag{6}$$

We see then that the smaller $d_k(p, q)$ is, the more similar p and q are. If d_k is 0, the distribution of the target project is the same as that of the source project. So the final loss function for train DPL-DP is

$$\frac{1}{n} \sum_{i=1}^{n} F(\phi(o_i), y_i) + \lambda \sum_{l=l_1}^{l_2} d_k^2(D_s^l, D_t^l) \tag{7}$$

where $D_s^l$is the lth layer hidden representation for the source and target, $d_k^2(D_s^l, D_t^l)$ is the MK-MMD between the source and target evaluated on the lth layer representation, λ is a penalty parameter, and l_1 and l_2 are fc layers to calculate the MK-MMD between source and target. In our implementation of DTL-DP, we set $\lambda = 1$, $l_1 = 6$ and $l_2 = 9$, as per the original DAN work [68].

3.3 Model Sensitivity to Parameter Choices

We have made a number of choices to make our modeling platform work effectively. Here we justify those choices by presenting sensitivity studies. For this analysis we used all the data.

Hyperparameter λ The DTL-DP model has a hyper-parameter λ for the MMD penalty, that adjusts the final loss. We conducted λ parameter sensitivity experiments in both the WPDP and CPDP setting. We fix the other parameters and range λ in $\{0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5\}$. The results are shown in Figure 5(a). While fairly constant across the range, small variations exist. In CPDP, the F-measure first increases and then decreases along the hyper-parameter λ. In WPDP, λ affects the performance of DTL-DP, but in the opposite direction. We chose $\lambda = 1$ here.

Different Color Orders The selection of the R,G,B permutations in the target project images may also potentially affect the outcome. We performed experiments with all six different orderings. When testing them, we set λ to 1 and changed the image type of the target project to each one in $\{RGB, RBG, BRG, BGR, GBR, GRB\}$. The results are shown in Figure 5(b). The target images in different color orders have somewhat different performance. But overall the values are close to DTL-DP's average performance.

Different Image Widths To explore the impact of different choices for image widths on the results, we performed additional experiments.

For our code images, 3 bytes of source code are needed to form 1 pixel, each byte for one of the R, G, and B channels. The sizes of the source files are between 0 and 100 kb, corresponding to images with pixel count between 0 and 34, 133 (100*1024/3). The size of the image is $(width, height)$, where $width$ is obtained from Table 1 according to the size of the code file ($size$).

$$height = \frac{size * 1024}{3 * width} \tag{8}$$

For example, the size of image converted from a 20 kb code file is (64 , 107).

We experimented with image widths of 1/8, 1/4, 1/2, 2, 4, and 8 times of the recommended widths in Table 1. The larger the multiplier, the wider the image. For images of different widths, we

repeated the previous experiment and obtained results for the mean F-measure, shown in Figure 5(c). We see that the closer to the recommended width from Table 1, the better the average result. In addition, we found that a wider image is better than a narrower one. And the closer the image is to a square, the better the prediction. Therefore, we made the generated images be as close to a square as possible.

3.4 Training and Prediction

When training the DTL-DP, code images are generated from the labeled instances in the source project and the unlabeled instances in the target project, and then, are simultaneously input into the model. They share the convolution layers $conv1-conv5$ and $AttentionLayer$ to extract their respective features, and calculate the MK-MMD between source and target projects in the fully connected layers $fc6-fc9$. It should be noted that we only calculate cross entropy for the source project, because the target project's labels are not provided. We use a mini-batch stochastic gradient descent along the loss function in order to train the parameters of the entire model. We train for 500 epochs for each pair of source and target projects, then pick the epoch with the best F-measure (described in section 4.3) from which to read out the parameters of the final model. Finally, the files from the target project that need to be predicted are converted into code images and then input into the trained model for classification.

Overfitting is always a possibility with such pipelines. We moderate it here with our choices of (1) augmenting the dataset by using six color channel permutations, described in section 3.1, (2) selecting a simple AlexNet model structure, and (3) using the ImageNet 2012 pre-trained model to reduce fluctuations.

4 EXPERIMENTAL SETUP

We conducted experiments to asses the performance of DTL-DP and to compare it with existing deep learning-based defect prediction approaches, for both within-project, WPDP, and cross-project, CPDP, defect prediction. We ran experiments on a modern-day Linux server with 3 Titan XP GPUs. Unless otherwise stated, each experiment was run 10 times and the average results are reported.

