Improving Code Representation Learning via Multi-view Contrastive Graph Pooling for Abstract Syntax Tree

Ruoting Wu, Yuxin Zhang, and Liang Chen(✉)

School of Computer Science, Sun Yat-sen University, Guangzhou, China
{wurt8,zhangyx355}@mail2.sysu.edu.cn, chenliang6@mail.sysu.edu.cn

Abstract. As the field of code intelligence continues to grow, Code representation learning has emerged as a research hot spot. Given that code structure can be naturally represented as graphs, Graph Neural Networks (GNNs) have proven highly effective for learning graph representations of source code. Pooling, as an essential operation for GNN-based models, is limited in its ability to leverage the rich hierarchical information presented in tree-like graph, especially Abstract Syntax Trees. In order to learn the graph representation of code more effectively, we propose a novel pooling method called TreePool. TreePool directly splits tree-like graphs using depth filtering based on the tree structure to form a sequence of pooled graphs sorted by descending size of subgraphs. Then local-local contrastive learning between these neighboring subgraphs is conducted to preserve the information of the graph before pooling. Through TreePool, multiple views of representation are learned and fused to obtain the final code graph representation. We conduct TreePool on a supervised framework and experimental results demonstrate that the average improvements on two real-world datasets in terms of accuracy are 1.1% and 3.3%. It also exhibits excellent performance in an unsupervised framework. Our results show that TreePool can effectively learn meaningful Abstract Syntax Tree representation of code and exhibit good performance in code classification tasks.

Keywords: Code Representation Learning · Graph Pooling · Graph Neural Networks

1 Introduction

As an essential application in the context of collaborative computing, intelligent software engineering provides intelligent solutions to the software development domain. With the successful application of deep learning, there is a growing trend in intelligent code-related tasks which will further enable multiple collaborative computing scenarios, e.g., collaborative code development, automated code analysis, or code project management. The performance of code-related tasks, method name prediction [2], code completion [5], or code classification [36]

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 562, pp. 242–261, 2024.
https://doi.org/10.1007/978-3-031-54528-3_14

etc., are relied on the precisely generation of the representation of code snippets. To further improve the code representation learning in these tasks, researchers exploit deep learning methods to extract deep features not only from the original sequence of code but also the structures generated from the compiler, such as Abstract Syntax Tree (AST), Control Flow Graph, etc. While sequential models [7,13] are generally used due to the similarity between code sequence and natural language, the study treating code as graph is still in its infancy. There is a challenge in developing graph-based code representation learning techniques.

Illustrated in Fig. 1, as Graph Neural Networks (GNNs) have widely applied to the non-Euclidean structure (e.g., graph) in multiple fields [18–20,33], methods on code representation learning exploit GNNs to [5,16] encode nodes in an AST or an enhanced code graph with the return of a final representation of code for downstream tasks. GNNs, which apply a message-passing mechanism to aggregate the neighborhood information, show powerful expressiveness on encode node and graph representation. However, existing GNN-based methods have some flaws: (1) Recent methods [5,32] designed for code datasets consider enhancing the AST by adding different types of edges but ignore the original tree structure of AST. (2) Typically graph learning methods with GNNs, from simple readout functions [28] to complex pooling methods [9,17,31] are mostly designed for traditional graphs and cannot easily be transferred to code area.

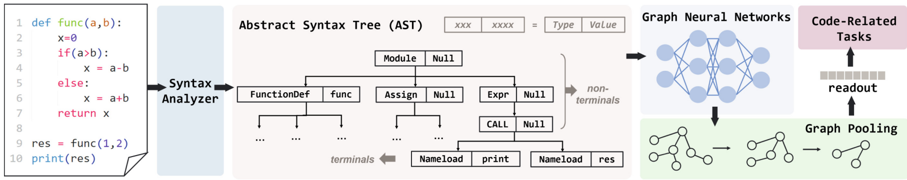

Fig. 1. The utilization of Abstract Syntax Tree for code representation learning

To obtain reasonable and effective graph-level representation, graph pooling methods are essential in GNN-based models for code representation learning. Trees have sequential layers, which indicates that the spatial information between nodes can be considered in pooling methods like the field of image, in which models such as Convolutional Neural Networks can easily select a small part based on the spatial proximity to generate max-pooling or mean-pooling [15]. Meanwhile, most of the information is in the leaf nodes of AST, while nodes in the upper layers contain little information in semantics. Simply adding more views may introduce more noise into the final graph representation.

In this work, we propose TreePool, a multi-view contrastive graph pooling method for Abstract Syntax Tree. The contributions of this work are:

- We propose a simple node selection strategy for pooling the tree-like graph, which creates views based on the spatial proximity of the tree structure. It reduces the calculation of the assignment matrix.

- We conduct a simplified local-local contrastive learning between graphs before and after pooling to preserve information.
- TreePool works on both supervised and unsupervised architecture. The results demonstrate the effectiveness of the code classification task.

The structure of the remaining sections in this paper is outlined as follows. Section 2 discusses the preliminaries and presents the problem definition. Section 3 introduces the detailed design of TreePool. The experimental setting and results are reported in Sects. 4 and 5, respectively. We provide a comprehensive discussion of the model in Sect. 6. The related works are presented in Sect. 7. Lastly, Sect. 8 concludes the paper.

2 Preliminaries

2.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) of code is a tree structure generated by Syntax Analyzer with a typical unambiguous context-free language grammar [1]. AST reflects the structural and syntax information of code. In a syntax tree, the top node represents the beginning symbol, the middle nodes denote the non-terminals, and the bottom nodes indicate the terminals, usually consisting of the variables and identifiers from the original code sequence.

AST can be seen as a tree-like graph denoted as $G = (V, E)$, where V and E are the sets of nodes and edges. The directed edges in E represent the parent-child node relationships, with each edge $e = (u, v) \in E$ indicating that node u is the parent of node v. To represent the tree-like graphs in deep learning models, we use the adjacency matrix $A \in \mathbb{R}^{n\times n}$ and the feature matrix $X \in \mathbb{R}^{n\times d}$, where n is the number of nodes and d is the number of features per node.

