The Journal of Systems & Software 197 (2023) 111557

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Efficient transformer with code token learner for code clone detection[☆]

Aiping Zhang [a], Liming Fang [a,b,*], Chunpeng Ge [a], Piji Li [a], Zhe Liu [a]

[a] Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, PR China
[b] Nanjing University of Aeronautics and Astronautics, Shenzhen Research Institute, Shenzhen, Guangdong, PR China

ARTICLE INFO

Article history:
Received 27 March 2022
Received in revised form 2 November 2022
Accepted 10 November 2022
Available online 14 November 2022

Keywords:
Code clone detection
Code token learner
Efficient transformer

ABSTRACT

Deep learning techniques have achieved promising results in code clone detection in the past decade. Unfortunately, current deep learning-based methods rarely explicitly consider the modeling of long codes. Worse, the code length is increasing due to the increasing requirement of complex functions. Thus, modeling the relationship between code tokens to catch their long-range dependencies is crucial to comprehensively capture the information of the code fragment. In this work, we resort to the Transformer to capture long-range dependencies within a code, which however requires huge computational cost for long code fragments. To make it possible to apply Transformer efficiently, we propose a code token learner to largely reduce the number of feature tokens in an automatic way. Besides, considering the tree structure of the abstract syntax tree, we present a tree-based position embedding to encode the position of each token in the input. Apart from the Transformer that captures the dependency within a code, we further leverage a cross-code attention module to capture the similarities between two code fragments. Our method significantly reduces the computational cost of using Transformer by 97% while achieves superior performance with state-of-the-art methods. Our code is available at https://github.com/ArcticHare105/Code-Token-Learner.

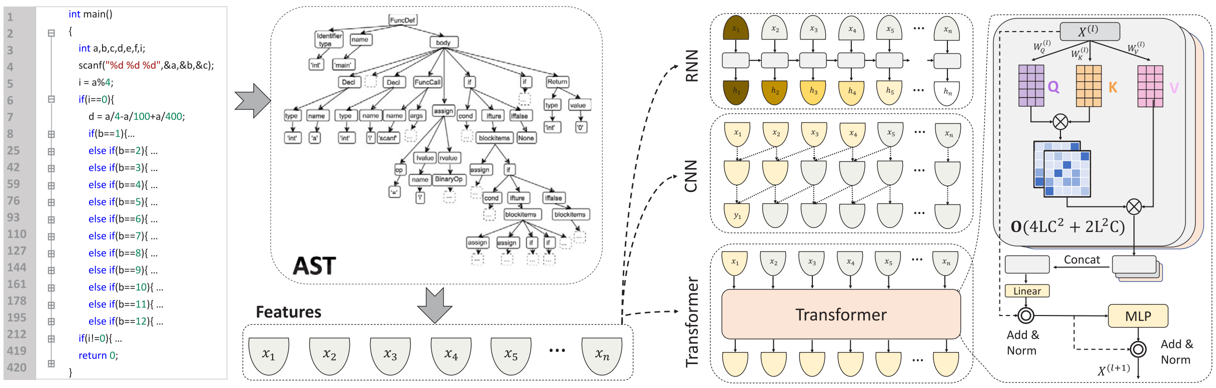
1. Introduction

Code clone detection aims to detection code fragments that are similar in the light of some similarity definitions. It is meaningful for the software development life-cycle and fundamental in many software engineering tasks, *e.g.*, code classification (Kamiya et al., 2002), bug detection, and malicious code detection (Jiang et al., 2007). Recently, a substantial amount of research effort has been devoted to detect clones. Some early methods used handcrafted lexical and syntactic program features to identify similar code pairs, showing promising performance in detecting lexically and syntactically similar code pairs. However, for detecting structurally flexible (*e.g.*, adding or removing several statements) and semantically similar code pairs, they obtain compromising performance.

To handle these difficult code clones, recently, more and more methods (Allamanis et al., 2017; Tufano et al., 2018; Fang et al., 2020; Feng et al., 2020; Guo et al., 2020) seek to exploit the powerful ability of deep neural networks to detect either syntactically similar or semantically similar code fragments. Existing methods usually adopt a multi-stage framework. The first stage is to transform code fragments, basically functions, into abstract syntax trees (ASTs) (White et al., 2016; Wei and Li, 2017; Zhang et al., 2019) or program dependence graphs (PDGs) (Zhao and Huang, 2018). At the second stage, they usually use convolutional neural networks (CNNs) (White et al., 2016), recurrent neural networks (RNNs) (Zhang et al., 2019) or graph neural networks (GNNs) (Wang et al., 2020a) to extract feature representations for each code fragment. Finally, they calculate the similarity of two code fragments.

Intuitively, code fragments are with different lengths. Especially, there are many long code fragments because of more functional demanding. In BigCloneBench (Svajlenko et al., 2014) dataset, the long code snippets (over 500 nodes) account for about 10% of the total dataset. Besides, there are some very long functions that have more than 10,000 nodes. However, despite the powerful ability of CNNs, RNNs and GNNs on feature extraction, these networks inherently lack the ability of capturing contextual information with full receptive field. The existing methods based on the above networks cannot model long-range dependencies between code tokens, constituting an important reason for the decrease in the accuracy of code clone detection. Recently, Transformer (Vaswani et al., 2017) based methods have demonstrated superior ability to model long-term dependencies and achieved impressive performance in natural language processing (Devlin et al., 2018) and image-related tasks (Dosovitskiy et al., 2020). However, these methods are usually adopted to the entity with relatively small number of tokens. For example,

☆ Editor: Fabio Palomba.
* Corresponding author.
E-mail addresses: zhangap@nuaa.edu.cn (A. Zhang), fangliming@nuaa.edu.cn (L. Fang), gecp@nuaa.edu.cn (C. Ge), lipiji.pz@gmail.com (P. Li), zhe.liu@nuaa.edu.cn (Z. Liu).



```
int main()
{
  int a,b,c,d,e,f,i;
  scanf("%d %d %d",&a,&b,&c);
  i = a%4;
  if(i==0){
    d = a/4-a/100+a/400;
    if(b==1){...
    else if(b==2){ ...
    else if(b==3){ ...
    else if(b==4){ ...
    else if(b==5){ ...
    else if(b==6){ ...
    else if(b==7){ ...
    else if(b==8){ ...
    else if(b==9){ ...
    else if(b==10){ ...
    else if(b==11){ ...
    else if(b==12){ ...
  if(i!=0){ ...
  return 0;
}
```

Fig. 1. An example that a long code is fed into different neural networks.

ViT (Dosovitskiy et al., 2020) take 16 × 16 tokens as input, while the sequence model GPT-3 (Brown et al., 2020) can only handle a sequences of a maximum length of 2,048 tokens (Li et al., 2019). This phenomenon makes it infeasible to transfer the success of existing Transformer-based methods to code clone detection.

In this paper, we propose a feasible way to apply the Transformer for efficient code clone detection. Specifically, our method takes the abstract syntax tree as input. A corresponding position embedding strategy is adopted to fit the tree structure of the abstract syntax tree. To reduce the computational cost, we propose to first reduce the number of nodes of the abstract syntax tree. We utilize a code token learner, which consists of the Graph Convolutional Networks (GCNs) and attention mechanism, to capture the tree structure of ASTs. The code token learner outputs an assignment matrix to aggregate nodes into a small number of code tokens. For the Transformer, we introduce an additional similarity token to gradually capture long-range dependencies within codes. Finally, a cross-code attention module is introduced to capture cross-code similarity.

The contributions of this paper are:

- We introduce Transformer into code clone detection to model long-range dependencies within codes to improve detection performance, especially for the long codes.
- We propose a code token learner to significantly reduce the size of the input and decrease the computational cost of using Transformers by 97% without requiring any manual design rules.
- We propose the tree-based position embedding to present the position of nodes in abstract syntax trees.
- Compared with existing methods, the proposed method yields improvement of 0.9% and 2.8% in terms of F1 score on two benchmarks, *i.e.*, BigCloneBench (Svajlenko et al., 2014) and OJClone (Mou et al., 2016), respectively.

The remainder of this paper is structured as follows: Section 2 introduces the background knowledge. We present the details of our approach in Section 3. We present our approach settings and propose some research questions in Section 4. In Section 5, we evaluate our approach and analyze its performance. Section 8 lists the related works. In Section 7, we show some potential threats of our method. Finally, in Section 9, we make a conclusion about our work and some possible future improvements to our work.

