A Tree-structured Transformer for Program Representation Learning

Wenhan Wang³, Kechi Zhang¹,², Ge Li¹,², Shangqing Liu³, Zhi Jin¹,², Yang Liu³
¹Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
²Institute of Software, EECS, Peking University
³Nanyang Technological University
wenhan.wang@ntu.edu.sg, {zhangkechi, lige}@pku.edu.cn, shangqin001@ntu.edu.sg, zhijin@pku.edu.cn, yangliu@ntu.edu.sg

Abstract

When using deep learning techniques to model program languages, neural networks with tree or graph structures are widely adopted to capture the rich structural information within program abstract syntax trees (AST). However, long-term/global dependencies widely exist in programs, and most of these neural architectures fail to capture these dependencies. In this paper, we propose Tree-Transformer, a novel recursive tree-structured neural network which aims to overcome the above limitations. Tree-Transformer leverages two multi-head attention units to model the dependency between siblings and parent-children node pairs. Moreover, we propose a bi-directional propagation strategy to allow node information passing in two directions: bottom-up and top-down along trees. By combining bottom-up and top-down propagation, Tree-Transformer can learn both global contexts and meaningful node features. The extensive experimental results show that our Tree-Transformer outperforms existing tree-based or graph-based neural networks in program-related tasks with tree-level and node-level prediction tasks, indicating that Tree-Transformer performs well on learning both tree-level and node-level representations.

1 Introduction

Program source code is a highly-structured data form. When using deep neural networks to model them, many advanced works leverage tree-structured neural networks [Mou et al., 2016; Zhang et al., 2019; Wang et al., 2020; Bui et al., 2021] or graph neural networks [Allamanis et al., 2018; Liu et al., 2021; Hellendoorn et al., 2020] to incorporate structural information like abstract syntax tree (AST) and data/control flow.

However, although graph-structured or tree-structured neural networks have achieved promising results on multiple code-related tasks, they also suffer from the following limitations: 1) Graph neural networks rely on the message passing to capture $k$-th local neighborhood information while the global dependencies among any pair of nodes are missed. As previous researchers have addressed [Hellendoorn et al., 2020; Liu et al., 2021], capturing long-term dependencies is crucial for source code understanding models. As shown in Figure 3(a), when the hyper-parameter $k$ is set to 2, node A can only capture the information of the nodes directly connected with A or the information from the neighbors of A. Furthermore, converting programs into graphs with different edge types to represent the program semantics is nontrivial and costly. Building program graphs require experts to leverage various static analysis techniques, and some of these techniques cannot be adopted to different program languages or program comprehension tasks. Finally, the principle to extract which

Preprint. Under review.
Figure 1: The comparison of three neural networks, where the arrow represents the direction of information propagation and the dash area is the context that node A can capture.

2-hop
1-hop

(a) Graph Neural Network  (b) (Recursive) Tree-structured Neural Network  (c) Tree-Transformer

The type of semantics like control flow, data flow cannot reach a consensus. 2) Tree-structured neural networks update node representations by receiving information from their children. Many existing tree-structured neural networks [Goller and Kuchler, 1996; Tai et al., 2015] belong to recursive neural network [Goller and Kuchler, 1996], where node representations are updated recursively by a neural network unit with the bottom-up topological order. Compared with GNNs, in recursive neural networks, the root in the tree captures all descendants’ features by the bottom-up information propagation, and it can be regarded as knowing the global context of the whole tree. However, in these models, the information propagation only follows one direction, i.e., from leaf nodes to the root node, hence the non-root nodes cannot capture global contexts. This is illustrated in Figure 3 (b). We can see that node A can only capture the context information from its descendants. For low-level nodes, their context is small, so (recursive) tree-structured neural networks cannot learn meaningful and well-contextualized representations for these nodes.