2.2 Code Representation Learning

The goal of code representation learning is to generate a useful representation of the code snippet (i.e., code files, functions) which captures the semantic and syntactic information of code and further improve various downstream tasks. In the context of graph representation learning, the goal is to learn a compact and low-dimensional global graph-level embedding $Z_g \in \mathbb{R}^{1\times k}$, where $k \ll d$, using $A \in \mathbb{R}^{n\times n}$ and $X \in \mathbb{R}^{n\times d}$ to represent graph structure such as AST.

2.3 Graph Pooling for Graph Learning

Graph pooling is a critical component in graph representation learning. As a down-sample operation, graph pooling refers to the process of aggregating information from a graph and creating a new coarsened graph with a smaller size or fewer nodes, while preserving the most important structural information.

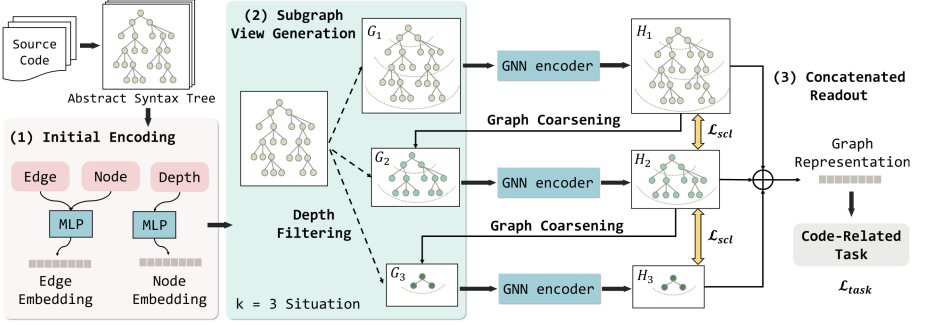

Fig. 2. The overview of TreePool in code representation learning architecture.

Formally, graph pooling process is shown as Eq. (1), which coarsen the i-th view G_i into a smaller subgraph view G_{i+1}, where X_i and A_i denote the node features and adjacency matrix of the original view respectively. G_i is first encoded by GNN layer, followed by the coarsening operation to obtain new adjacency matrix A_{i+1} and feature matrix X_{i+1} with new node indexes idx.

$$Z_i = \sigma(GNN(X_i, A_i));\; X_{i+1} = Z_i(idx, :);\; A_{i+1} = A_i(idx, idx) \tag{1}$$

Typically, some standard readout functions [28], such as mean, max, or sum, take the updated node representations and aggregate them to produce the graph-level representation.

3 Present Work: TreePool

Illustrated in Fig. 2, TreePool can incorporate Graph Neural Networks for graph learning. (1) The initial Encoding module first models the information from AST, then (2) Subgraph View Generation module utilizes depth filtering to determine the subgraphs that will be coarsened. In pooling process, the GNN encoder and the coarsening operation based on the views previously decided upon alternates. Finally, all subgraph representations used (3) Concatenated Readout to combine as a graph-level representation for downstream tasks. In the following section, we will describe each part separately.

3.1 Subgraph View Generation

The key step in graph pooling is simplifying graphs to new small graph views by reducing the number of nodes and edges, i.e., the assignment from the original view to the new view. In TreePool, the procedure is succinct which requires no further calculation for the assignment matrix compared to typical graph pooling methods. The nodes and edges are decided only by the depth of the nodes through the subgraph view generation module.

TreePool emphasizes the significance of the natural spatial neighbors in tree-like graphs. Throughout each pooling iteration, the generation of smaller subgraph views is the opposite of the direction of node derivation in an Abstract Syntax Tree using semantic rules. Each subgraph view can be regarded as a subtree encompassing the root node, with varying sizes due to the depth of information. Since each subgraph view can be perceived as an incomplete derived Abstract Syntax Tree, all views share a semantic similarity.

The depth of the Abstract Syntax Tree is decided by the node which is the deepest in the graph. For each graph G, the depth is denoted as d_g. The tree depth is divided average with the fixed view number k. The i-th view contains the nodes with depth $d_v \in [0, (k-i)\frac{d_g}{k})$ and the edges between these nodes.

The number of views k, which can be customized, defines the time that coarsening procedure is utilized. As the module generates a sequence of subgraph views length of k, the graph coarsening procedure is conducted $k-1$ times.

3.2 Mutual Information Maximization

TreePool introduces contrastive learning to maximize the mutual information between the subgraph views to let the coarsened graph reflects the previous graph maximally. It promotes alignment and semantic consistency between different views while avoiding the problem of information redundancy that may arise from simply combining multiple views. Therefore, the node features in the coarsened graph preserve the information of the original graph before pooling.

Maximizing the mutual information between each of the graph pairs is a local-local mutual contrastive learning problem. Mathematically, the mutual information between the graph before and after pooling (i.e., graph G_i and G_{i+1}) can be defined as $I(G_i, G_{i+1})$.

However, the input graphs have different scales, which are unable to use the common paradigm of local-local contrastive learning, such as [37]. To address this issue, we first simplify it into a local-global contrastive learning problem. Inspired by previous work on contrastive learning [27,35], we later maximize $I(G_i, G_{i+1})$ by using a binary cross entropy (BCE) loss between the node and graph representations of these views, as shown in Fig. 3.