4.1 Dataset Description

In order to directly compare our work with prior research, we used publicly available data from the PROMISE[3] data repository, which has been widely used in defect prediction work [16–18, 75]. We selected all open source Java projects from this repository, and collected their version number, class names, and the buggy label for each file. Based on the version number and class name, we obtained the source code for each file from GitHub[4] and fed it to our end-to-end framework. In total, data for 10 Java projects were collected. Table 2 shows the details of these projects, including project description, versions, the total and average number of files and the defect rate. It should be noted that the average number of files over all projects ranges between 150 and 1046, and the defect rates of the projects have a minimum value of 13.4% and a maximum value of 49.7%. The number of files in some projects is not sufficient to train deep models, and the classes are imbalanced, thus augmentation is needed, as described above.

[3] http://openscience.us/repo/defect/
[4] https://github.com/apache

4.2 Baseline Comparison Methods

To evaluate the performance of our end-to-end framework DTL-DP for defect prediction, we compare it with the following baseline methods in the WPDP setting:

- *Semantic [16, 75]:* the state-of-the-art method which employs deep belief networks, DBN, on source code to extract semantic features for defect prediction.
- *PROMISE-DP [16]:* a traditional method which builds an alternating decision tree, ADTree, classifier based on the original 20 features of the PROMISE dataset.
- *DP-LSTM [18]:* a long short-term memory, LSTM,-based deep neural network model which uses ASTs to represent source files and predict whether the file is defective or not.
- *DP-CNN [17]:* a convolutional neural network, CNN,-based model which is seeded by AST-derived numerical vectors to automatically learn semantic and structural features of programs. The CNN-learned features are used to train the final classifier in combination with traditional features.

For the cross-project settings, CPDP, *Semantic* and *PROMISE-DP* could not be used directly. Instead, we used the following:

- *DBN-CP [16, 75]:* a variant of *Semantic* which trains a DBN by using the source project and generates semantic features for both the source and target projects.
- *TCA+ [45]:* the state-of-the-art technique for CPDP.

To obtain the training and test data, we followed the processes established in [16]. For WPDP, we use two consecutive versions of each project listed in Table 2. The older version is used to generate the training data, and the more recent version is used as test data. For CPDP, we pick versions randomly from each project, for 11 target projects. And for each target project, we select 2 source projects that are different from the target projects. We use the same 22 test pairs as in [16]. When implementing the baseline methods, we use the same network architecture and parameter settings as described in the papers that introduced them.

4.3 Performance measures

To evaluate the prediction performance we use the F-measure, a widely adopted metric in the literature [16–18, 43, 45]. The F-measure captures a predictor's accuracy and combines both precision and recall, for a comprehensive evaluation of predictive performance.

Specifically, a prediction that a file is defective is called a true positive (TP) if the file is in fact defective, and false positive (FP) otherwise. Similarly, a prediction that a file is not defective is a true negative (TN) if the file is in fact not defective, and false negative (FN) otherwise. Then, the precision (P), recall (R), and F-measure are defined as:

$$P = TP/(TP + FP) \tag{9}$$

$$R = TP/(TP + FN) \tag{10}$$

$$F = (2 \times P \times R)/(P + R). \tag{11}$$

5 RESULTS

This section discusses our results of comparing DTL-DP to baseline tools for defect prediction.

Table 2: Dataset Description

Project	Description	Versions	#Files	Avg files	Avg size(kb)	% Defective
ant	Java based build tool	1.5,1.6,1.7	1,465	488	6.2	13.4
camel	Enterprise integration framework	1.2,1.4,1.6	3,140	1,046	2.9	18.7
jEdit	Text editor designed for programmers	3.2,4.0,4.1	1,935	645	8.7	19.2
log4j	Logging library for Java	1.0,1.1	300	150	3.4	49.7
lucene	Text search engine library	2.0,2.2,2.4	607	402	3.8	35.8
xalan	A library for transforming XML files	2.4,2.5	1,984	992	4.6	29.6
xerces	XML parser	1.2,1.3	1,647	549	2.9	15.7
ivy	Dependency management library	1.4,2.0	622	311	4.1	20.0
synapse	Data transport adapters	1.0,1.1,1.2	661	220	3.8	22.7
poi	Java library to access Microsoft format files	1.5,2.5,3.0	1,248	416	3.6	40.7

Table 3: F-measure of DTL-DP, Semantic (Seman), PROMISE-DP (PROM), DP-LSTM (LSTM) and DP-CNN (CNN) for WPDP