To address these challenges, in this paper, we propose our novel Tree-Transformer, a transformer-based approach to learn the global dependencies among any pair of nodes in a tree. Specifically, we define the bidirectional propagation method along opposite directions based on multi-head attention [Vaswani et al., 2017]. The bidirectional propagation is achieved with two different Tree-Transformer units: a bottom-up unit and a top-down unit, where the bottom-up unit aggregates the contextual information from leaves to the root node by defining the parent nodes as queries and their children as keys/values. In contrast, the top-down unit distributes the root information to its descendants by regarding the children as queries, and their parents are keys-values. In this way, the global dependencies among any pair of nodes in the tree can be captured by Tree-Transformer. The schematic diagram is shown in Figure 3 (c), where node A can capture the global contexts.

We conduct extensive experiments on both the tree-level prediction task, e.g., program classification, and node-level prediction task, e.g., wrong operator localization on different public-available datasets to demonstrate the effectiveness of our proposed approach. Overall, we highlight our main contributions as follows:

- We propose Tree-Transformer, a recursive tree-structured neural network which formulates the parent-child and sibling relations on the trees with multi-head attention to learn vector representations for trees and their nodes.

- We innovate a bidirectional information propagation method both from the bottom-up and top-down manner on the Tree-Transformer to capture the global dependencies among any pair of nodes.

- Extensive experiments on tree-level and node-level prediction tasks on program syntax trees illustrate that our approach outperforms existing tree-structured neural networks, e.g., Tree-LSTM, and graph neural networks, e.g., GIN, by a significant margin. To evaluate Tree-Transformer for node-level prediction, we proposed a novel wrong operator localization and repair task.
2 Related Work

There has been a long research interest in applying deep neural networks to tree-structured data. The earliest work on tree-structured neural networks is recursive neural network [Goller and Kuchler 1996]. Recursive neural networks calculate the representation of a tree by accumulating node representations in a bottom-up manner. The recursive architecture further inspires later works [Tai et al., 2015; Teng and Zhang, 2017; Ahmed et al., 2019; Wang et al., 2020]. For example, Tai et al. [2015] proposed Tree-LSTM, which uses an LSTM unit to replace the fully-connect neural network unit in recursive neural networks. Teng and Zhang [2017] proposed a top-down Tree-LSTM inspired by the bi-directional sequential LSTM.

Apart from recursive neural networks, some researchers built tree-structured neural networks with different formulas [Mou et al., 2016; Bui et al., 2021]. For example, Mou et al. [2016] proposed tree-based convolutional neural network (TBCNN). Instead of gathering information bottom-up recursively, TBCNN applies a convolution window over subtrees. The idea of tree convolution is similar to graph convolution in message-passing graph neural networks. In recent years, as the popularity of graph neural networks grows, researchers started directly applying GNNs to trees [Yao et al., 2021; Puri et al., 2021].

Recently there have been some attempts to integrate tree structures into the popular transformer model. Some of these works manipulate the attention mechanism or position encoding to provide the model with structural information [Shiv and Quirk, 2019; Li et al., 2021; Zügner et al., 2021; Peng et al., 2021], while others directly build recursive-structured transformers on trees [Ahmed et al., 2019]. For example, Shiv and Quirk [2019] proposed a novel position encoding technique to extend transformers to N-ary trees. Zügner et al. [2021] proposed Code Transformer, which integrates multiple distance metrics of AST nodes into the relative position encoding of transformers. Several works use masks on the self-attention component in transformers to address tree structures [Li et al., 2021]. In these approaches, if two nodes are not connected by a tree edge, their attention is masked out. Peng et al. [2021] proposed TPTrans, which integrates the information of paths in trees by encoding the paths as the position encodings in a transformer model. Ahmed et al. [2019] proposed a recursive-structured transformer by performing self-attention on siblings. Their model includes two different variants built for natural language constituency tree and dependency tree.

3 Approach

The overall architecture of Tree-Transformer is illustrated in Figure 2(a). Tree-Transformer computes node representations with two steps: 1) Bottom-up propagation, which propagates the node messages recursively from children to parents to obtain the bottom-up states of nodes; 2) Top-down propagation,
which distributes the learned contextual information from the parent node to their children. After that, the obtained top-down node states can be utilized for the node-level prediction tasks. Similar to GNNs, we can pool the node representations learned by Tree-Transformer into a single vector for tree-level prediction tasks, such as tree classification.