Neural Estimation. The mutual information $I(\cdot,\cdot)$ between the views before and after pooling is defined as the KL-divergence between the joint distribution $P(G_i, G_{i+1})$ and the product of their marginal distributions $P(G_i) \otimes P(G_{i+1})$:

$$I(G_i, G_{i+1}) = D_{KL}(P(G_i, G_{i+1}) || P(G_i \otimes G_{i+1})). \tag{2}$$

The KL-divergence can be effectively estimated by the f-divergence representation [3]. The lower bound $I_\theta(G_i, G_{i+1})$ defines as Eq. (3), where σ represents the activation function, $L(\cdot,\cdot)$ represents the projection function from node space to the real domain. In practice, we can maximize the mutual information by optimizing the lower bound. We use neural network L_θ to replace the $L(\cdot,\cdot)$ to estimate the mutual information of the inputs G_i and G_{i+1}.

$$I_\theta(G_i, G_{i+1}) \geq \mathbb{E}_{G_i,Gi+1} log(\sigma(L(G_i, G_{i+1})))+$$
$$\mathbb{E}_{G_i,Gi+1} log(1 - \sigma(L(G_i, G_{i+1}))). \quad (3)$$

Simplified Contrastive Learning. To transform the local-local contrastive learning into a simpler local-global problem, TreePool encode the graph before pooling as a graph-level representation. We use a standard readout operation (max, mean or sum) and a single-layer MLP to encode and project the graph representation. As Eq. (4), the output is a summary vector s_i.

$$s_i = W_s \boldsymbol{readout}(X_i) + b_s, \quad (4)$$

where X_i represents the total node embedding in graph G_i, W_s and b_s represent the learnable weight and bias from the projection layer for G_i.

For the local representation of the new graph G_{i+1}, we followed the previous design for G_i, with a single-layer MLP for graph after pooling to make sure that hidden dimensions are the same as the summary vector s_i.

We first employ negative sampling to create a corrupted graph $\tilde{G}_{i+1}$ after pooling the new graph G_{i+1}. It is commonly used in creating negative sampling to randomly permute the embedding of nodes within a view [27]. However, it not only fails to truly separate the positive and negative local nodes from one another when projected into a new dimension, but also does not capture the information between different features within each node.

Therefore, TreePool generates negative samples by permuting the feature dimensions of each node. Specifically, we randomly select a feature index and permute the feature values of all nodes up to that index, resulting in a randomized feature matrix $X_{i+1}^{neg} = X_{i+1}[: random(feature)]$ for $\tilde{G}_{i+1}$.

The projection layer takes both the original and corrupted nodes as input. Equation (5) shows the result for the original graph as input, while the corrupted graph receives $\tilde{l}_{i+1}$ as the output, where W_l and b_l represent the learnable weight and bias from the projection layer for both G_i and $\tilde{G}_{i+1}$.

$$l_{i+1} = W_l X_{i+1} + b_l. \quad (5)$$

In a typical graph contrastive learning framework, the discriminator scores the local-global pairs using a bilinear scoring function, which indicates the function L_θ. It can be formulated as follows:

$$D(l_{i+1}, s_i) = \sigma(l_{i+1}^T W s_i), \quad (6)$$

where W is a learnable scoring matrix, σ is the logistic Sigmoid nonlineaity.

To reduce the calculation of the bilinear layer, we follow the simplification of Group Discrimination [35], combine the projected positive and negative node representation (after pooling)with the projected graph embedding (before pooling). After adding s_i and $l_{i+1}(\tilde{l}_{i+1})$, we use sum operation feature-wise to get the final score as the input of the discriminator, to further predict the result of

positive and negative samples, as shown in Eq. (7), where $\boldsymbol{combine}(\cdot,\cdot)$ contains the add and sum operation conducted on the s_i and l_{i+1} ($\tilde{l}_{i+1}$).

$$D(l_{i+1}, s_i) = \boldsymbol{combine}(l_{i+1}, s_i), \tag{7}$$

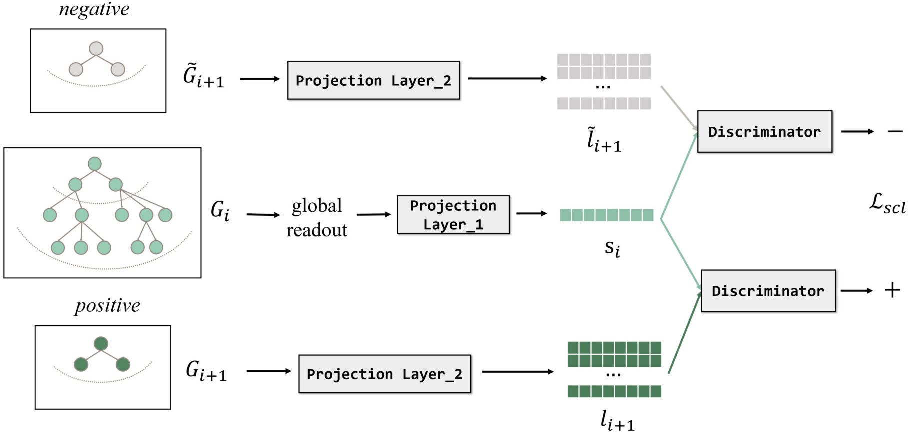

Fig. 3. The computation of the simplified contrastive loss.

Objective Function. The objective function of the simplified contrastive learning on the adjacent subgraph views can be formulated as Eq. (8).

$$L_{scl} = \frac{1}{2N}(\Sigma_{i=1}^{N} logD(l_{i+1}, s_i) + log(1 - D(\tilde{l}_{i+1}, s_i))), \tag{8}$$

where N represents the node num after pooling. Following Group Discrimination, we can rewrite the loss as a simple BCE loss, where $y_i' = combine(l_{i+1}, s_i)$.

$$L_{scl} = -\frac{1}{2N}(\Sigma_{i=1}^{N} y_i(log(y_i')) + (1 - y_i)(log(1 - y_i'))). \tag{9}$$

As shown in Eq. (9), we give a simplified contrastive loss L_{scl} for the adjacent views during graph coarsening.

3.3 Graph Encoder

For the node encoder, we choose Graph Convolutional Network (GCN) [14]. The type of GNN encoder is easily replaced by other types, and we provide a detailed discussion on the GNN encoder type in Sect. 5.5. The GCN encoder processes the nodes in the view to generate node embeddings, propagating information via the input graph's structure using the propagation rule:

$$X^{(m+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} A \tilde{D}^{-\frac{1}{2}} X^{(m)} W^{(m)}), \tag{10}$$

where $X^{(m)}$ is the input feature matrix at network layer m, $W^{(m)}$ is the trainable weight matrix, $\tilde{D}$ is the diagonal degree matrix of adjacency matrix A, and σ is an activation function.