Project	Version	Seman	PROM	LSTM	CNN	DTL-DP
ant	1.5->1.6	**91.4**	47.7	45.2	53.2	70.0
	1.6->1.7	**94.2**	54.2	37.9	56.6	59.3
camel	1.2->1.4	**78.5**	37.3	32.3	46.1	40.6
	1.4->1.6	37.4	39.1	40.0	**50.8**	38.5
jEdit	3.2->4.0	**57.4**	55.6	47.2	56.4	49.4
	4.0->4.1	61.5	54.6	49.0	58.0	**68.7**
log4j	1.0->1.1	**70.1**	58.7	53.9	63.2	68.8
lucene	2.0->2.2	65.1	50.2	75.1	76.1	**78.3**
	2.2->2.4	77.3	60.5	75.1	72.1	**77.6**
xalan	2.4->2.5	59.5	51.8	65.7	60.1	**79.4**
xerces	1.2->1.3	41.1	23.8	26.8	37.4	**82.0**
ivy	1.4->2.0	35.0	32.9	19.1	34.7	**82.9**
synapse	1.0->1.1	54.4	47.6	45.4	53.9	**54.7**
	1.1->1.2	58.3	53.3	53.3	55.6	**65.9**
poi	1.5->2.5	64.0	55.8	**80.8**	58.9	68.5
	2.5->3.0	**80.3**	75.4	77.7	78.4	42.6
Average		64.1	49.9	51.5	57.0	**64.2**

5.1 RQ1: How does DTL-DP compare to feature-based machine learning methods and AST-based deep learning methods, in WPDP?

We compare DTL-DP to 4 baseline approaches, representing two different kinds of defect prediction methods. *PROMISE-DP* is the baseline representative of traditional feature-based machine learning methods. *Semantic*, *DP-LSTM* and *DP-CNN* are the baselines for deep learning-type methods, based on extracting features from AST. Guided by prior work, we conducted 16 sets of WPDP experiments, each using two versions of the same project. The older version is used to train the prediction model, and the newer version is used to evaluate the trained model.

Table 3 shows the F-measure values for the within-project, WPDP, defect prediction experiments. The highest F-measure values of the 5 methods are shown in bold. Since the methods based on deep learning include some randomness, we run DTL-DP, *Semantic*, DP-LSTM and DP-CNN 10 times for each experiment. On average, the F-measure of our approach is 0.642, and the *PROMISE-DP*, *Semantic*, *DP-LSTM* and *DP-CNN* achieve 0.499, 0.641, 0.515 and 0.570, respectively. The results demonstrate that our approach is competitive, and may improve on defect prediction compared to *PROMISE-DP*, *DP-LSTM* and *DP-CNN*. The results of *Semantic* and our approach are similar.

> The proposed DTL-DP is effective, and it could improve the performance of WPDP tasks. Like other deep learners, it is sensitive to small file sizes and unbalanced data.

5.1.1 Case Study: WPDP Discrimination of DTL-DP. t-SNE is a non-linear dimensional reduction algorithm that is effective in visualizing similarities and helping identify clusters in complex data sets [76]. To give insight in the performance of DTL-DP, we demonstrate feature transferability by showing t-SNE embeddings in Figure 6. The blue points are non-defective files, and the red are defective ones[5]. We observe the following: (1) The target instances are not discriminated very well, using either the traditional manual features or the TCA+ improved manual features or the semantic features extracted from ASTs, while with our approach, the points are discriminated much better. (2) With the other three approaches the categories between source and target projects are not well aligned, while with our approach, the categories between the projects are more consistent. These conclusions are derived from the intra- and inter-class distances of the two categories in Figures 6 and 7. They are visually apparent.

Our method did not perform as well as the comparison methods on some projects, such as *ant*, which is likely caused by large variance in file sizes. While the sizes of *ant-1.5* and *ant-1.6* are close, there is a marked difference between *ant-1.6* and *ant-1.7*, the former being much smaller than the latter. From Table 3, the performance of our method on *ant1.5->ant1.6* is better than that on *ant1.6->ant1.7*. On the contrary, baseline methods other than *DP-LSTM* perform better for the task *ant1.6->ant1.7*, notably, *Semantic* is dominant, indicating that the semantic feature-based method is more robust to file size variability. Moreover, the *ant* dataset has two shortcomings, the first is that the amount of data is small, cf. Table 2, where the average amount is only 488 files in one project. The second is that

[5]Since we use data augmentation, we have more samples than the three baselines.