2. Backgroud and motivation

2.1. AST-based methods and their limitations

Abstract Syntax Tree (AST) is an abstract representation of the syntactic structure of the source code (Baxter et al., 1998). As shown in Fig. 1, it represents the syntactic structure of the programming language as a tree, with each node in the tree representing a structure in the source code. The syntax is "abstract" because the syntax here does not represent every detail that occurs in the real syntax. Therefore, the abstract syntax tree is a tree structure and can be also represented as a graph. To take advantage of the AST, tree-based methods (White et al., 2016; Mou et al., 2016; Wei and Li, 2017; Zhang et al., 2019) have been developed to take ASTs as input. There is a recent method (Wang et al., 2020a) formulate the abstract syntax tree as a graph, which is then processed by a graph neural network.

2.2. Revisiting transformer

The Transformer block (Vaswani et al., 2017) consists of alternating layers of multi-headed self-attention (MSA) and multilayer perceptron (MLP) blocks. Layernorm (LN) is applied before every block, and residual connections (Wang et al., 2019; Baevski and Auli, 2018) after every block. For the self-attention, three matrices $\boldsymbol{Q}$ (Query), $\boldsymbol{K}$ (Key), and $\boldsymbol{V}$ (Value) come from the same input by linear projection. First, it calculates the dot product between $\boldsymbol{Q}$ and $\boldsymbol{K}$. Then, to prevent the result from being too large, the matrix $\boldsymbol{QK}^T$ is divided by a scale $\sqrt{d_k}$, where d_k is a dimension of the query and key vectors. The results are normalized to the probability distribution by softmax operation, and then multiplied by the matrix $\boldsymbol{V}$ to obtain the representation of the sum of weights. This operation can be expressed as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{QK}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{1}$$

The whole block of Transformer can be formulated as:

$$\begin{aligned} \mathbf{z}'_\ell &= \text{MSA}\left(\text{LN}\left(\mathbf{z}_{\ell-1}\right)\right) + \mathbf{z}_{\ell-1} \\ \mathbf{z}_\ell &= \text{MLP}\left(\text{LN}\left(\mathbf{z}'_\ell\right)\right) + \mathbf{z}'_\ell \end{aligned} \tag{2}$$

where l denotes the lth layer, MSA is the multi-headed fashion of Eq. (1). Multi-head attention is an attention mechanism module that can run several times in parallel within an attention mechanism. The independent attention outputs allows for attending to parts of the sequence differently. Specifically, the matrix $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are the same and the attention is performed multiple times, and then the output heads are concatenated as the final output h. This process can be formulated as

$$\text{head}_i = \text{Attention}(\boldsymbol{QW}_i^Q, \boldsymbol{KW}_i^K, \boldsymbol{VW}_i^V) \tag{3}$$

$$h = \text{concat}([\text{head}_1, \ldots, \text{head}_n])\boldsymbol{W} \tag{4}$$

where $\boldsymbol{W}$ is a weight matrix to linearly fuse the outputs of multiple attention heads.

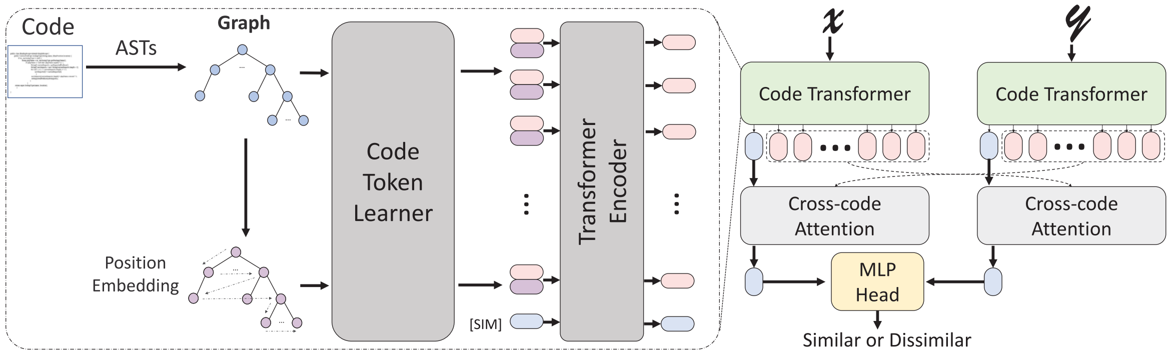


Fig. 2. The overview of our method. **Left:** code Transformer. First, source code fragments are parsed into ASTs. We follow (Wang et al., 2020a) to represent the ASTs as graphs. We adopt depth first search and breadth first search to define tree position embeddings for the ASTs. A code token learner takes the graph representation as input and outputs an assignment matrix, which is used to learn a compact set of code tokens and corresponding position embeddings. A [SIM] token is appended on the code tokens and then fed into the Transformer. **Right:** code clone detection with code Transformer. Given two code fragments, we feed them into individual code Transformers to outputs their [SIM] tokens and code tokens. The [SIM] tokens are attended to the code tokens in a cross-code manner. The output [SIM] tokens are directly used to calculate the similarity of the two codes.

2.3. Motivating example

We present an example here to verify the above statements and highlight our motivation. In Fig. 1, we show a long code with 420 lines, whose function is to calculate the day of the week given year, month and day. After transforming this long code into the abstract syntax tree, which has more than one thousand nodes, the corresponding number of features is also more than one thousand. Usually, these extracted features are forwarded into recurrent neural networks or convolutional neural networks. However, there are some intrinsic drawbacks. For the RNN-based methods, as show in Fig. 1, their receptive fields are theoretically the whole sequence. However, the long-range forgetting is a non-negligible problem. Even if the LSTM (Hochreiter and Schmidhuber, 1997) or GRU (Cho et al., 2014) is tailored to address this issue, there still exists information forgetting when the length of the input sequence is too long. In contrast to the RNN-based methods, the CNN-based methods are known to have only limited receptive fields. For example, in Fig. 1, we demonstrate the case with kernel size of 2 and stride 1. After two convolutional layers, the receptive field is only 4, which is insufficient to capture long-range dependencies. Moreover, even the well-known ResNet-50 (He et al., 2016) (ResNet with 50 layers) just has the receptive field of 483, but with about 6.96 billion FLOPs (floating point operations). This drawback hinders the CNN-based methods to capture long-range dependencies within codes, leading to compromising performance. To tackle the above issue, we can resort to the Transformer, which is good at modeling long-range dependencies. As stated above, the Transformer contains two parts, namely multi-head self attention (MSA) and multi-layer perception (MLP). Given the input with L tokens, the FLOPs of the MSA is $\mathbf{O}(4LC^2+2L^2C)$, where C is the dimension of the query, key and value. If we take an input with size of 1000×64, the FLOPs would be 144 million for one Transformer layer, which is quite large for portable devices or embedded devices, much less the input with more than 10 thousands nodes in the BigCloneBench dataset, which would lead to more than 12 billion FLOPs.

There are some existing methods (Zhang et al., 2019; Hua and Liu, 2021) tackle this issue by splitting the large AST into a series of small ASTs. However, this setting would destroy the original structure of the code. Besides, the manual splitting cannot guarantee end-to-end training and apply to all programming languages, leading to sub-optimal performance. Based on the above observation, we need to reduce the size of the abstract syntax tree, so as to makes it possible to leverage Transformer for detecting long codes.

3. Methodology

In this section, we first introduce an overview of our proposed approach. Fig. 2 shows an illustration of our method. Specifically, our method is based on Transformer (Vaswani et al., 2017), on which a siamese network is built. The motivation of introducing Transformer originates from the observation that existing methods usually lack the ability of modeling long-range dependencies within a code fragments, leading to coarse and shallow understandings about the whole code, especially for those codes with complex structures. Taking account of the phenomenon that the commonly used abstract syntax trees (ASTs) usually contain a large number of nodes, it is infeasible to directly adopt a Transformer-based architecture to ASTs, which would result in significant computation and memory overhead. To address this issue, we propose an attention-based code learner (Section 3.2), which is composed of graph convolutional networks and attention mechanisms to extract informative parts of code fragments and largely reduce the token number for Transformer. After that, the learned code tokens are forwarded into a siamese Transformer network (Section 3.3), which includes an additional representation token to gradually combine information from the learned code tokens into a discriminative representation of the whole code. A cross-code attention mechanism is finally introduced to attend to similar parts across codes for code clone detection.