To incorporate edge information, we extend the standard GCN by adding edge embeddings. Specifically, as shown in Eq. (11), where $X_n^{(m)}$ and $X_e^{(m)}$ are the node and edge embedding at layer m. Other symbols are the same as Eq. (10). The GCN layer takes both node and edge embeddings generated by the initial encoding module as illustrated in Fig. 2.

$$X_n^{(m+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} A \tilde{D}^{-\frac{1}{2}} (X_n^{(m)} + X_e^{(m)}) W^{(m)}). \tag{11}$$

3.4 Final Readout

The final multi-view representations Z_g will be obtained by aggregating the view-level representations Z_i across all views. Each view-level representation are generated by element-wise readout function across all node embedding. Equation (12) show the concatenated mean readout, where n represents the number of nodes in the i-th view, k represents the view number.

$$Z_i = \frac{1}{n} \Sigma_{i=1}^{n} h_i;\ Z_g = \Sigma_{i=1}^{k} Z_i. \tag{12}$$

3.5 Architecture

TreePool is mainly used in supervised framework as depicted in Fig. 2. Nonetheless, it can be conveniently adapted to an unsupervised model as well, and we have conducted experiments on both architectures. In this section, we aim to clarify the supervised and unsupervised architectures and provide details on how the objective function is computed.

Supervised Architecture. In supervised model architecture, we follow the previous works, which mainly have two basic architectures [17]. One architecture is global, in which coarsening operations are performed after multiple GNN layers pass messages through the graph. The other architecture is hierarchical, in which the single GNN layer and coarsening layer are arranged alternately. We represent the global one and the hierarchical one as $TreePool_g$ and $TreePool_h$.

For each framework, we combined the simplified contrastive loss, (i.e., L_{scl}) and task-related loss as the final loss can be calculated as followed:

$$L = \alpha L_{scl} + \beta L_{task}, \tag{13}$$

where α and β are weight parameters that control the contribution of two losses. It is noted that L_{scl} may represents the sum of multiple contrastive loss if the subgraphs sequence length is at least 2. The process of supervised learning is shown in Algorithm 1.

Unsupervised Architecture. In the unsupervised learning model, TreePool serves as a pre-training stage and learns the program representations by the task of local-local contrastive learning between the adjacent subgraph views.

Algorithm 1: Supervised Representation Learning

Input: the training set v_{train} and the number of pooling layers N
Output: The classified result
1 Initialize supervised model M's parameters θ ;
2 **while** M *not converged* **do**
3 $L_{scl} \leftarrow 0$;
4 Sample a mini-batch of graphs and get the total adjacent matrix of graph A_0, the encoded node feature X_n^0, and edge feature X_e^0 ;
5 **for** $i \in \{0, 1, ...N-2\}$ **do**
`// compute with GNN layer`
6 $X_n^i \leftarrow GNN(A_i, X_n^i, X_e^i)$;
`// filter new subgraph`
7 $A_{i+1}, X_n^{i+1} \leftarrow filter(A_i, X_n^i)$;
8 Calculate L_{scl}^i for this layer;
9 $L_{scl} \leftarrow L_{scl} + L_{scl}^i$;
10 **end**
11 Calculate task-related loss L_{task} ;
12 Update θ by minimizing $\alpha L_{scl} + \beta L_{task}$;
13 **end**
14 **return** *the classified result using trained* M

Unlike Algorithm 1, the graph pooling stage acts as a pre-training step in which the graph embedding is trained unsupervised only based on the simplified contrastive loss L_{scl}. Then the encoder's parameters are frozen for downstream tasks and train the new supervised model using only the task loss L_{task}. The final graph readout are the same as supervised architecture.

4 Experimental Setting

4.1 Research Questions

In order to evaluate TreePool, we design and answer the following research questions with the experimental results:

RQ1: **Comparison of Pooling baselines** How does TreePool perform compared to recent pooling methods on the benchmark datasets?

RQ2: **Ablation Study** How effective are the components of TreePool, such as the projection layer and simplified process?

RQ3: **Unsupervised Scene** Can the mutual information loss be transferred to an unsupervised learning framework?

RQ4: **Study of Views** How the number and generation of view affect the final result? Which view affects the final result most?

RQ5:**Model Analysis** What is the impact of hyper-parameters on the performance of TreePool?

4.2 Datasets

In experiments, we evaluate TreePool on graph classification task with OJ-104 dataset [23]. The dataset contains C/C++ code collected from Online Judge system in 104 classes. Programs with the same target label have the same functionality. We also created a new dataset named OJ-DEEP which includes the top 20% deepest graphs (no less than 15) in order to more accurately evaluate the effectiveness of TreePool on large AST representation. OJ-DEEP contains only the large graphs on the origin dataset. The statistical information of the two datasets is summarized in Table 1.

4.3 Baselines

Table 1. Summary of the datasets

Dataset	Graph	Node			Depth		
		avg	min	max	avg	min	max
OJ-104	51,976	189.56	29	7,027	13.31	6	76
OJDEEP	13,371	259.25	42	7,027	17.92	15	76

Since TreePool is designed initially for the supervised framework, we report the performance between TreePool and the supervised graph pooling methods in the study of RQ1, while also reporting TreePool with the unsupervised learning framework in the study of RQ4. We select the following methods as baselines, which could be divided into two groups: (1) supervised graph pooling approaches SAGPool [17], DiffPool [31], and Graph U-net [9], and (2) unsupervised graph contrastive learning methods DGI [27], GRACE [37].

SAGPool [17] use the graph convolution layer to calculate the self-attention score in each pooling stage, which indicates the selection of the next-level nodes with the pooling ratio. The SAGPool has two architectures: the hierarchical architecture $SAGPool_h$ and the global architecture $SAGPool_g$.