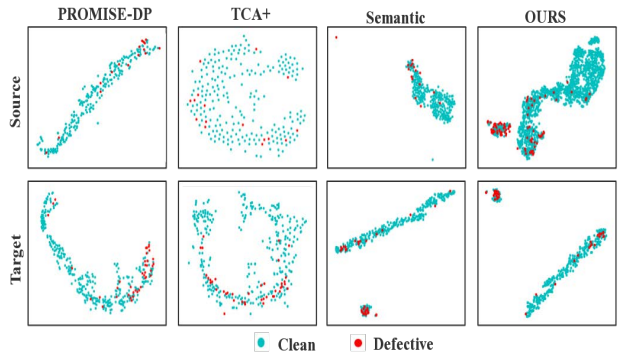


Figure 6: t-SNE mapping of source and target features (WPDP)

the classes are unbalanced, and the proportion of defective files is 13.4%, which is the least of all our projects. This makes training of the deep model more difficult, which leads to the poorer performance of our method on some projects.

Table 4: F-measure of DTL-DP, DBN-CP (DBN), TCA+, DP-LSTM (LSTM) and DP-CNN (CNN) in CPDP

Source	Target	DBN	TCA+	LSTM	CNN	DTL-DP
ant1.6	camel1.4	31.6	29.2	32.1	32.3	**39.5**
jEdit4.1	camel1.4	**69.3**	33.0	31.8	65.1	40.7
camel1.4	ant1.6	**97.9**	61.6	44.8	60.7	59.1
poi3.0	ant1.6	47.8	59.8	38.6	53.2	**69.3**
camel1.4	jEdit4.1	**61.5**	53.7	39.4	54.7	53.1
log4j1.1	jEdit4.1	50.3	41.9	38.9	42.3	**63.9**
jEdit4.1	log4j1.1	64.5	57.4	57.4	65.6	**78.3**
lucene2.2	log4j1.1	61.8	57.1	57.8	63.2	**79.4**
lucene2.2	xalan2.5	55.0	53.0	68.0	54.0	**68.9**
xerces1.3	xalan2.5	57.2	58.1	67.6	56.2	**68.6**
xalan2.5	lucene2.2	59.4	56.1	75.0	62.1	**78.3**
log4j1.1	lucene2.2	69.2	52.4	75.0	66.3	**76.9**
xalan2.5	xerces1.3	38.6	39.4	34.0	39.1	**40.0**
ivy2.0	xerces1.3	**42.6**	39.8	26.1	42.1	42.0
xerces1.3	ivy2.0	45.3	40.9	26.4	46.7	**47.2**
synapse1.2	ivy2.0	**82.4**	38.3	26.1	37.1	49.4
ivy1.4	synapse1.1	48.9	34.8	45.1	49.1	**54.5**
poi2.5	synapse1.1	42.5	37.6	43.5	43.6	**59.7**
ivy2.0	synapse1.2	43.3	57.0	53.0	45.6	**62.0**
poi3.0	synapse1.2	51.4	54.2	50.3	53.2	**62.3**
synapse1.2	poi3.0	66.1	65.1	78.5	67.1	**82.7**
ant1.6	poi3.0	61.9	34.3	78.5	62.7	**82.7**
Average		56.8	47.9	49.5	52.8	**61.8**

5.2 RQ2: How does DTL-DP compare to feature-based Machine Learning and AST-based deep learning methods, in CPDP?

Here we compare to TCA+ and DBN-CP, instead of PROMISE and *Semantic*, as the baseline approaches, as explained in Sect. 4.2. Again, guided by prior work, we conducted 22 sets of CPDP experiments. In each we randomly select two project versions from two different projects, one as a training set and the other as a test set.

Table 4 presents the F-measure results of DTL-CP and the 4 baseline approaches. The highest F-measure values are in bold. On average, the F-measure of our approach in CPDP is 0.618, and the *DBN-CP*, *TCA+*, *DP-LSTM* and *DP-CNN* achieve 0.568, 0.479, 0.495 and 0.528. Thus, DTL-DP outperforms them by 8.8%, 29.0%, 24.8% and 15.5%, respectively. In addition, we found that for projects *log4j1.1* and *poi3.0* our method does better than the corresponding WPDP best performing method.

> Our proposed DTL-DP shows significant improvements on the state-of-the-art in cross-project defect prediction.

5.2.1 Case Study: CPDP Discrimination of DTL-DP. Our method is more obviously dominant in CPDP than in WPDP. A possible reason for that is that deep transfer learning makes the distributions of the training and test samples more similar in feature space. Another reason might be the superior ability of deep models to represent features, enabling the model to obtain more transferable features from the images. To gain more insight, we choose the task *poi3.0* → *ant1.6*, and show the t-SNE embeddings in Figure 7. We make similar observations as in the RQ1 case study, that (1) the target instances are more easily discriminated with our approach, and (2) the target instances can be better discriminated with the source classifier. This implies that our approach can learn more transferable features for more effective defect prediction.