AST parsing and embedding. Since the code fragment cannot be directly taken as the input, existing methods usually transform it into some abstract feature representations. Abstract syntax tree is a kind of tree target at representing the abstract syntactic structure of the source code (Baxter et al., 1998). An abstract syntax tree contains multiple nodes and edges, each node on an AST is usually represented by a feature vector learned by word embedding, and each edge represents the close relation between two nodes. Our method follows FA-AST (Wang et al., 2020a) to construct the AST representation, that is, our method uses the name of variable and function names to construct the representation of an AST's node. Given the AST $\mathcal{T} = \{\mathcal{V}, \mathcal{E}\}$ of a code, where $\mathcal{V}$ is the set of node representations with $|\mathcal{V}| = N$, and $\mathcal{E}$ is the set of edges, the set of representations of all nodes can be formulated as $\boldsymbol{H} \in \mathbb{R}^{N\times F}$.

3.1. Tree position embedding

Because of the permutation-invariant characteristic of self-attention mechanism, position embedding is necessary in

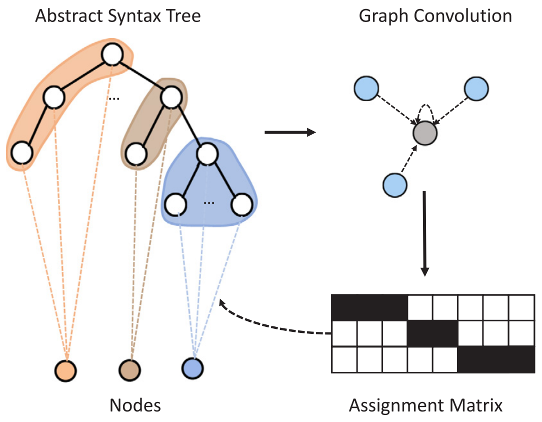


Fig. 3. An overview of the code token learner.

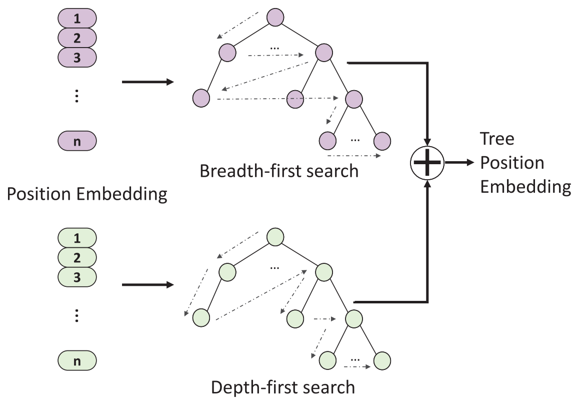


Fig. 4. Illustration of the tree position embedding.

Transformer-based architectures to incorporate position information. For a sentence or an image of regular structures, it is easy to give meaningful position embeddings, such as the sinusoidal position embedding (Devlin et al., 2018), for their tokens according to the regular relative positions. In contrast, the source code has different structures with images, videos and languages which has regular structures. Therefore, the commonly used position embedding for images, videos and languages cannot be directly transferred to the source code.

In light of this observation, we propose a tree position embedding strategy. Specifically, as shown in Fig. 4, we randomly initialize two sets of the position embedding $\boldsymbol{P}_0^b \in \mathbb{R}^{N\times F/2}$ and $\boldsymbol{P}_0^d \in \mathbb{R}^{N\times F/2}$. We assign the position embedding to each node of input through depth-first search and breadth-first search, respectively. The final position embeddings $\boldsymbol{P}$ are the concatenation of the two position embeddings. With the tree position embedding, the final input is the combination of the original feature $\boldsymbol{H}$ and the position embedding $\boldsymbol{T} = \boldsymbol{H} + \boldsymbol{P}$.

3.2. Code token learner

As we can see, an AST contains the semantic and structural information of the whole code, however, we would like to argue that existing methods cannot well leverage ASTs. First, the number of nodes (features) of an AST is usually large, which makes it hard to fully capture rich information of the whole code by existing neural networks, such as graph convolutional networks (GCNs) (Wang et al., 2020a) or tree-based LSTM (Wei and Li, 2017). For the GCNs, because of its smoothing nature (Li et al., 2018), it is non-trivial to stack multiple graph convolutional layers, leading to poor ability in capture long-range information of the whole code. Besides, for the RNNs or LSTMs (Hochreiter and Schmidhuber, 1997), AST is usually transformed into a sequence as input, which cannot handle very long sequences because of their forgetting characteristic.

In light of the challenge of capturing long-range information for code clone detection, we propose to introduce Transformer into this task. However, despite the good ability of Transformer in capturing long-range dependency, it usually requires large computational and memory cost, this requirement is inaccessible when we take the large AST as input. To address this issue, as shown in Fig. 3, we propose a code token learner to reduce the number of feature tokens before feeding them into the Transformer. Specifically, given the AST $\mathcal{T}$ of a code, we first leverage a graph convolutional network to capture the intrinsic tree structure of the code. In detail, the graph convolution $\mathcal{G}(\cdot)$ can be formulated as:

$$\mathcal{G}(\boldsymbol{T}) = \widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{T}\boldsymbol{W} \tag{5}$$

where $\boldsymbol{T} \in \mathbb{R}^{N\times F}$ is the input feature and $\boldsymbol{W} \in \mathbb{R}^{F\times F'}$ is the weight matrix, $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ denotes the adjacency matrix, whose element a_{ij} is the weight assigned to the edge (i, j). We set $a_{ij} = 1$ if node i and j are connected in AST and $a_{ij} = 0$ otherwise. Besides, we also use the self-connection, *i.e.*, $\widetilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_N$ and $\widetilde{\boldsymbol{D}} \in \mathbb{R}^{N\times N}$ is the corresponding degree matrix. The matrix multiplication of $\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}$ can be viewed as a symmetric normalization. Combining graph convolution with ReLU σ, we learn an assignment matrix $\boldsymbol{S} \in \mathbb{R}^{N\times N_t}$ by a two-layer GCN as:

$$\boldsymbol{S} = \mathcal{G}(\sigma(\mathcal{G}(\boldsymbol{T}))) \tag{6}$$

where N_t is the number of the input tokens. With the input feature $\boldsymbol{T} \in \mathbb{R}^{N\times F}$, the assignment matrix aggregates the attended features in $\boldsymbol{T}$ into different code tokens $\boldsymbol{C} \in \mathbb{R}^{N_t\times F}$:

$$\boldsymbol{C}_j = \sum_k \frac{exp(S_{kj})}{\sum_i exp(S_{ij})}\boldsymbol{T}_k \tag{7}$$

where $\boldsymbol{C}_j$ represents the j-th row of $\boldsymbol{C}$, *i.e.*, the j-th code token. In this way, we transform the feature $\boldsymbol{T}$ with large number of input tokens into several compact tokens $\boldsymbol{C}$, which would benefit a lot in reducing computation and memory costs for the following Transformer.

3.3. Code transformer

Given the learned compact tokens $\boldsymbol{C}$, we propose a code Transformer encoder to learn code representations. It consists of a Transformer for individual codes and a cross-code attention module for capturing similarity between two code fragments. In this section, we will present the details of our code Transformer.

Similarity token. Motivated by the recent success in introducing different tokens into Transformer (*e.g.*, [CLS] token in BERT (Devlin et al., 2018) and detection tokens in DETR (Carion et al., 2020)), we append a [SIM] token on the learned code tokens $\boldsymbol{C}$, *i.e.*, $\boldsymbol{C}_{new} = \{\boldsymbol{T}_{[SIM]} \parallel \boldsymbol{C}\}$, where $\boldsymbol{T}_{[SIM]}$ is the [SIM] token and $\parallel$ denotes the concatenate operation. The feature $\boldsymbol{C}_{new}$ is fed into the Transformer encoder of L layers, and finally we can obtain the output feature $\boldsymbol{C}_L$.