DiffPool [31] learns differentiable soft cluster assignments, which indicates the relation between the node of the origin and the coarsened graph, to coarsen graph. The new adjacency matrix and feature matrix are the aggregation of the original graph generated with the assignment matrix.

Graph U-net [9] has an encoder-decoder architecture with gPool and gUnpool operations. gPool operation pools the graph into top-k nodes subgraphs based on the value produced from the trained projection vector. gUnpool recovers the node embedding from the coarsened graph to the original graph.

DGI [27] relies on maximizing mutual information between patch representations and corresponding high-level summaries of graphs, which are derived using graph convolutional network architectures.

GRACE [37] generates two correlated graph views by randomly performing corruption, specifically, removing edges and masking node features. the model maximizes the mutual information between the two views.

4.4 Implementation Details

Following the AST extraction and the dataset splitting step of TBCNN [23], for both datasets, we randomly split them with the train/validation/test ratio of 3.2:0.8:1. We use the same data splitting for all methods and each experiment is run 5 times to calculate the average performance for fairness. We report the mean accuracy along with the standard deviation of five runs for each model and dataset. The standard deviation is represented in percentages while the accuracy is in decimals. We train all the models on a maximum of 100 epochs with 10-epoch patience for early stopping. We used the Adam optimizer and hyperparameter selection strategy. The learning rate is selected from the range [0.01,0.05,0.005,0.0001,0.0005]. All the baseline models are reproduced with Pytorch 1.8 on two parallel GPUs and adjust the parameters to get a better result.

We set the subgraph view number 3, indicating that the original graph will be separated into three subgraphs and pooled twice. The hyperparameter of the subgraph view number is studied in the Sect. 5.4. For the supervised framework, we set the parameter α and β in the loss function as 0.5 and 0.5.

The depth and type of a node in an AST are initially encoded using two one-hot vectors, which are then compressed into a 300-dimensional vector via a single-layer MLP. To ensure the same level of expressiveness across different graph learning model architectures, equal numbers of layers (n = 3) are used, and the hidden dimensions are set equal to the embedding dimension. Following the GNN layer, a ReLU layer and dropout layer are applied[1].

4.5 Evaluation Metric

As defined in Eq. (14), we use accuracy for evaluation, which measures the proportion of correctly predicted labels out of the total number of instances.

$$\mathrm{Acc} = \frac{\sum_{i=1}^{|\mathcal{D}|} \mathcal{I}(\hat{y}, y)}{|\mathcal{D}|} \tag{14}$$

where $\hat{y}$ is the model's prediction, y is the ground truth label, $|\mathcal{D}|$ is the total number of samples, $\mathcal{I}(\cdot, \cdot)$ the indicator function which returns 1 if the predicted label is the same as the true label, and 0 otherwise.

5 Results

5.1 RQ1: Comparison of Pooling Baselines

To answer the first question, we compare TreePool with 4 baseline graph pooling methods on the supervised learning framework on OJ-104 and OJ-DEEP. Based on the results in Table 2, we have the following observations.

[1] Our implementation is available at: https://github.com/codingClaire/TreePool.

Deeper Graphs are Harder to Learn the Graph Representation. We observe that the performance of most methods is higher on the OJ-104 compared to OJ-DEEP, indicating that the former is an easier dataset for graph classification. The result indicates that graph-based models may perform badly as the code snippets become longer and complicated.

Original Node Information is Essential for Code Representation. Among the baseline methods, we note that Graph U-net achieves the highest accuracy on both datasets. In contrast, DiffPool achieves the lowest. We found out that DiffPool requires multiple loss functions, which makes the model difficult to converge. Although SAGPool or DiffPool works well on small graph classification, the result in Table 2 shows that when the scale of the graph dataset increases, they are more likely to lose information and be unable to fit this situation. For example, DiffPool generates new virtual nodes for the views after the original graph as the first view, which may be the reason of its results.

TreePool Outperforms All Baseline Methods. We observe that TreePool outperforms all the baseline methods on the two datasets in terms of accuracy. In particular, in OJ-104 and OJ-DEEP datasets, $TreePool_g$ achieves better performance compared to the best baseline methods with an improvement of 0.2% and 2.9% respectively. The corresponding relative improvements are 1.1% and 3.3% in $TreePool_g$. The results show that TreePool is a promising method for code classification, with comparable performance to SAGPool and Graph U-net.

Global Framework is Better than Hierarchical Framework. Compare to the result of the hierarchical and global framework of SAGPool and TreePool, we found out that the global framework works better. Compare to the hierarchical framework, the global framework of SAGPool relatively improves by 1.1% and 1.8% in OJ-104 and OJ-DEEP. For TreePool, we observe similar results in which the improvements are 1.2% and 0.8% respectively. The results indicate that a simple framework can make the result of graph representation learning

Table 2. Comparison of graph pooling methods (Accuracy)

Method	OJ-104	OJ-DEEP
$SAGPool_h$	0.9070 ± 3.15	0.8927 ± 2.87
$SAGPool_g$	0.9169 ± 3.28	0.9091 ± 2.73
$DiffPool$	0.7148 ± 0.90	0.7362 ± 1.68
$Graph\ U-net$	0.9430 ± 0.28	0.9024 ± 0.58
$TreePool_h$	0.9453 ± 0.18	0.9287 ± 0.32
$TreePool_g$	0.9563 ± 0.12	0.9323 ± 0.32
w/o GNN encoder	0.8943 ± 0.08	0.8734 ± 0.40
w/o projection layer	0.9451 ± 0.06	0.9175 ± 0.42
w/o simplified process	0.9527 ± 0.07	0.9257 ± 0.71

effective. One hypothesis to explain is that when multiple losses are added to the downstream loss, the hierarchical model may be more complicated to optimize.

Answer of RQ1: Our experiments demonstrate that TreePool outperforms all the baselines in terms of accuracy. In particular, TreePool achieves better performance with 1.1% and 3.3% improvements on both datasets, respectively.