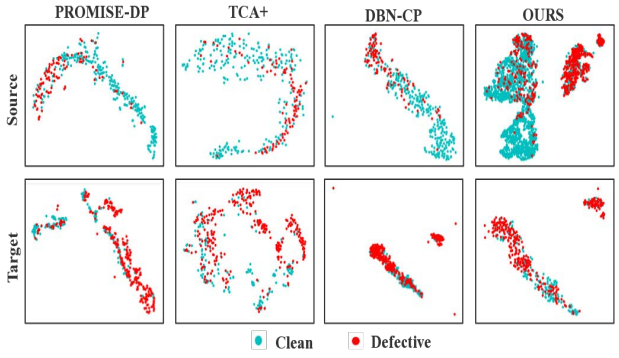


Figure 7: t-SNE mapping of source and target features (CPDP)

5.3 RQ3: How much does each of the three mechanisms, i.e., *data augmentation*, *transfer learning* and *self-attention mechanism*, contribute to DTL-DP's performance?

To find out the specific contributions of the three parts to defect prediction, we conducted further experiments. We built the original AlexNet model for binary classification and use it as the *Base*. *+TL*, *+Attention* and *+DataAug* are three new baselines built by adding to the base AlexNet one of three mechanisms (transfer learning, self attention and data augmentation), respectively. It should be noted

that, in addition to *+DataAug*, we use images generated by one sequence of R, G, and B as training and test sets in each experiment. Therefore, the experimental results of *Base*, *+TL* and *+Attention* in Tables 5 and 6 are average results obtained over the 6 different RGB permutations, as described above.

Table 5: Contributions of the three mechanisms to WPDP

Project	Version	Base	+TL	+Atten	+Aug	DTL-DP
ant	1.5->1.6	66.0	67.6	66.4	67.7	**70.0**
	1.6->1.7	52.5	55.2	52.4	59.1	**59.3**
camel	1.2->1.4	41.5	38.1	**41.6**	40.3	40.6
	1.4->1.6	41.2	39.6	**42.3**	38.4	38.5
jEdit	3.2->4.0	48.4	**50.5**	49.7	48.5	49.4
	4.0->4.1	62.9	64.9	64.5	67.9	**68.7**
log4j	1.0->1.1	68.8	67.9	**69.2**	65.8	68.8
lucene	2.0->2.2	66.7	77.3	77.8	76.1	**78.3**
	2.2->2.4	74.9	71.3	76.9	75.8	**77.6**
xalan	2.4->2.5	75.7	75.4	75.8	77.8	**79.4**
xerces	1.2->1.3	80.9	82.2	80.0	78.8	**82.0**
ivy	1.4->2.0	82.0	82.4	82.7	81.2	**82.9**
synapse	1.0->1.1	52.5	48.4	51.1	53.0	**54.7**
	1.1->1.2	61.8	60.8	61.2	61.6	**65.9**
poi	1.5->2.5	66.3	59.3	65.4	61.4	**68.5**
	2.5->3.0	41.0	**44.9**	42.0	43.2	42.6
Average		61.4	61.6	62.4	62.3	**64.2**

Table 6: Contributions of the three mechanisms to CPDP

Source	Target	Base	+TL	+Atten	+Aug	DTL-DP
ant1.6	camel1.4	38.3	37.9	36.5	37.7	**39.5**
jEdit4.1	camel1.4	36.8	39.4	39.5	37.1	**40.7**
camel1.4	ant1.6	**67.0**	66.7	65.3	59.6	59.1
poi3.0	ant1.6	60.6	62.7	63.4	59.3	**69.3**
camel1.4	jEdit4.1	**66.3**	65.7	65.2	64.7	53.1
log4j1.1	jEdit4.1	59.9	59.9	59.2	59.4	**63.9**
jEdit4.1	log4j1.1	75.4	**80.0**	77.4	74.6	78.3
lucene2.2	log4j1.1	75.9	75.0	75.3	78.3	**79.4**
lucene2.2	xalan2.5	66.1	66.4	65.0	64.7	**68.9**
xerces1.3	xalan2.5	51.0	63.8	62.6	66.2	**68.6**
xalan2.5	lucene2.2	73.0	75.1	71.0	76.9	**78.3**
log4j1.1	lucene2.2	71.3	74.5	74.0	72.7	**76.9**
xalan2.5	xerces1.3	38.7	35.5	39.3	37.3	**40.0**
ivy2.0	xerces1.3	41.3	40.8	39.8	37.8	**42.0**
xerces1.3	ivy2.0	**51.8**	50.0	50.4	50.0	47.2
synapse1.2	ivy2.0	**52.1**	49.0	50.9	47.6	49.4
ivy1.4	synapse1.1	55.1	58.1	**55.3**	53.6	54.5
poi2.5	synapse1.1	50.8	54.1	51.6	54.5	**59.7**
ivy2.0	synapse1.2	60.4	**62.2**	61.9	61.1	62.0
poi3.0	synapse1.2	60.0	**63.2**	61.3	61.2	62.3
synapse1.2	poi3.0	77.0	80.4	78.1	81.7	**82.7**
ant1.6	poi3.0	77.7	66.2	70.0	81.2	**82.7**
Average		59.4	60.3	59.7	59.9	**61.8**