### 3.1 Bottom-up Propagation

Formally, a code snippet $c$ can be parsed into an AST $\mathcal{T} = (\mathcal{V}_{\text{leaf}}, \mathcal{V}_{\text{non}})$, where $\mathcal{V}_{\text{leaf}}$ and $\mathcal{V}_{\text{non}}$ denote the set of AST leaf nodes and non-leaf nodes respectively. Any node $i \in \mathcal{V}_{\text{non}}$ connects with a set of child nodes $c_i = \{i_1, i_2, ..., i_n\}$ where $n$ is the number of child nodes for $i$. In ASTs, different nodes may have different numbers of $n$. For any node $v \in \mathcal{T}$, we utilize a learnable embedding matrix $E$ to obtain the initial node embedding $e_v \in \mathbb{R}^d$, where $d$ is the dimension of node embeddings.

To capture the children $\mathcal{V}_{\text{child}}$ information for the non-leaf node $i$, we design the bottom-up propagation based on the multi-head attention [Vaswani et al. 2017] as shown in Figure 2 (b). The bottom-up Tree-Transformer unit obtain the bottom-up states of nodes in a bottom-up manner similar to recursive neural networks [Goller and Kuchler 1996], i.e., a node $i$’s bottom-up state $h_{i\uparrow}$ is updated from its initial node embedding $e_i$ and its children’s bottom-up states $H_{c_i\uparrow} = \{h_{i_1\uparrow}, h_{i_2\uparrow}, ..., h_{i_n\uparrow}\}$. If the child nodes of $i$ are leaf nodes, i.e., $\{i_1, ..., i_n\} \subseteq \mathcal{V}_{\text{leaf}}$, $H_{c_i\uparrow}$ are equal to their leaf node embeddings i.e., $H_{c_i\uparrow} = (e_1, e_2, ..., e_n)$.

In a bottom-up Tree-Transformer unit, we first apply a fraternal self-attention $\text{MultiHead}_{f\uparrow}$ on $H_{c_i\uparrow}$ to model the sibling dependency between nodes in $c_i$. Then we use a parental multi-head attention $\text{MultiHead}_{p\uparrow}$, which is the same with the vanilla attention in [Vaswani et al. 2017], to capture the dependency between $i$ and its children. In the parental attention of $i$, the query is its initial node embedding $e_i$, and the key/value are the output of the fraternal attention on $i$’s children. The output of the parental attention is thereafter used to update the bottom-up state of $i$. Like the sequential transformer model, each attention module in the Tree-Transformer unit is followed by layer normalization and residual connection. Finally, we use a position-wise feed-forward layer same as [Vaswani et al. 2017] calculates the bottom-up state. Formally, the bottom-up Tree-Transformer unit calculates $h_{i\uparrow}$, the bottom-up state of $i$ by:

$$
H'_{c_i\uparrow} = \text{MultiHead}_{f\uparrow}(H_{c_i\uparrow}, H_{c_i\uparrow}, H_{c_i\uparrow})
$$

(1)

$$
H'_{c_i\uparrow} = \text{LayerNorm}(H'_{c_i\uparrow} + H_{c_i\uparrow})
$$

(2)

$$
A'_{\uparrow} = \text{MultiHead}_p(e_i, H'_{c_i\uparrow}, H'_{c_i\uparrow})
$$

(3)

$$
A'_{\uparrow} = \text{LayerNorm}(A + e_i)
$$

(4)

$$
h_{i\uparrow} = \text{LayerNorm}(\text{FFN}\uparrow(A'_{\uparrow}) + A'_{\uparrow})
$$

(5)

To capture the sibling order information in the tree, we apply position encodings [Vaswani et al. 2017] in the fraternal self-attention $\text{MultiHead}_{f\uparrow}$. This allows our model to handle sibling orders for arbitrary trees, while most existing order-sensitive models on trees [Tai et al. 2015; Shiv and Quirk 2019] only work on trees with a fixed branching factor (N-ary trees).

### 3.2 Top-down Propagation

After bottom-up propagation, Tree-Transformer obtains the bottom-up states of all nodes in AST i.e., $\{h_{i\downarrow} | \forall v \in \mathcal{V}\}$. Tree-Transformer then performs the top-down propagation, which is shown in Figure 2 (c). When the top-down propagation is finished, each node can obtain the contextual information from all other nodes, thus enables to capture the global dependency which is missed in most existing tree/graph-structured neural networks.