5.2 RQ2: Ablation Study

To further explore the effectiveness of different components in TreePool and answer RQ2, we conduct multiple ablation experiments using the OJ-DEEP dataset and evaluate the effectiveness of the TreePool model on the following models. Besides the original architectures, we report the best results with the hierarchical and global architectures. The results are shown in Table 2.

- w/o GNN encoder: The model is trained without GNN, which means that the node and edge embeddings are directly fed into the pooling layer.
- w/o projection layer: The model is trained without the projection layers for the graphs before and after pooling.
- w/o simplified process: The model is trained without simplified loss, which means trains by the original contrastive learning loss proposed by DGI.

Our ablation studies demonstrate the importance of each component in the TreePool model. The GNN component helps capture the relationships between nodes and edges in the graph, while the projection layer helps to align the representations between the global and local levels. Although the result of model without simplified calculation is close to the original results, the simplified calculation still works with less calculation as previous analysis.

Answer of RQ2: We observe a consistent performance improvement on different modifications of original contrastive loss and framework, confirming their benefits on the pooling methods.

5.3 RQ3: Unsupervised Scene

We aim to explore whether simplified contrastive loss can transfer to the unsupervised learning scene. We conduct experiments on 2 view layers of $TreePool_g$ in unsupervised learning framework and compare the model with our unsupervised baselines. The detailed results are shown in Table 3.

In general, compared to a supervised learning framework, the performance of the unsupervised learning framework is worse due to the absence of labels. In both the DGI and GRACE methods, DGI outperformed GRACE significantly

in both datasets. When compared to DGI, TreePool, which utilizes an unsupervised framework, performed slightly worse on the OJ-104 dataset with a decrease of approximately 0.4%. However, on the OJ-DEEP dataset, TreePool's performance showed a remarkable improvement, surpassing DGI by 3.0% and even outperforming its performance on the OJ-104 dataset.

Table 3. Comparison of graph contrastive methods (Accuracy)

Method	OJ-104	OJ-DEEP
DGI	0.9191 ± 0.59	0.9001 ± 0.54
$GRACE$	0.8761 ± 0.54	0.8204 ± 0.68
$URL + TreePool$	0.9154 ± 0.24	0.9270 ± 0.35

Answer of RQ3: Our experiments demonstrate that TreePool is effective in both supervised and unsupervised learning tasks. TreePool in unsupervised representation learning obtains significant improvements on the OJ-DEEP dataset.

5.4 RQ4: Study of Views

To answer the fourth question, we explore the impact of views during pooling, including the way of view generation, the number of views, and the impact of multiple views on performance.

View Generation. TreePool generates multiple subgraph views based on the hierarchical relation of tree architecture. Based on the hierarchical form of TreePool, we change the way of generating views to study whether this way of view generation is reasonable. With these views, the model will calculate simplified contrastive loss and further predict the result as the original model.

$TreePool_A$: Randomly select the nodes with the same amount in the subgraph view generation module instead of selecting hierarchically.

$TreePool_B$: Randomly select half of amount of nodes from hierarchical views.

The best result of all architectures is shown in Table 4. When the nodes are randomly chosen, the accuracy of $TreePool_A$ will decrease by 0.2% and 1.4% in OJ-104 and OJ-DEEP, indicating the importance of hierarchical view generation. In $TreePool_B$, the model only chooses half of the nodes from each hierarchical view, which influence the process of calculating the simplified contrastive loss. The accuracy of $TreePool_B$ decreases by 0.4% and 1.0%. The result of $TreePool_A$ shows that it is meaningful to generate views hierarchically. Furthermore, the experiments of $TreePool_B$ prove the effectiveness of preserving the information from all nodes of views.

View Numbers. Another factor that may affect the pooling effect is the number of subgraph view numbers. More views leads to more pooling times and calculation. We conducted experiments to investigate the impact of view numbers. Specifically, we explored the effects of 2 to 6 views on supervised $TreePool_h$.

As shown in Table 5, the best result is 1.7% higher than the worst result, while only 0.9% for OJ-104. We expected that when the number of view layers becomes higher, even OJ-DEEP, which contains larger depth data, may have a minimal difference in using contrastive learning before and after pooling due to the small space size. Therefore, in this situation, employing TreePool may lead to a decrease in performance and is not cost-effective in terms of efficiency. However, selecting a smaller number of views, such as 2, may cause the model to lose some hierarchical information by comparing two graphs with significant differences. Therefore, determining a suitable number of views is crucial for the results and is one of the future works that deserves investigation.

Table 4. Result of view generation study (Accuracy)

Number	OJ-104	OJ-DEEP
$TreePool_A$	0.9540 ± 0.12	0.9161 ± 0.38
$TreePool_B$	0.9523 ± 0.02	0.9193 ± 0.24
$TreePool$	0.9563 ± 0.12	0.9323 ± 0.32

Table 5. Result of View Number Study (Accuracy)

View	OJ-104	OJ-DEEP
2	0.9491 ± 0.18	0.9127 ± 0.38
3	0.9453 ± 0.18	0.9287 ± 0.32
4	0.9535 ± 0.14	0.9248 ± 0.30
5	0.9540 ± 0.07	0.9283 ± 0.34
6	0.9523 ± 0.20	0.9248 ± 0.27

View Importance. To study which readout of view contribute more to the final result, we conduct experiments on $TreePool_g$ with total view numbers of 2,3,4 on OJ-DEEP to check which view matters more on the performance of the model. For each model, we maintain the consistency with the calculation of loss function but used every single view's readout generated through pooling to get the final graph representation. All results are presented in Fig. 4, where last data point of the line represents the final result of the concatenated mean readout.