Table 7: Time cost of the three mechanisms: defect visualization (Visualiz.), self-attention (Attent.) and transfer learning (TL)

Project	Time (s)		
	TL	Attent.	Visualiz.
ant	45.04	1.31	4.59
camel	96.55	2.42	7.85
jEdit	59.15	2.66	4.97
log4j	13.71	1.31	1.76
lucene	37.39	1.86	3.54
xalan	94.44	2.71	8.23
xerces	51.17	2.14	2.75
ivy	29.02	1.42	3.05
synapse	20.42	1.26	2.09
poi	38.65	1.57	2.21
Average	48.56	1.87	4.10

To discern the contribution of each mechanism we compare the performance of these baseline methods in WPDP and CPDP. From the results in Table 5 and Table 6, we observe that the three mechanisms each contribute substantially to the accuracy of DTL-DP, in both WPDP and CPDP. In terms of the F-measure, *transfer learning* contributes the least improvement, and *self-attention* and *data augmentation* contribute similarly to the final result. But in CPDP, *transfer learning* contributes to the F-measure the most. This is likely because of the applicability of transfer learning in the CPDP setting, and is, in a way, a validation of the approach.

We also noted the time cost for the 3 mechanisms in our proposed DTL-DP. Table 7 shows the result. The most time spent during data augmentation is on converting code into images, i.e., source code visualization. For transfer learning, most time is spent on the MK-MMD calculation, and for self-attention, on the calculation of the attention layer in Fig. 4. E.g., for the project *ant*, Table 3 shows two sets of WPDP experiments, *ant 1.5 → 1.6* and *ant 1.6 → 1.7*. On average, it takes 45.04 seconds, 13.01 seconds and 4.59 seconds for the 3 parts, respectively, for both the training data and the test data. Transfer learning takes the longest time, more than the sum of the other two. This is because of the large number of matrices needed to calculate MMD with the kernel function. The least time cost is incurred by the attention mechanism, and its contribution to WPDP is the largest of the three, i.e., it is most cost-effective.

> The three mechanisms all contribute toward the accuracy of our proposed end-to-end framework, in WPDP and CPDP. *Self-attention* has the greatest contribution to WPDP, and *transfer learning* contributes most to CPDP.

6 THREATS TO VALIDITY

Threats to internal validity come from experimental errors and the replication of the baseline methods. In order to compare and analyze the deep learning-based defect prediction techniques, we compare our proposed DTL-DP method with *Sementic*, *DBN-CP*, *DP-LSTM* and *DP-CNN*. In addition, our method is also compared with the transfer learning-based method TCA+, which is the state-of-the-art CPDP technique. Since the original implementations were not

available, we re-implemented our own versions of the baselines. Although we strictly follow the procedures described in their work, our new implementations may not completely restore all of their original implementation details. And the randomness of the deep learning-based approach also makes the results of our implemented experiments different from the original. Since we have removed the entries in the *PROMISE* dataset that cannot retrieve the corresponding source files, our re-implemented experimental results may not be consistent with the original baselines.

The external threat to the validity of the results lies in the generalizability of the results. We have tested our method on 10 open source Java projects, including 14600 files. Defect predictions for instances of other languages, such as C, C++, etc., need to be validated by additional experiments in the future.

Threats to construct validity depend on the appropriateness of the evaluation measurement. The F-measure is used as our main evaluation measure, which has been applied in many previous efforts to evaluate defect prediction comprehensively.

7 CONCLUSIONS AND FUTURE WORK

Here we made two main contributions, code visualization for defect prediction and an improved deep transfer learning model. Our experimental results on 10 open source projects show that deep learning can be effectively applied directly for defect prediction after applying visualization methods to the code. Specifically, our approach, DTL-DP, performs at the top of the range of state-of-the-art WPDP approaches. For CPDP, DTL-DP improves on the state-of-the-art technique TCA+, built on traditional features, by 29.0% (in the F-measure). It also bests the deep learning-based approaches DBN-CP, DP-LSTM and DP-CNN by 8.8%, 24.8% and 15.5%, respectively.