Cross-code attention. Even if the code Transformer could model the long-range dependencies within a code, it is insufficient to capture the similarity of two codes because there are no interaction between two code fragments. In order to capture the relations between two codes, we propose a cross-code attention strategy. Specifically, given two codes' [SIM] tokens (i.e., the first token in $\boldsymbol{C}_L$) in the outputs of the code Transformer, *i.e.*, $\boldsymbol{C}_L^x[0]$ and $\boldsymbol{C}_L^y[0]$, we utilize them as query tokens to attend to the code tokens of another code. Specifically, the cross-code attention from code x to code y can be represented as:

$$\tilde{\boldsymbol{C}}_x = \mathrm{Attention}(\boldsymbol{C}_L^x[0]\boldsymbol{W}^Q, \boldsymbol{C}_L^y[1:]\boldsymbol{W}^K, \boldsymbol{C}_L^y[1:]\boldsymbol{W}^V) \tag{8}$$

Likewise, the cross-code attention can be also represented with the multi-head attention manner (please refer to Section 2.2). After cross-code attention, we can obtain the refined code token $\tilde{\boldsymbol{C}}_x$ and $\tilde{\boldsymbol{C}}_y$, which contains long-range information of both code x and code y.

3.4. Calculating code similarity

With the refined tokens $\tilde{\boldsymbol{C}}_x$ and $\tilde{\boldsymbol{C}}_y$, we calculate their residual as follow

$$\boldsymbol{R} = abs(\tilde{\boldsymbol{C}}_x - \tilde{\boldsymbol{C}}_y), \tag{9}$$

where $abs(\cdot)$ denotes element-wise absolution operation. Therefore, the similarity between code x and code y is calculated as

$$B = \mathrm{sigmoid}(\mathrm{FC}(\boldsymbol{R})), \tag{10}$$

where $\mathrm{FC}(\cdot)$ is a fully connected layer to map the dimensionality to 1, and $\mathrm{sigmoid}(\cdot)$ enforces the output value to be in between 0 and 1.

Finally, to deal with the imbalance between the positive and negative samples, we utilize the focal loss (Lin et al., 2017) as

$$\mathcal{L}^i = -(1-B)^\gamma \log(B), B = \begin{cases} B^i, \text{ when } y^i = 1 \\ 1 - B^i, \text{ when } y^i = 0 \end{cases} \tag{11}$$

where y^i and B^i represent the ground truth and similarity of the ith code pair in a mini-batch of size N. γ is a tunable focusing parameter, which is set as 2.0 in our method.

Note that, our method now works at the method-level granularity. But we think it is possible to apply our method to larger clones, e.g., file-level clone detection. Actually, file-level clones can be decomposed into a hierarchical organization of method-level fragments. Therefore, we can decompose a file into several methods and aggregate the detection results of these methods into a file-level detection result.

4. Experimental settings

Empirical evaluation of our proposed methods is performed through several experiments. Before showing the experimental results, we first describe some research questions, the used benchmarks, and the overall experimental setup.

4.1. Research questions

Based on the evaluation on the two benchmarks, we target at investigating the following research questions:

- **RQ1: What is the effect of the designed code learner?** With this RQ, we aim to validate the effectiveness and efficiency of the proposed code learner. We will show the performance and computation cost after adding the code learner.
- **RQ2: Is the proposed tree position embedding suitable for the abstract syntax tress?** With this RQ, we would like to compare the proposed tree position embedding with some commonly used position embeddings, so as to verify the claim that the position of a tree structure cannot be captured by regular position embeddings.
- **RQ3: What is a proper setting for a code Transformer?** We would like to conduct some ablation studies to obtain a proper setting for the code Transformer to reach the optimal result.
- **RQ4: Is the cross-attention necessary?** Usually, a siamese network has no interaction between the two parallel sub-networks. With this RQ, we aim to evaluate the necessity of the cross-attention module.
- **RQ5: How does our approach perform in code clone detection compared with state-of-the-art methods?** We aim to compare our method with the state-of-the-art methods to show the effectiveness of our method. Besides, it can also demonstrate the necessity of modeling long-range dependency within the code.

4.2. Dataset

We train and evaluate our method on two datasets: BigCloneBench (Svajlenko et al., 2014) and OJClone (Mou et al., 2016). BigCloneBench is a widely used benchmark for code clone detection, which contains most of the type-3 and type-4 clone types. For OJClone, programs have the same functionality if they aim to solve the same problem. Besides, the BigCloneBench and OJClone are based on Java and C, respectively, which allows us to evaluate the generalization ability of our approach in detecting code clones.

BigCloneBench. This dataset contains over 6,000,000 positive clone pairs and 260,000 negative clone pairs. In BigCloneBench, all the code fragments are based on Java language. Due to the ambiguity between the definitions of type-3 and type-4, the code pairs of type-3 and type-4 are further partitioned by a similarity score on statement-level: strongly type-3 (ST3), moderately type-3 (MT3) and weakly type-3/type-4 (WT3/T4) are defined with similarity in [0.7, 1.0), [0.5, 0.7) and [0.0, 0.5), respectively. The majority of code clone pairs are weakly type-3/type-4.

OJClone. This dataset contains 104 programming problems together with different source codes students submit for each problem (Mou et al., 2016). In OJClone, two different source codes that solve the same programming problem are considered as a code clone pair, due to their same functionality, which belongs to type-3 or type-4.

We follow FA-AST (Wang et al., 2020a) to set up the datasets. OJClone dataset has 15 programming problems and each of it have 500 problems. It will produce over 28 million clone pairs, so 50 thousand samples are selected randomly. Similarly, 6 million positive clone pairs and 260 thousand negative clone pairs are randomly chosen form BigCloneBench dataset.

4.3. Implementation details

Our method takes the abstract syntax tree as input, the nodes of which are represented as features of 256 channels by word embedding. The graph convolutional network of the code learner module consists of two graph convolutional layers, whose channels are 256. We learn 8 code tokens for both the OJClone and BigCloneBench datasets. For the code Transformer, all the linear weights for query, key, value, *i.e.*, , $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$, $\boldsymbol{W}^V$, are used to map the input to 256 dimensional embeddings.

Our method is implemented by PyTorch (Paszke et al., 2019). During training, we loop through each code pair in the current training batch and accumulate the gradient to deal with code fragments of different lengths. We use Adam (Kingma and Ba, 2014) to optimize the model with momentum 0.99. The training procedure stops at 8 epochs with the learning rates 0.01. We set

Table 1
Evaluation of the code learner on the OJClone dataset. TL denotes token learner, while PE denotes position embedding. The MACs (Multiply–Accumulate Operations) of these methods are calculated with the same input of 500 nodes. In general, MACs is half of the FLOPs. † denotes our re-produced results.

Method	F1	Para (M)	MACs (M)
ASTNN (Zhang et al., 2019)	95.7	0.89	0.51
FA-AST + GMN (Wang et al., 2020a)	97.9 †	28.22	3.45
Transformer	97.9	**23.52**	82.09
Transformer + PE	98.5	23.59	82.09
TL (MLP) + Transformer	97.3	24.52	**1.99**
TL (MLP) + Transformer + PE	97.8	24.59	**1.99**
TL (1D Conv) + Transformer	98.0	24.53	2.00
TL (1D Conv) + Transformer + PE	98.2	24.60	2.00
TL (GCN) + Transformer	98.2	24.52	2.48
TL (GCN) + Transformer + PE	**98.8**	24.59	2.48

Table 2
Evaluation of different position embedding variants on OJClone. PE denotes position embedding. DFS and BFS denote depth first search and breadth first search.

Method	Metric		
	P	R	F1
w/o PE	98.5	97.5	98.0
Random PE (Dosovitskiy et al., 2020)	98.4	97.5	98.0
Sinusoidal PE (Vaswani et al., 2017)	98.2	97.3	97.7
DFS PE	99.1	98.1	98.6
BFS PE	99.0	97.9	98.4
DFS + BFS PE	**99.3**	**98.3**	**98.8**

batch size as 10 for OJClone and 16 for BigCloneBench, and the weight decay is chosen as 0.0005.

We evaluate our method on the evaluation set of each dataset. We follow previous methods (Zhang et al., 2019; Wang et al., 2020a) to randomly sample positive (clone) and negative (non-clone) code pairs for evaluation. For fair comparison, we directly use the partition/sampling results of FA-AST (Wang et al., 2020a). According to the sampled positive/negative pairs, we can evaluate our method by the metric of precision, which presents the fraction of the accurately detected positive/negative pairs among the whole code pairs in the evaluation set.

5. Experimental results and analyses

We present the experimental results in this section to answer the research questions. For Q1, Q2, Q3 and Q4, we conduct experiments on the OJClone dataset to evaluate the components of our method. For Q5, we compare our method with contemporary works on the OJClone and BigCloneBench datasets.

5.1. Effectiveness of the code token learner

RQ1: What is the effect of the designed code token learner? In our method, we propose a code learner to reduce the number of nodes of the abstract syntax tree. In Table 1, we evaluate the effectiveness of the proposed code learner. Note that, the code learner not only reduces the number of node but also influences the position embedding. Therefore, we do not use the position embeddings in these methods for fair comparisons.