We observed that when view number is 3 and 4, both single-view and multi-view models exhibited a similar trend in which an decreased number of nodes

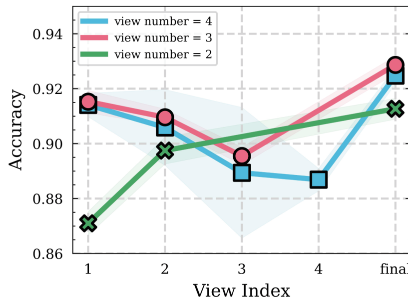

Fig. 4. The result of the experiment of view importance study on OJ-DEEP.

led to worse classification performance. Overall, the results obtained using single views were inferior to those achieved with multi-view models. These findings demonstrate that the multi-view approach introduced more information into the model, resulting in more comprehensive representations being learned.

Answer of RQ4: We examine the impact of views on model performance and demonstrate the effectiveness of hierarchically generating the views, as well as the importance of properly selecting view numbers and employing multi-view training methods.

5.5 RQ5: Model Analysis

To answer RQ5, we analyzed two hyperparameters that affect model performance: the layer number and type of GNN encoder.

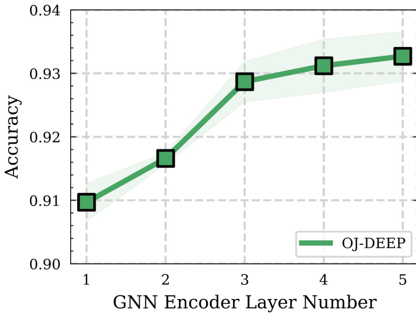

Fig. 5. The effect of the number of GNN encoder layers on performance.

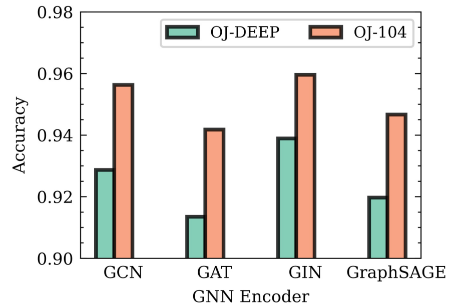

Fig. 6. The result of different GNN types.

Layer Number. We investigate the impact of the number of layers in GNN encoder on the performance of our proposed method. By varying the number of layers, we aim to find the optimal depth of the GNN model for our task.

The result in Fig. 5 indicate that the model perform best when the GNN encoder has three layers, which is also slightly improved with four or more layers. However, this leads to an increase in model parameters and training time. We speculate that increasing the number of layers in the GNN encoder could result in over-smoothing, as commonly observed. Due to the high depth of the overall OJ-DEEP dataset, no such issue is observed within the five layers.

Type of GNN Encoder. We analyze the impact of different GNN encoders on the performance of our proposed method. Specifically, we evaluate three popular GNN-based models as encoders, including GAT [26], GIN [30], and GraphSAGE [10], on the code classification task.

The results in Fig. 6 show that GIN outperforms all other GNN encoders followed by GCN and GraphSAGE. GAT has inferior performance compared to the other models, indicating its limitation in capturing global-level graph structures. Although GIN performs better than GCN, it is computationally expensive, as it requires multiple layers of MLPs to aggregate the node features. Therefore, the respective strengths of GNN models must be considered along with their limitations for specific datasets and tasks.

Answer of RQ5: Our experiments provide a thorough evaluation of how the number and type of GNN layers can influence the performance, and we analyze their respective strengths and limitations.

6 Discussion

The effectiveness of TreePool can be attributed to its ability to capture the information flow within the graph and coarsen it in a hierarchical way. By maximizing the mutual information between the graph before and after pooling, TreePool is able to disperse the information from leaf nodes of lower layers to multiple subgraph views based on semantic relationships. By conducting a concatenated readout, the generated multi-view representation of the graph can capture the hierarchical information of tree-like graphs.

Despite the promising results of TreePool, there are several potential threats and limitations that should be addressed. One limitation is the datasets we choose for evaluation. We only conduct our experiments in C/C++ language, so they may be not representative of other languages. In the future, we will evaluate more datasets in other programming languages. Another limitation is that TreePool is compared solely with graph representation methods due to the method being designed for tree-like graphs in GNN-based models. As the large language models have taken great advantages in the code intelligence area, we will extend our approach combined with other sequential models.

7 Related Work

7.1 AST-Based Code Representation Learning

Abstract Syntax Trees are widely used for encoding program representations on code-related tasks, which can be categorized into tree-based and graph-based approaches. Graph Neural Networks are less commonly used in tree-structured AST approaches, where Basic Recursive Neural Networks [6] and Convolutional Neural Networks [22] are more prevalent. On the other hand, some works focus on the graph perspective and employ modified GGNN [8], combinations with RNN or sequential models [25], convolutional GNN [16], Graph Attention Network [29] for encoding AST nodes and edges. To fully taking into account the topology differences between Abstract Syntax Tree and other typical graphs,

certain existing methods add edges to enhance the AST [5,32]. Compared to these methods, our approach introduces the graph pooling module to directly extract more effective graph representations from the original AST.

7.2 Graph Pooling

Based on the differences in the coarsening process, the pooling methods can be classified into two categories: node clustering pooling and node drop pooling. Node clustering pooling [4,21] aims to partition nodes into clusters and subsequently treat these clusters as new nodes in a coarsened graph. However, these approaches are subject to a drawback regarding time and storage complexity, which arises from the computation of the dense cluster assignment matrix. Node drop pooling [9,34] utilizes learnable scoring functions to keep nodes with higher significant scores. These methods are prone to losing a significant amount of nodes, resulting in inevitable information loss.

7.3 Neural Estimation of Mutual Information Maximization

The goal of contrastive learning is to improve the agreement between positively paired samples, which are jointly sampled, while contrasting them with negatively paired samples that are independently sampled. Estimating Mutual Information (MI) is a commonly used technique but difficult to calculate. Neural estimation of mutual information is extensively studied in unsupervised and self-supervised learning. Previous works have used it to learn graph representations, including DGI [27] between patch representations of subgraphs and global high-level summaries of the graph, Deep InfoMax [12] between input data and learned high-level representations and InfoGraph [24] between the graph-level representation and the representations of substructures of different scales. Some works also leverage a contrastive object between views, such as GRACE [37] and MVGRL [11]. The former method contrasts between two correlated graph views by randomly performing corruption, while the latter between views from first-order neighbors and a graph diffusion.