DTL-DP still has some limitations. For some projects, a problem of negative transfer occurs, resulting in a worse prediction than a direct prediction. Large differences between two projects can cause such negative transfer. Reducing the impact of negative transfer is one of the problems to be solved in the future. Additionally, the amount of training data we had was small and class imbalance is inherent in software defect prediction. Both can be improved with more data, which we plan to obtain in the future.

ACKNOWLEDGEMENTS

This work is partially supported by Zhejiang Natural Science Foundation(LY19F020025), National Natural Science Foundation of China (61502423, 61572439, 61702534), Zhejiang University Open Fund(2018KFJJ07), Signal Recognition Based on GAN, Deep Learning for Enhancement Recognition Project, Zhejiang Science and Technology Plan Project (LGF18F030009, 2017C33149), and Zhejiang Outstanding Youth Fund (LR19F030001).

REFERENCES

[1] Romi Satria Wahono. A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.

[2] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[3] Lech Madeyski and Marian Jureczko. Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[4] Thomas Shippey, David Bowes, and Tracy Hall. Automatically identifying code features for software defect prediction: Using ast n-grams. *Information and Software Technology*, 106:142–160, 2019.

[5] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[6] Karim O Elish and Mahmoud O Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[7] Puja Ahmad Habibi, Victor Amrizal, and Rizal Broer Bahaweres. Cross-project defect prediction for web application using naive bayes (case study: Petstore web application). In *2018 International Workshop on Big Data and Information Security (IWBIS)*, pages 13–18. IEEE, 2018.

[8] Michael J Siers and Md Zahidul Islam. Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. *Information Systems*, 51:62–71, 2015.

[9] R Jayanthi and Lilly Florence. Software defect prediction techniques using metrics based on neural network classifier. *Cluster Computing*, pages 1–12, 2018.

[10] Yi Liu, Yu Fan, and Junghui Chen. Flame images for oxygen content prediction of combustion systems using dbn. *Energy & Fuels*, 31(8):8776–8783, 2017.

[11] Yi Liu, Chao Yang, Zengliang Gao, and Yuan Yao. Ensemble deep kernel learning with application to quality prediction in industrial polymerization processes. *Chemometrics and Intelligent Laboratory Systems*, 174:15–21, 2018.

[12] Qi Xuan, Binwei Fang, Yi Liu, Jinbao Wang, Jian Zhang, Yayu Zheng, and Guanjun Bao. Automatic pearl classification machine based on a multistream convolutional neural network. *IEEE Transactions on Industrial Electronics*, 65(8):6538–6547, 2018.

[13] Qi Xuan, Haoquan Xiao, Chenbo Fu, and Yi Liu. Evolving convolutional neural network and its application in fine-grained visual categorization. *IEEE Access*, 6:31110–31116, 2018.

[14] Qi Xuan, Zhuangzhi Chen, Yi Liu, Huimin Huang, Guanjun Bao, and Dan Zhang. Multiview generative adversarial network and its application in pearl classification. *IEEE Transactions on Industrial Electronics*, 66(10):8244–8252, 2019.

[15] Qi Xuan, Fuxian Li, Yi Liu, and Yun Xiang. MV-C3D: A spatial correlated multi-view 3d convolutional neural networks. *IEEE Access*, 7:92528–92538, 2019.

[16] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 297–308. IEEE, 2016.

[17] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 318–328. IEEE, 2017.

[18] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*, 2018.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I Jordan. Learning transferable features with deep adaptation networks. *arXiv preprint arXiv:1502.02791*, 2015.

[21] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. *arXiv preprint arXiv:1805.08318*, 2018.

[22] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 279–289. IEEE Press, 2013.

[23] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23. ACM, 2008.

[24] Raimund Moser, Witold Pedrycz, and Giancarlo 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*, pages 181–190. ACM, 2008.

[25] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE, 2013.

[26] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[27] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[28] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[29] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[30] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *null*, pages 364–373. IEEE, 2007.

[31] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[32] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 19. ACM, 2010.

[33] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321. ACM, 2011.

[34] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[35] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.

[36] Steffen Herbold, Alexander Trautsch, and Jens Grabowski. A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, 44(9):811–833, 2018.

[37] Chao Liu, Dan Yang, Xin Xia, Meng Yan, and Xiaohong Zhang. A two-phase transfer learning model for cross-project defect prediction. *Information and Software Technology*, 107:125–136, 2019.