Here, we use the multiply–accumulate operations (MACs) to measure the computational efficiency of methods. MAC is a common step that computes the product of two numbers and adds that product to an accumulator. The lower the MAC value, the faster the inference speed of the method. First, we remove the code learner (*i.e.*, w/o code learner). As we can see, when we do not use the code learner, the performances are comparable,

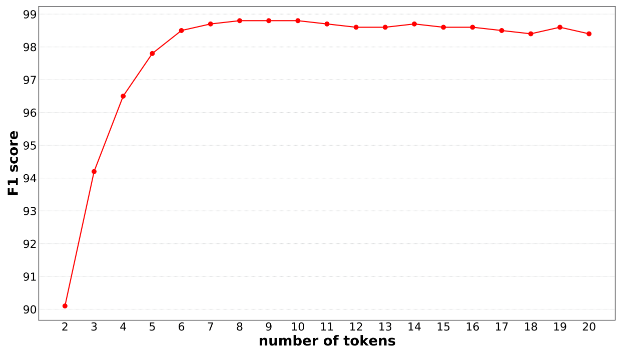


Fig. 5. Evaluation of the sensitivity on the number of code tokens.

but the MACs is exponentially larger. Moreover, we also evaluate different variants of the code learner. Specifically, we evaluate the *MLP*, which uses two 1×1 convolutional layers and the *Temporal Conv*, which uses two temporal convolutional layers, whose kernel size and dilation rate are 3 and 1, respectively. As we can see, our code learner that is based on the graph convolutional network achieves superior performance. This makes sense because our method captures the tree structure of the AST, whereas other methods do not.

Besides, since the code learner also influences the position embeddings, we further evaluate the performance of code learner with or without the position embeddings. As we can see, when we use the position embeddings, the performance of the code learner is comparable with the one without the code learner. In summary, our proposed code learner can reserve the intrinsic syntactic and structural information of the code to maintain the performance while largely reducing the computation cost.

Apart from the comparison between different variants of our method, we also presents the MACs of some previous methods. Specifically, the efficiency of our method is in between the ASTNN (Zhang et al., 2019) and FA-AST (Wang et al., 2020a). We think that the computational efficiency of our method is within an acceptable range, especially we achieve much better performance than ASTNN (Zhang et al., 2019) and better efficiency than FA-AST (Wang et al., 2020a).

Finally, we evaluate the sensitivity of our method on the number of code tokens. We conduct experiments to obtain the code clone detection results with the number of code tokens from 2 to 20. As shown in Fig. 5, our method is not sensitive to the number of code tokens ($6 \sim 16$). However, setting a small or large number of code tokens would also deteriorates the performance, because of the insufficient or redundant code tokens.

5.2. Evaluation of the position embedding

RQ2: Is the proposed tree position embedding suitable for the abstract syntax tress? Due to the position-agnostic property of Transformer architectures, position embedding is a prefer solution to introduce position information into Transformer. In Table 2, we evaluate different position embedding ways, such as sinusoidal position embedding (Vaswani et al., 2017), random position embedding (Dosovitskiy et al., 2020). As we can see, the commonly used position embeddings are unsuitable for this task, which achieves much low performance. In contrast, the proposed tree position embedding can obtain much better results than other commonly used position embedding methods and obtains the best result. This phenomenon manifests the fact that the abstract syntax tree is a specific structural data, the relative position of the nodes in the abstract syntax tree cannot be captured by regular position embeddings. Besides, the position embeddings

Table 3
Evaluation of the code Transformer on OJClone, including different methods of generating the class token, and different number of Transformer blocks.

Method		Metric		
		P	R	F1
Average pooling		99.0	98.0	97.9
Max pooling		98.7	97.6	98.1
Layer number	1	99.0	97.8	98.4
	2	99.3	98.3	98.8
	3	99.3	98.4	98.8
	4	99.2	98.1	98.6

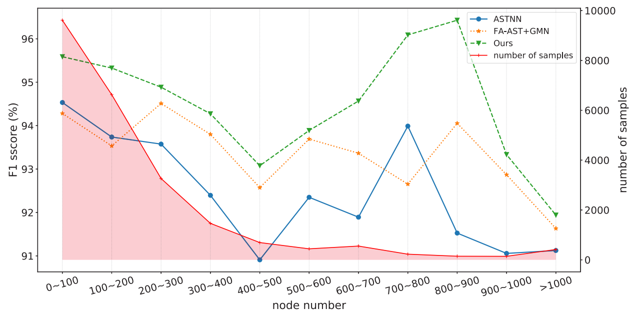


Fig. 6. Evaluation of the performance of different methods with different numbers of nodes of the abstract syntax trees on the BigCloneBench. The right *y*-axis denotes the number of samples, which corresponds to the red line. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

that are assigned with depth first search (DFS) or breadth first search (BFS) achieves comparable performance. We also evaluate the way of integrating DFS and BFS, which can better localize the position of each nodes in the tree structure. We can see that this way further improves the performance, indicating the importance of position embedding.

5.3. Evaluation of the code transformer

RQ3: What is a proper setting for a code Transformer? In our method, code Transformer plays an important role to capture long-range dependencies within the code. Here, we evaluate some designs of the code Transformer in Table 3.

Similarity token. In our method, the code token serves as a *global* token to gradually aggregates informative features within a code. Apart from the code token, we can also adopt some pooling strategies, such as average pooling or max pooling, on the learned tokens to obtain a global token. Note that, for fair comparison, we do not adopt the cross-code attention here, that is, we directly utilize the learned [SIM] token to calculate the similarity of two codes. As we can see, the [SIM] token achieves better performance.

Number of transformer layers. We also evaluate the influence of the number of Transformer layers. As shown in Table 3, our method uses two Transformer layers to reach the best performance. This phenomenon may attribute to the code learner, which largely reduces the number of node, and thus the model requires a smaller number of layers to well capture the whole information of the code.

Long code evaluation. As stated above, the code Transformer can model the long-range dependencies within a code, so as to improve the code clone detection ability for long codes. We first quantitatively evaluate our method on detecting code clones of

Table 4
F1 scores of training and testing with long code fragments on BigCloneBench.

Method	ST3	MT3	WT3/T4
ASTNN (Zhang et al., 2019)	91.9	87.1	85.3
FA-AST (Wang et al., 2020a)	97.9	96.3	93.2
Ours	99.5	98.4	93.8

Table 5
Evaluation of the cross-code attention on OJClone.

Method	Metric		
	P	R	F1
w/o cross-code attention	97.1	96.5	96.8
Vanilla attention mechanism	98.4	97.5	97.9
Cross-code attention	99.3	98.3	98.8

long codes. Specifically, in Table 4, we only utilize the code pairs, whose node number is bigger than 200, for training and testing. Since the F1 scores of these methods are close to 100% in both T1 and T2, we only report the results for ST3, MT3 and T4. As we can see, after filtering those short code fragments, our method demonstrates the evident superiority on code clone detection over two methods ASTNN (Zhang et al., 2019) and FA-AST+GMN (Wang et al., 2020a), showing the strong ability of our method on detecting code clones of long codes. In Fig. 6, we show the code clone detection results of two recent methods ASTNN (Zhang et al., 2019) and FA-AST+GMN (Wang et al., 2020a), as well as ours, under the setting of different numbers of nodes of the input abstract syntax trees. In our experiments, even if our method is based on the Transformer, which is good at modeling the long-range dependencies, our method still obtains slightly higher performance (about 1.0%) when the number of nodes is small, indicating that our method is also effective in modeling relationships between code segments. In addition, as in Fig. 6, when the number of nodes becomes large, the performance gap between our method and the two existing methods is evident. This phenomenon is evidence when the number of nodes is very large. Especially, when the number of nodes is large than 500, our method can achieve about 2.0% improvement in terms of the F1 score compared with the two existing methods. This phenomenon indicates that our method ca improve the ability of detecting long codes.

5.4. Evaluation of the cross-code attention

RQ4: Is the cross-attention necessary? Several comparative experiments are conducted to evaluate the module of cross-code attention. As we can see in Table 5, removing the cross-code attention results in inferior performance. Moreover, we also utilize a vanilla attention mechanism, which directly calculates the cosine similarity between the [SIM] token and code tokens and then aggregates features in accordance with the affinities with another code. We can see that this way achieves slightly lower performance than ours, owing to the flexible ability of our cross-code attention benefited from the learnable parameters.