The top-down Tree-Transformer unit uses the state $h_{i\downarrow}$ of a single node $i$ to simultaneously update its children’s bottom-up states $H_{c_i\downarrow} = \{h_{i_1\downarrow}, h_{i_2\downarrow}, ..., h_{i_n\downarrow}\}$ into top-down states $H_{c_i\downarrow} = \{h_{i_1\downarrow}, h_{i_2\downarrow}, ..., h_{i_n\downarrow}\}$. If $i$ is the root node, $h_{\text{root}\downarrow} = h_{\text{root}\uparrow}$. In the top-down parental attention, we aim to use a top-down parental attention to pass information from parent to children. In contrast to the bottom-up parental attention, in the top-down attention, children states $H_{c_i\uparrow}$ are used as query, and their parent top-down state $h_{i\downarrow}$ as key/value. This is not a common case for multi-head attention, because when the length of key/value is 1, the softmax function over keys/values is meaningless.
So we make a slight change and simplification to the top-down parental “attention” function. Instead of computing attention scores, we directly add the top-down states of the parent node to all its children (this acts similar to a residual connection, the attention is omitted). The calculation process of the top-down unit is demonstrated below:

\[ A_{\downarrow} = \text{LayerNorm}(1 \cdot h_{\downarrow} + H_{c_{\uparrow}}) \]  

\[ H_{c_{\downarrow}} = \text{LayerNorm}(\text{FFN}_{\downarrow}(H'_{c_{\downarrow}}) + H'_{c_{\downarrow}}) \]

After top-down propagation, Tree-Transformer obtains the top-down node states \( \{h_{v_{\downarrow}} \mid v \in V\} \) which are used as the final node representations.

By combining bottom-up and top-down propagation, Tree-Transformer is capable for modeling dependencies between any node pairs along paths with arbitrary lengths. On the contrary, although traditional Transformers can capture global dependencies, they can only model paths with a maximum length (the number of Transformer layers).

Another advantage of Tree-Transformer over traditional Transformers is its memory efficiency. With recursive structure, Tree-Transformer only requires two groups of transformer parameters (one for bottom-up unit and one for top-down unit), which are far fewer than sequential Transformers (in most of these models parameters are not shared across different layers). For example, a BERT [Devlin et al., 2019] encoder has around 85M parameters (without embeddings), while a Tree-Transformer with the same embedding and feed-forward size has only 12M parameters.

\[ h_T = \sum_{v \in T} \text{softmax}(W_{\text{gate}} h_{v_{\downarrow}}) \odot h_{v_{\downarrow}} \]  

\( W_{\text{gate}} \) is a weight of \( \mathbb{R}^d \), \( \odot \) is the element-wise multiplication and \( h_T \) can be utilized as the tree-level prediction.

3.3 Calculate Tree Representation

Different from previous (recursive) tree-structured neural networks, which use the state of root nodes as representation vector for trees, we use a pooling function over the top-down states for all nodes \( \{v \mid v \in T\} \) to obtain the final representation of a tree \( T \). We adopt the global attention pooling function proposed in [Li et al., 2016]:

3.4 Limits of Tree-Transformer

The core of our Tree-Transformer is the parental attention between the parent node and children nodes, which allows the information to propagate bi-directionally between nodes with different hierarchies. This means that the parent nodes (non-leaf nodes) must contain rich semantic information. So apart from program syntax trees, our model can be extended to natural language dependency trees where both leaf and non-leaf nodes are words, but it is not suitable for constituency trees since their non-leaf nodes do not contain specific values.

4 Evaluation

In this section, we first illustrate the selected tasks and baselines for evaluation, then present the configuration of our experiments and analyse the experimental results.

4.1 Tasks and Datasets

We select two different tasks to evaluate the learning capacity of Tree-Transformer on learning tree-level and node-level representations. The detailed statistics of all datasets are listed in Table 1.