[38] Jinyin Chen, Keke Hu, Yitao Yang, Yi Liu, and Qi Xuan. Collective transfer learning for defect prediction. *Neurocomputing*, 2019. doi:https://doi.org/10.1016/j.neucom.2018.12.091.

[39] Jinyin Chen, Yitao Yang, Keke Hu, Qi Xuan, Yi Liu, and Chao Yang. Multiview transfer learning for software defect prediction. *IEEE Access*, 7:8901–8916, 2019.

[40] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. Cross-project defect prediction models: L'union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 164–173. IEEE, 2014.

[41] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. volume 14, pages 540–578. Springer, 2009.

[42] Fayola Peters, Tim Menzies, and Andrian Marcus. Better cross company defect prediction. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 409–418. IEEE Press, 2013.

[43] Xin Xia, LO David, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977, 2016.

[44] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

[45] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 382–391. IEEE, 2013.

[46] Sinno Jialin Pan, Ivor W Tsang, James T Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2011.

[47] Blaine A Price, Ronald M Baecker, and Ian S Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266, 1993.

[48] Thomas Ball and Stephen G Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[49] Sarita Bassil and Rudolf K Keller. Software visualization tools: Survey and analysis. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 7–17. IEEE, 2001.

[50] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[51] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th international workshop on principles of software evolution*, pages 37–42. ACM, 2001.

[52] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[53] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223. ACM, 2005.

[54] Abien Fred Agarap and Francis John Hill Pepito. Towards building an intelligent anti-malware system: a deep learning approach using support vector machine (svm) for malware classification. *arXiv preprint arXiv:1801.00318*, 2017.

[55] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, 2016.

[56] Rahul Krishna and Tim Menzies. Bellwethers: A baseline method for transfer learning. *IEEE Transactions on Software Engineering*, 2018.

[57] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *ICANN*, 2018.

[58] Wenyuan Dai, Qiang Yang, Gui Rong Xue, and Yong Yu. Boosting for transfer learning. In *International Conference on Machine Learning*, pages 193–200, 2007.

[59] Chang Wan, Rong Pan, and Jiefei Li. Bi-weighting domain adaptation for cross-language text classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1535, 2011.

[60] Yonghui Xu, Sinno Jialin Pan, Hui Xiong, Qingyao Wu, Ronghua Luo, Huaqing Min, and Hengjie Song. A unified framework for metric transfer learning. *IEEE Trans. Knowl. Data Eng.*, 29(6):1158–1171, 2017.

[61] Chetak Kandaswamy, Luís M. Silva, Luís A. Alexandre, and Jorge M. Santos. Deep transfer learning ensemble for classification. In Ignacio Rojas, Gonzalo Joya, and Andreu Catala, editors, *Advances in Computational Intelligence*, pages 335–348, Cham, 2015. Springer International Publishing.

[62] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[63] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[64] Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, and Mario Marchand. Domain-adversarial neural networks. *arXiv preprint arXiv:1412.4446*, 2014.

[65] Yaroslav Ganin and Victor Lempitsky. Unsupervised domain adaptation by backpropagation. *arXiv preprint arXiv:1409.7495*, 2014.

[66] Jing Zhang, Wanqing Li, and Philip Ogunbona. Joint geometrical and statistical alignment for visual domain adaptation. *arXiv preprint arXiv:1705.05498*, 2017.

[67] Eric Tzeng, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell. Deep domain confusion: Maximizing for domain invariance. *arXiv preprint arXiv:1412.3474*, 2014.

[68] Mingsheng Long, Yue Cao, Zhangjie Cao, Jianmin Wang, and Michael I Jordan. Transferable representation learning with deep adaptation networks. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[69] Arthur Gretton, Dino Sejdinovic, Heiko Strathmann, Sivaraman Balakrishnan, Massimiliano Pontil, Kenji Fukumizu, and Bharath K Sriperumbudur. Optimal kernel choice for large-scale two-sample tests. In *Advances in neural information processing systems*, pages 1205–1213, 2012.

[70] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I Jordan. Deep transfer learning with joint adaptation networks. *arXiv preprint arXiv:1605.06636*, 2016.

[71] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[72] Kesav Kancherla and Srinivas Mukkamala. Image visualization based malware detection. In *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 40–44. IEEE, 2013.

[73] Xin Zhou, Jianmin Pang, and Guanghui Liang. Image classification for malware detection using extremely randomized trees. In *2017 11th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 54–59. IEEE, 2017.

[74] John Murphy. An overview of convolutional neural network architectures for deep learning. 2016.

[75] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[76] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.