5.5. Comparison with SOTA methods

RQ5: How does our approach perform in code clone detection compared with state-of-the-art methods? In this question, we would like to find out whether our method is effective on code clone detection. First, we briefly introduce several code clone approaches as follows: (1) Deckard (Jiang et al., 2007): a code clone approach based on syntax tree, which adopts Euclidean distance to calculate the similarity between code fragments. (2) DLC (White et al., 2016): a deep-learning-based method that uses

Table 6
Code clone detection results on BigCloneBench. P, R, F1 denote precision, recall and F1 score, respectively. Note that, the column of *ALL* is a weighted sum result according to the percentage of various clone types (Wei and Li, 2017). The results of Deckard (Jiang et al., 2007) and SourcererCC (Sajnani and Saini, 2016) are copied from Wei and Li (2017) and Sajnani and Saini (2016). † means that the methods are pre-trained on large datasets.

Method	T1			T2			ST3			MT3			WT3/T4			ALL		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Deckard (Jiang et al., 2007)	93.0	60.0	73.0	92.0	58.0	71.0	76.0	42.0	54.0	84.0	12.0	21.0	93.0	1.0	2.0	93.0	2.0	3.0
SourcererCC (Sajnani and Saini, 2016)	90.0	100.0	94.0	90.0	98.0	93.0	98.0	61.0	77.0	90.0	5.0	10.0	88.0	0.0	0.0	88.0	1.0	1.0
RtvNN (White et al., 2016)	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	95.0	1.0	1.0
CDLH (Wei and Li, 2017)	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	92.0	74.0	82.0
ASTNN (Zhang et al., 2019)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.6	99.8	100.0	97.9	98.9	93.3	92.2	92.8	93.4	92.3	92.9
SSFL (Fang et al., 2020)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.5	99.7	100.0	98.0	99.0	93.5	92.7	93.1	93.5	93.0	93.2
FA-AST+GMN (Wang et al., 2020a)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.6	99.8	100.0	96.5	98.2	95.7	93.5	94.6	95.8	93.6	94.7
CodeBERT (Feng et al., 2020) †	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.8	99.9	100.0	97.0	98.4	94.6	93.2	93.9	94.7	93.4	94.1
GraphCodeBERT (Guo et al., 2020) †	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.8	99.9	100.0	98.2	99.1	94.7	**95.1**	94.9	94.8	**95.2**	95.0
Ours	100.0	100.0	100.0	100.0	100.0	100.0	99.8	**100.0**	**99.9**	100.0	**98.5**	**99.2**	**95.8**	94.8	**95.3**	**96.1**	**95.2**	**95.6**

a recursive auto-encoder to extract deep features. (3) CDLH (Wei and Li, 2017): a deep learning approach to learn syntactic features for code clone detection. (4) Deepsim (Zhao and Huang, 2018): a semantic-based approach that utilizes supervised deep learning to measure functional code similarity. (5) ASTNN (Zhang et al., 2019): a state-of-the-art model, which splits a large AST into a sequence of small statement trees and encodes the statement trees into lexical and syntactical vectors. (6) FA-AST+GMN (Wang et al., 2020a): a state-of-the-art model, which augments original ASTs with explicit control and data flow edges, and then Graph Matching Networks (GMN) is utilized to measure the similarity of code pairs. We can see that, this method also strives to capture long-range dependency by graph operations. (7) SSFL (Fang et al., 2020): a state-of-the-art model, which learns a novel joint code representation to learn hidden syntactic and semantic features of source codes.

As shown in Table 6, on BCB dataset, all the methods obtain promising performance in detecting similar code fragments in type-1 and type-2, since both code fragments are almost the same except some different function and variable names. While for other types, the effectiveness of our method can be represented. Since the Transformer not only captures the long-range dependencies within the code but also learn the relations between the statements of the code, which can be effective to detect those functionally similar codes whose code fragments are highly coupled to achieve a function. As we can see, Deckard only utilizes hand-crafted features as code representations and calculated similarity by using euclidean distance. Therefore, it considers each dimension the same for calculating similarity, which is not enough and thus obtains a low recall value. Moreover, even though the DLC and CDLH adopt an RNN language model to learn code embeddings, they treat the AST as a binary tree, which destroy the original structure of the code, leading to inferior performance. Even if the ASTNN (Zhang et al., 2019) explicitly models sequential naturalness of statements in code fragments, they use the bi-directional RNN to capture the dependencies within the code, which is insufficient to capture long-range dependencies. Therefore, it is unsurprising that our method outperforms ASTNN in the clone types of MT3 and T4, which requires the ability of capturing relations between code fragments. Moreover, compared with the state-of-the-art method FA-AST+GMN (Wang et al., 2020a), our methods still achieves better performance. Even if FA-AST+GMN utilizes the graph neural network, which can also capture the dependencies within the code, it still cannot capture the long-range dependencies due to the sparse connections between different nodes. Besides, due to the smoothing nature of the GNN, they cannot stack too many layers, limiting its modeling capacity.

Besides, we also compare our method with two pre-trained model, CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020), which are multi-programming-lingual models pre-trained on NL–PL pairs in 6 programming languages (Python, Java, PHP, et al.). Considering the two models are not pre-trained on the programming language C, we only compare with them on the BigCloneBench dataset. We can see that our method still outperforms the two models even they are pre-trained on large-scale datasets, indicating our method can indeed capture some aspects ignored by these models, e.g., long code fragments.

Table 7
Code clone detection results on OJClone. P, R, F1 denote precision, recall and F1 score, respectively.

Method	Metric		
	P	R	F1
Deckard (Jiang et al., 2007)	99.0	5.0	9.5
SourcererCC (Sajnani and Saini, 2016)	7.0	74.0	12.8
DLC (White et al., 2016)	52.5	68.3	59.4
CDLH (Wei and Li, 2017)	47.0	73.0	57.2
PDG+HOPE (Tufano et al., 2018)	76.2	7.0	12.8
PDG+GGNN (Allamanis et al., 2017)	77.3	43.6	55.8
Deepsim (Zhao and Huang, 2018)	70.0	83.0	75.9
ASTNN (Zhang et al., 2019)	98.9	92.7	95.7
SSFL (Fang et al., 2020)	97.0	95.0	96.0
Ours	**99.3**	**98.3**	**98.8**

We can see the similar results on the OJClone dataset in Table 7. It can be observed that CDLH, DeepSim, ASTNN, SSFL and our approach obtain better results than those unsupervised methods, such as Deckard, DLC, and PDG. Since the code types in OJClone are almost functional clones, this phenomenon demonstrates that unsupervised methods cannot capture the similarity of functional clones. Compared with other supervised method: CDLH, Deepsim and SSFL, our method still obtains better performance. Besides, compared with the FA-AST+GMN (Wang et al., 2020a), our method significantly surpasses it by **0.3%**, **1.6%** and **0.9%** on the metrics of precision, recall and F1 score, respectively.

Traditionally, code clone can be also grouped into intra-project and inter-project clones, which are located in the same/different software system and show different properties. To show the powerful ability of our method in both intra-project and inter-project code clone detection, in Table 8, we compare our method with some previous methods on BigCloneBench. Note that, we follow (Svajlenko and Roy, 2015) to report recall results. As we can see, our method can achieve better performance, demonstrating the ability of our method in detecting different types of clones. Moreover, compared with these methods that exhibit significant differences between their intra-project and inter-project performance, our method can obtain consistent performance for both intra-project and inter-project clones, indicating the generalization ability of our method.

6. Qualitative results

To attain further insights in our method, in this section, we present some qualitative results, including the intermediate results of the code token learner and feature visualization.

Table 8
Intra-project and inter-project recall results on BigCloneBench.