Program Classification. It requires a model to classify program ASTs based on the functionalities they implemented. We select this task to measure the ability of Tree-Transformer on learning tree-level representations. Specifically, we use two different datasets for evaluation. The first dataset is
Table 1: Basic statistics of the datasets we use in this paper. For CodeNet datasets, their vocabulary size is the size of their token vocabulary plus type vocabulary.

|                  | POJ   | Java250 | Python800 | C++1000 | C++1400 | Wrong Operator |
|------------------|-------|---------|-----------|---------|---------|----------------|
| Train samples    | 36,400| 45,000  | 144,000   | 300,000 | 252,000 | 155,628        |
| Validation samples| 5,200 | 15,000  | 48,000    | 100,000 | 84,000  | 16,868         |
| Test samples     | 10,400| 15,000  | 48,000    | 100,000 | 84,000  | 86,231         |
| Avg. nodes       | 189.58| 339.33  | 232.27    | 376.90  | 472.04  | 222.39         |
| Avg. children per node | 1.90  | 3.05    | 2.77      | 3.09    | 3.12    | 2.84           |
| Avg. depth       | 13.32 | 17.26   | 14.48     | 15.46   | 16.43   | 13.28          |
| Vocabulary       | 44    | 222     | 161       | 346     | 346     | 286,456        |

POJ algorithm classification dataset [Mou et al., 2016], which has been widely adopted to evaluate the capability of program representation models. POJ dataset contains 104 classes of C programs from student programming platforms. In our experiment, we follow the AST parsing process of [Mou et al., 2016], using pycparser\(^1\) to parse the functions to obtain ASTs and discard identifier names. The second one is the CodeNet dataset [Puri et al., 2021] which contains over 14M code sample from two open judge platforms. Here we use the code classification benchmarks, which include 4 classification datasets in three programming languages: Java250, Python800, C++1000, and C++1400. Since CodeNet has already provided simplified parse trees (SPT) for those benchmarks, we directly use them for evaluation.

Wrong Operator Localization and Repair. In order to evaluate Tree-Transformer on node-level prediction, we propose a novel tree-based wrong operator localization/repair task. Given the syntax tree of a code snippet with an erroneous binary operator (e.g., changing “+” into “-”), this task requires a model to locate the position of the misused operator node among all binary operator nodes, and predict the correct operator for this position. We synthesize a new dataset from the wrong operator detection dataset, released by CuBERT [Kanade et al., 2020]. The original dataset is built for the binary classification between correct and buggy code snippets. To enable the localization and repair task, we only keep code snippets with more than one binary operators. For this dataset, we use tree-sitter\(^2\) to parse source code into concrete syntax trees (CST). On average, each CST in our dataset contains 5.98 binary operator nodes.

4.2 Compared Baselines

We compare our approach against existing tree-structured and graph-structured neural networks. We further compare with the transformer-based model used in the program scenario. To sum up, we choose the following models as our baselines.

- **Tree-structured neural networks.** We compare Tree-Transformer with Tree-LSTM [Tai et al., 2015], TBCNN [Mou et al., 2016] and TreeCaps [Bui et al., 2021]. Tree-LSTM is a popular recursive-structured neural network and has been used in various program-related tasks. As program syntax trees can have arbitrary branching factors, we use Child-Sum Tree-LSTM as our baseline. Both TBCNN and TreeCaps are originally proposed to learn representations for program ASTs. Currently, TreeCaps is the state-of-the-art model on POJ program classification. Among the two routing algorithms of TreeCaps, we use the variable-to-static (VTS) routing in our experiments.

- **Graph neural networks.** We choose graph convolutional network (GCN) [Kipf and Welling, 2017], graph isomorphism network (GIN) [Xu et al., 2019], and gated graph neural network (GGNN) [Li et al., 2016] as our baselines. These models have been widely used in program representation learning [Allamanis et al., 2018] [Puri et al., 2021]. For the GNN baselines, their input is the original tree with bidirectional edges.