8 Conclusion

In this paper, we propose a novel graph pooling method named TreePool, which aims to address the challenge of generating an informative graph representation for Abstract Syntax Trees, a kind of tree-like code graph. TreePool incorporates spatial proximity of nodes and local-local contrastive learning to preserve leaf information and hierarchical information. With TreePool, code representation learning with Abstract Syntax Trees can be accomplished more effectively by leveraging the hierarchical structure of code graphs. The improved code representation can be used in multiple code tasks for collaborative computing. Our experiments demonstrate the effectiveness of TreePool in both supervised and unsupervised frameworks for the code classification task. In the future, we plan

to explore the potential of TreePool in other code languages and tasks, as well as investigate its scalability to sequential language models.

Acknowledgement. The research is supported by the National Key R&D Program of China under grant No. 2022YFF0902500, the Guangdong Basic and Applied Basic Research Foundation, China (No. 2023A1515011050). Liang Chen is the corresponding author.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques and tools (2020)
2. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code (2019)
3. Belghazi, M.I., et al.: Mutual information neural estimation. In: International Conference on Machine Learning, pp. 531–540. PMLR (2018)
4. Bianchi, F.M., Grattarola, D., Alippi, C.: Spectral clustering with graph neural networks for graph pooling. In: International Conference on Machine Learning, pp. 874–883. PMLR (2020)
5. Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O.: Generative code modeling with graphs. In: International Conference on Learning Representations (2018)
6. Chakraborty, S., Ding, Y., Allamanis, M., Ray, B.: Codit: code editing with tree-based neural models. IEEE Trans. Softw. Eng. **48**(4), 1385–1399 (2020)
7. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. arXiv Computation and Language (2020)
8. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: International Conference on Learning Representations (2018)
9. Gao, H., Ji, S.: Graph u-nets. In: international Conference on Machine Learning, pp. 2083–2092. PMLR (2019)
10. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
11. Hassani, K., Khasahmadi, A.H.: Contrastive multi-view representation learning on graphs. In: International Conference on Machine Learning, pp. 4116–4126 (2020)
12. Hjelm, R.D., et al.: Learning deep representations by mutual information estimation and maximization. arXiv preprint arXiv:1808.06670 (2018)
13. Hussain, Y., Huang, Z., Zhou, Y., Wang, S.: CodeGRU: context-aware deep learning with gated recurrent unit for source code modeling. Inf. Softw. Technol. **125**, 106309 (2020)
14. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
15. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
16. LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020)
17. Lee, J., Lee, I., Kang, J.: Self-attention graph pooling. In: International Conference on Machine Learning, pp. 3734–3743. PMLR (2019)
18. Li, J., Peng, J., Chen, L., Zheng, Z., Liang, T., Ling, Q.: Spectral adversarial training for robust graph neural network. IEEE TKDE (2022)

19. Li, J., Xie, T., Chen, L., Xie, F., He, X., Zheng, Z.: Adversarial attack on large scale graph. IEEE TKDE **35**(1), 82–95 (2021)
20. Liu, Y., Chen, L., He, X., Peng, J., Zheng, Z., Tang, J.: Modelling high-order social relations for item recommendation. IEEE TKDE **34**(9), 4385–4397 (2020)
21. Ma, Y., Wang, S., Aggarwal, C.C., Tang, J.: Graph convolutional networks with eigenpooling. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 723–731 (2019)
22. Mou, L., Li, G., Jin, Z., Zhang, L., Wang, T.: TBCNN: a tree-based convolutional neural network for programming language processing. arXiv preprint arXiv:1409.5718 (2014)
23. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 30 (2016)
24. Sun, F.Y., Hoffmann, J., Verma, V., Tang, J.: Infograph: unsupervised and semi-supervised graph-level representation learning via mutual information maximization. arXiv preprint arXiv:1908.01000 (2019)
25. Tarlow, D., et al.: Learning to fix build errors with graph2diff neural networks. arXiv preprint arXiv:1911.01205 (2019)
26. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
27. Veličković, P., Fedus, W., Hamilton, W.L., Liò, P., Bengio, Y., Hjelm, R.D.: Deep graph infomax. arXiv preprint arXiv:1809.10341 (2018)
28. Wagstaff, E., Fuchs, F., Engelcke, M., Posner, I., Osborne, M.A.: On the limitations of representing functions on sets. In: International Conference on Machine Learning, pp. 6487–6494. PMLR (2019)
29. Wang, Y., Li, H.: Code completion by modeling flattened abstract syntax trees as graphs. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 14015–14023 (2021)
30. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? arXiv preprint arXiv:1810.00826 (2018)
31. Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., Leskovec, J.: Hierarchical graph representation learning with differentiable pooling. In: Advances in Neural Information Processing Systems, vol. 31 (2018)
32. Zhang, K., Wang, W., Zhang, H., Li, G., Jin, Z.: Learning to represent programs with heterogeneous graphs. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 378–389 (2022)
33. Zhang, Z., Zhuang, F., Zhu, H., Shi, Z., Xiong, H., He, Q.: Relational graph neural network with hierarchical attention for knowledge graph completion. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 9612–9619 (2020)
34. Zhang, Z., et al.: Hierarchical graph pooling with structure learning. arXiv preprint arXiv:1911.05954 (2019)
35. Zheng, Y., Pan, S., Lee, V., Zheng, Y., Yu, P.S.: Rethinking and scaling up graph contrastive learning: an extremely efficient approach with group discrimination. In: Advances in Neural Information Processing Systems, vol. 35, pp. 10809–10820 (2022)
36. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: DevIGN: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems, vol. 32 (2019)
37. Zhu, Y., Xu, Y., Yu, F., Liu, Q., Wu, S., Wang, L.: Deep graph contrastive representation learning. arXiv preprint arXiv:2006.04131 (2020)