Method	Intra-project			Inter-project		
	ST3	MT3	WT3/T4	ST3	MT3	WT3/T4
CCFinderX (Kamiya et al., 2002)	10.0	1.0	0.2	17.0	1.0	0.2
Deckard (Jiang et al., 2007)	31.0	10.5	1.8	30.0	9.8	1.9
CDLH (Wei and Li, 2017)	93.0	90.2	73.8	93.0	90.0	73.2
Ours	100.0	98.7	95.0	100.0	98.1	94.4

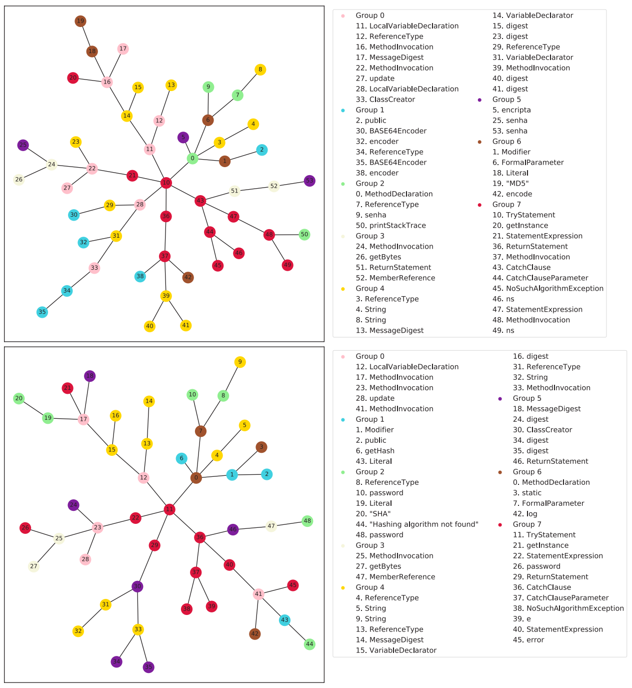


Fig. 7. We visualize the graph representation of the ASTs of two similar codes. The nodes with the same color mean they are grouped into the same super-node. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

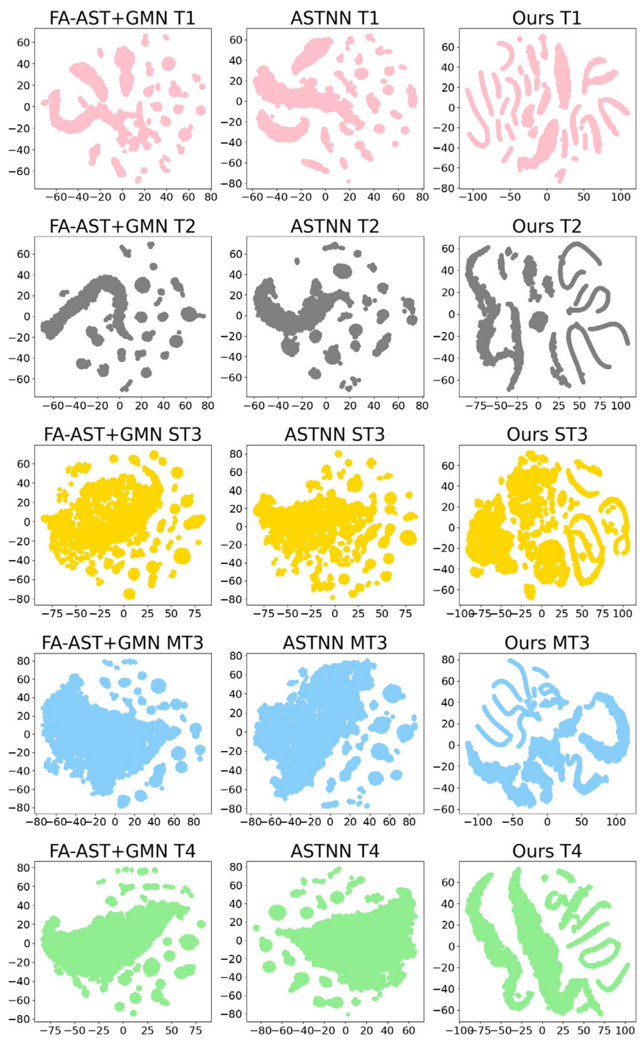


Fig. 8. Feature visualization of different clone types on the BigCloneBench dataset. T1, T2, ST3, MT3, T4 represent type-1, type-2, strong type-3, moderate type-3 and type-4, respectively.

6.1. What learned by code token learner

In Fig. 7, we show the intermediate results of the code token learner. Specifically, we visualize the graph representation of the input AST with its node types. The nodes with the same color mean they are grouped into the same super-node. Specifically, We assign each node to its group according to the assignment matrix **S**. For example, for the ith node, we collect the ith row $\mathbf{S}[i, :]$ of the assignment matrix **S**, and find the index $j \in [1, N_t]$ of the maximum value in $\mathbf{S}[i, :]$. Therefore, the ith node belongs to the jth group. As we can see, some code tokens focus on the leaf nodes, while some code tokens focus on the nodes around the root node. Moreover, neighbor nodes, which follows the sequential execution, are usually grouped into the same group, which is reasonable and interpretable.

6.2. Feature visualization

We visualize the learned features that are before calculating similarity by using t-SNE (Van der Maaten and Hinton, 2008) in Fig. 8. We show the feature visualization of different clone types on the BigCloneBench dataset and compare our method with two recent methods, ASTNN (Zhang et al., 2019) and FA-AST+GMN (Wang et al., 2020a). We can see that, our method obtains more separated feature representations compared with the other two methods. Besides, may be due to the different backbone architecture of the Transformer, our learned features of similar codes lie in a mainfold, while the other two methods are prone to learn clustered features. This phenomenon may manifest that our Transformer can capture the structural information of codes to achieve better code clone detection performance.

7. Threats to validity

We demonstrate the internal and external validity of our method as follow.

Internal validity: We only consider the granularity of method or function and overlook code clone detection at other code levels, so we may miss the code snippets with function calls. Moreover, the effectiveness of our approach is limited by the quality of current benchmarks. With the introduction of deep learning and the addition of various strategies, such as code alignment, the existing datasets have entered performance saturation and cannot well reflect the effectiveness of the method.

External validity: Our method are evaluated on only two programming languages, *i.e.*, Java and C. Actually, our method can be potentially used to detect code clones for other programming languages, whose ASTs can be also extracted. However, since we have not evaluated this, we cannot claim the effectiveness on these programming languages. In addition, even if we propose a code learner to largely reduce the computational cost under the Transformer-based paradigm, the required computational cost and memory is still higher than those CNN-based methods and RNN-based methods, which may limit the application scenarios of this method.

8. Related work

In this section, we introduce the prior studies most related to code clone detection and our proposed algorithms, which can be classified into two categories, namely application of deep learning in code clone detection and the Transformer algorithms done on code analysis domain.

8.1. Deep learning in code clone detection

Traditional code clone detection techniques meanly use representations such as text (Baker, 1993, 1996), tokens (Sajnani and Saini, 2016), syntax indicators or abstract syntax trees (Jiang et al., 2007) for measuring the similarity between code snippets. With the breakthrough of deep learning in Natural Language Processing (NLP), deep learning has been applied in the field of software engineering due to the similarities between programs and natural languages. DLC (White et al., 2016) introduces a language model to detect clones, where a recursive learning process are proposed to learn fragment representations. CDLH (Wei and Li, 2017) propose to transform ASTs into binary hash codes by leaning hash functions, and then utilizes an AST-based LSTM to capture the lexical and syntactical information of source codes. Deepsim (Zhao and Huang, 2018) attempt to encode the control flow and data flow into a semantic matrix, which are used to detect functional clones with deep neural networks. ASTNN (Zhang et al., 2019) splits a large AST into a sequence of small statement trees, and encodes the statement trees to vector representations, which are further forwarded into a bi-directional recurrent neural network and finally produce the vector representation of a code fragment. The most related work is FA-AST (Wang et al., 2020a), It presents a siamese network based on the graph neural network. Theoretically, it can capture more broader information compared with those CNN-based methods and RNN-based methods with a densely connected graph. However, because of the graph is generated from the abstract syntax tree, the receptive field of FA-AST is also limited.

8.2. Code analysis with transformer

Transformer has achieved astonishing success in many research fields, such as neural machine translation (Vaswani et al., 2017), image classification (Dosovitskiy et al., 2020; Liu et al., 2021), object detection (Carion et al., 2020; Chen et al., 2021), semantic segmentation (Wang et al., 2021), *etc*. With the success of Transformer in these fields, there are more and more researchers apply it to code analysis. Chirkova and Troshin (2021) presented an empirical study to validate the Transformer on different software engineering tasks, such as code completion, bug fixing. They argued that Transformer can consistently improve the performance via fully modeling the input codes, especially the ASTs. Svyatkovskiy et al. (2020) deployed a novel end-to-end code sequence completion system based on a pretrained multi-layer generative Transformer model, which is a variant of the GPT-2 (Radford et al., 2019). Sun et al. (2020) employed the attention mechanism of Transformers to enable long-dependency for code generation, which introduces a novel AST encoder to incorporate grammar rules and AST structures into the network. Ahmad et al. (2020) mainly leverage the ability of capturing long-range dependencies of the Transformer for code summarization. Wang et al. (2020b) incorporated the Transformer with reinforcement learning to unify code summarization and code search. Ahmad et al. (2020) also proposed a Transformer-based method for code summarization. They used a novel module, namely copying attention, to leverage the Transformer on code summarization, which generates summarized texts by using the words and tokens simultaneously. To this end, they can generate some readable code in summarized texts. According to the performance of Transformer-based methods in multiple tasks of code analysis, we have the reason to believe that it can also show excellent performance in code clone detection.