- **Transformer-based models.** Our Transformer baselines include [Shiv and Quirk, 2019], which proposed an absolute position encoding on N-ary trees. We convert the input trees to

\(^{1}\)https://github.com/eliben/pycparser
\(^{2}\)https://tree-sitter.github.io/tree-sitter/
10-ary trees for program classification and 15-ary for wrong operator localization and repair. We also compare our model with GREAT [Hellendoorn et al., 2020], which models the edges in program graphs with relative position encodings. As GREAT usually takes graphs with multiple edge types as input, its input in our paper is a program graph built on AST with additional NextToken edges (connect a terminal token node to the next terminal) [Allamanis et al., 2018].

4.3 Experimental Settings

4.3.1 General Settings

We set the Tree-Transformer embedding dimensional size to 128 for POJ and 256 for other datasets respectively. The number of attention heads is set to 4. The dropout is set to 0.2 for all experiments. We train our models with an Adam optimizer with a default learning rate of 0.002 and a warm-up phase of 2,000 steps. We implement Tree-Transformer with DGL[3] to enable efficient batching. All experiments are run on a NVIDIA RTX 8000 GPU.

4.3.2 Settings for Program Classification

For the POJ dataset, we follow previous works [Bui et al., 2021] and split the dataset into train/validation/test sets by the ratio of 7:1:2. For CodeNet, the train/validation/test ratio is 3:1:1. Since each node in CodeNet SPTs contains two parts of information: parsing rules and tokens, thus we concatenate the token embeddings and the parsing rule embeddings as the initial node embeddings. For GNN baselines, we adopt the same pooling function as Tree-Transformer. For Tree-LSTM, we employ two different approaches to acquire the representation vectors for trees: using the root node’s hidden state or using the same attention pooling as our model. For TreeCaps, we follow its original setting and use its “code capsule” to compute the probabilities of output classes.

4.3.3 Settings for Wrong Operator Localization and Repair

Locating the wrong operator node is achieved by learning a pointer pointing to a single node in a tree, which is the same as previous works on localization and repair tasks [Vasic et al., 2019; Hellendoorn et al., 2020]. Unlike [Vasic et al., 2019] which also uses a pointer for the repair task, We treat this step as a node classification task: a classifier predicts the label of the repair operator within the set of all binary operators. In our dataset, the operator set $OPs = \{-, +, *, \%, >, >=, or, <, /, and, >=, <=, !=, in, is, is not, not in\}$. We sum up the localization loss and the repair loss as the training loss for this task.

In this task, all the wrong operator nodes are located on the leaf nodes in CSTs. However, the tree-structured neural networks, e.g., Tree-LSTM, follow a bottom-up manner to propagate information, which means that the leaf nodes cannot receive the information from other nodes and cannot learn well-contextualized node representations. Thus we directly omit these tree-structured baselines and only utilize graph-based models for comparison.

4.4 Experimental Results

4.4.1 Results for Program Classification

Table[2] shows the classification results on CodeNet and POJ datasets. We can see that on all five datasets, Tree-Transformer outperforms the baseline tree-structured baselines and graph-structured baselines by a significant margin. This highlights the effectiveness of modeling global dependencies along trees with multi-head attention. When using the attention pooling same as Tree-Transformer, Tree-LSTM do not show significant improvements, suggesting that our improvements over Tree-LSTM mainly come from our model design, rather than the pooling strategy. Moreover, Tree-Transformer outperforms the recent GREAT baseline, which is integrated with additional NextToken information.

[3]https://www.dgl.ai
From the results, we notice that Tree-LSTM outperforms the GNN baselines when the given inputs are trees. Although the research interest of tree-structured neural networks is undermined by the advance of GNNs, tree-structured models are still competitive and should not be ignored.

We also compare with the large-scale pre-trained model C-BERT[4][Buratti et al., 2020], and we can find that Tree-Transformer is also competitive. The average accuracy of C-BERT is lower than ours i.e., 94.48 VS 96.16, although for some programming language, for example on Java250 and Python800, it has a higher performance than ours. Compared with C-BERT, which requires massive data for pre-training, Tree-Transformer is much more light-weight, and this further confirms the effectiveness of our model.