9. Conclusion and future work

In this paper, we proposed an efficient siamese Transformer network for code clone detection. To make it possible to apply Transformer to long codes, we propose a code learner, which leverages the graph convolutional network to capture structure of the code, to largely reduce the number of tokens for each code fragment. Besides, we also proposed a position embedding strategy, which can represent the position of a node in the tree structure of the abstract syntax tree. The learned code tokens are combined with the position embedding, which are forwarded into the Transformer with an additional similarity token. The output token corresponding to the similarity token captures global information of the entire code, which is further used to capture cross-code similarities by a cross-code attention module. We conduct experiments on two benchmark datasets, and experimental results indicate that the proposed method can obtain the state-of-the-art performance without requiring much computation costs.

Currently, our proposed methods only takes the abstract syntax tree as input. However, there are some information that cannot be represented in the abstract syntax tree, such as control flow, *etc*. In the future, we aim to use more inputs to enable the our method to capture different views of code fragments.

CRediT authorship contribution statement

Aiping Zhang: Conceptualization, Methodology, Software, Writing. **Liming Fang:** Writing – review, Supervision. **Chunpeng Ge:** Investigation, Editing. **Piji Li:** Review, Validation. **Zhe Liu:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported by the National Key R&D Program of China (Grant No. 2021YFB3100700), the National Natural Science Foundation of China (Grant No. 62272228) and Shenzhen Science and Technology Program (Grant No. JCYJ20210324134408023).

References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W., 2020. A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 4998–5007.
Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.
Baevski, A., Auli, M., 2018. Adaptive input representations for neural language modeling. arXiv preprint arXiv:1809.10853.

Baker, B.S., 1993. A program for identifying duplicated code. Comput. Sci. Stat. 49.

Baker, B.S., 1996. Parameterized pattern matching: Algorithms and applications. J. Comput. System Sci. 52 (1), 28–42.

Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, pp. 368–377.

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. arXiv preprint arXiv:2005.14165.

Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., Zagoruyko, S., 2020. End-to-end object detection with transformers. In: European Conference on Computer Vision. Springer, pp. 213–229.

Chen, T., Saxena, S., Li, L., Fleet, D.J., Hinton, G., 2021. Pix2seq: A language modeling framework for object detection. arXiv preprint arXiv:2109.10852.

Chirkova, N., Troshin, S., 2021. Empirical study of transformers for source code. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 703–715.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al., 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

Fang, C., Liu, Z., Shi, Y., Huang, J., Shi, Q., 2020. Functional code clone detection with syntax and semantics fusion learning. In: ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 516–527.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. GraphCodeBERT: Pre-training code representations with data flow. In: International Conference on Learning Representations.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Hua, W., Liu, G., 2021. Transformer-based networks over tree structures for code classification. Appl. Intell. 1–15.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: International Conference on Software Engineering. IEEE, pp. 96–105.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Li, M.Q., Fung, B., Charland, P., Ding, S.H., 2019. I-MAD: a novel interpretable malware detector using hierarchical transformer. arXiv preprint arXiv:1909.06865.

Li, Q., Han, Z., Wu, X.-M., 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In: Thirty-Second AAAI Conference on Artificial Intelligence.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., Dollár, P., 2017. Focal loss for dense object detection. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 2980–2988.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B., 2021. Swin transformer: Hierarchical vision transformer using shifted windows. arXiv preprint arXiv:2103.14030.

Van der Maaten, L., Hinton, G., 2008. Visualizing data using t-sne. J. Mach. Learn. Res. 9 (11).

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: AAAI Conference on Artificial Intelligence. 30, (1).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al., 2019. Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems. 32, pp. 8026–8037.

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

Sajnani, H., Saini, V., 2016. Sourcerercc: Scaling code clone detection to big-code. In: International Conference on Software Engineering. IEEE, pp. 1157–1168.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L., 2020. Treegen: A tree-based transformer architecture for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence. 34, (05), pp. 8984–8991.

Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M., 2014. Towards a big data curated benchmark of inter-project code clones. In: International Conference on Software Maintenance and Evolution. IEEE, pp. 476–480.

Svajlenko, J., Roy, C.K., 2015. Evaluating clone detection tools with bigclonebench. In: 2015 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 131–140.

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. Intellicode compose: Code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1433–1443.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D., 2018. Deep learning similarities from different representations of source code. In: IEEE/ACM 15th International Conference on Mining Software Repositories. IEEE, pp. 542–553.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems. pp. 5998–6008.

Wang, W., Li, G., Ma, B., Xia, X., Jin, Z., 2020a. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: International Conference on Software Analysis, Evolution and Reengineering. IEEE, pp. 261–271.

Wang, Q., Li, B., Xiao, T., Zhu, J., Li, C., Wong, D.F., Chao, L.S., 2019. Learning deep transformer models for machine translation. arXiv preprint arXiv:1906.01787.

Wang, W., Xie, E., Li, X., Fan, D.-P., Song, K., Liang, D., Lu, T., Luo, P., Shao, L., 2021. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. arXiv preprint arXiv:2102.12122.

Wang, W., Zhang, Y., Zeng, Z., Xu, G., 2020b. Trans^3: A transformer-based framework for unifying code summarization and code search. arXiv preprint arXiv:2003.03238.

Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: International Joint Conference on Artificial Intelligence. pp. 3034–3040.

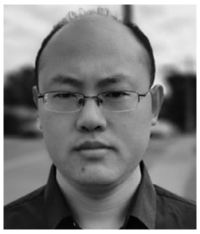
White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 87–98.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: International Conference on Software Engineering. IEEE, pp. 783–794.

Zhao, G., Huang, J., 2018. Deepsim: deep learning code functional similarity. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 141–151.

Aiping Zhang received the bachelor's degree in electronic information engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016, and the master's degree at the Nanjing University of Aeronautics and Astronautics, in 2021. Her current research interests include AI security, cyberspace security, code clone detection, and machine learning.

Liming Fang (Member, IEEE) received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics, in 2012, and has been a postdoctor with the information security from the City University of Hong Kong. He is the associate professor with the School of Computer Science, Nanjing University of Aeronautics and Astronautics. Currently, he is a visiting scholar with the Department of Electrical and Computer Engineering New Jersey Institute of Technology. His current research interests include cryptography and information security. His recent work has focused on the topics of public key encryption with keyword search, proxy re-encryption, identity-based encryption, and techniques for resistance to CCA attacks.



Chunpeng Ge (Member, IEEE) received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics, in 2016. He was a research fellow with the Singapore University of Technology and Design in 2018. Currently, he is an associate professor with the Nanjing University of Aeronautics and Astronautics and a research fellow with the University of Wollongong. His current research interests include cryptography, information security, and privacy preserving for blockchain. His recent work has focused on the topics of public key encryption with keyword search, proxy reencryption, identity-based encryption, and privacy preserve for blockchain systems.

Piji Li received the master's and bachelor's degree from School of Computer Science and Technology in Shandong University from 2005 to 2012, under the supervision of Prof. Jun Ma. He received the Ph.D. degree from the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong in 2018, under the supervision of Prof. Wai Lam. He is now a professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. Previously, he was a senior researcher at Tencent AI Lab.

Zhe Liu (Senior Member, IEEE) received the B.S. and M.S. degrees from Shandong University, in 2008 and 2011, respectively, and the PhD degree from the Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg, Luxembourg, in 2015. He is a professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA), China. Before joining NUAA, he was a researcher with the SnT, University of Luxembourg, Luxembourg. His Ph.D. thesis has received the prestigious FNR Awards 2016 — Outstanding Ph.D. Thesis Award for his contributions in cryptographic engineering on IoT devices. His research interests include computer arithmetic and information security. He has co-authored more than 70 research peer-reviewed journal and conference papers.