Table 2: Program classification accuracy(%) on CodeNet and POJ datasets, where * marks the results are reported by the corresponding paper, the others are reproduced by us.

|            | Java250 | Python800 | C++1000 | C++1400 | Overall | POJ  |
|------------|---------|-----------|---------|---------|---------|------|
| GCN        | 89.06   | 91.81     | 93.54   | 92.89   | 91.83   | 93.93|
| GIN        | 90.76   | 93.17     | 95.54   | 94.50   | 93.49   | 95.76|
| GGNN       | 88.46   | 89.92     | 89.75   | 88.01   | 89.04   | 93.22|
| Tree-LSTM (root) | 93.19 | 93.95     | 95.79   | 95.20   | 94.53   | 94.70|
| Tree-LSTM (attention) | 93.71 | 93.83     | 95.79   | 95.24   | 94.64   | 94.95|
| TBCNN      | 90.32   | 91.10     | 93.17   | 93.03   | 91.91   | 94.15|
| TreeCaps   | 91.42   | 90.26     | 93.55   | 93.24   | 92.12   | 95.88*|
| Tree-PE [Shiv and Quirk, 2019] | 91.65 | 91.11     | 92.30   | 88.97   | 91.01   | 94.19|
| GREAT      | 93.15   | 93.30     | 93.46   | 93.72   | 93.41   | 92.64|
| C-BERT [Puri et al., 2021] | 97.60* | 97.30*    | 93.00*  | 90.00*  | 94.48*  | N/A  |
| Tree-Transformer | 95.34 | 95.32     | 97.09   | 96.89   | 96.16   | 96.12|

|            | Localization Accuracy(%) | Loc & Rep Accuracy (%) |
|------------|--------------------------|------------------------|
| GCN        | 84.97                    | 60.44                  |
| GIN        | 85.69                    | 61.61                  |
| GGNN       | 84.77                    | 58.71                  |
| Tree-PE [Shiv and Quirk, 2019] | 78.65 | 53.99              |
| GREAT      | 85.43                    | 59.32                  |
| Tree-Transformer | **88.26** | **68.58** |

4The results of C-BERT are from the CodeNet paper [Puri et al., 2021].

4.4.2 Results for Wrong Operator Localization and Repair

Table 3 demonstrates the results of wrong operator localization and repair. We report two accuracy metrics: localization accuracy and joint accuracy of localization and repair. We can find that compared with graph-structured baselines, Tree-Transformer achieves a significantly better performance, especially on the joint loc&rep accuracy. Our model gains an improvement of 7% on joint accuracy compared with the best-performing baseline GIN. This indicates that our bi-directional propagation enables the model to learn effective node representations for node-level prediction tasks. The Transformer-based model [Shiv and Quirk, 2019] performs poorly on this task, its accuracies lower than all GNN baselines. This suggests that changing the original tree structures by converting arbitrary trees to N-ary trees is harmful for node-level prediction. In the wrong operator dataset, the branching factor of 10% syntax trees is larger than 15, so the structures of these trees are changed for this baseline.

Table 3: The accuracy of wrong operator localization and repair.

| Model                  | Localization Accuracy(%) | Loc & Rep Accuracy (%) |
|------------------------|--------------------------|------------------------|
| GCN                    | 84.97                    | 60.44                  |
| GIN                    | 85.69                    | 61.61                  |
| GGNN                   | 84.77                    | 58.71                  |
| Tree-PE [Shiv and Quirk, 2019] | 78.65 | 53.99              |
| GREAT                  | 85.43                    | 59.32                  |
| Tree-Transformer       | **88.26**                | **68.58**              |
4.4.3 Ablation Study

We perform an ablation study to further investigate the impact of each component in Tree-Transformer. Table 4 shows the results of Tree-Transformer when removing each component on program classification (Java250) and wrong operator localization. Noted that after removing fraternal attention, the position encoding within it is also removed accordingly. To separate position encodings from the fraternal attention, we add a new variant of Tree-Transformer where fraternal attention is removed, and position embedding vectors are added before the bottom-up parental attention.

From the results, we can see that first our top-down propagation is succinct and powerful on both tasks. Without it, the performance drops greatly, which indicates that it is effective on both node-level prediction and tree-level classification tasks. Furthermore, even with only bottom-up propagation, Tree-Transformer still outperforms tree-structured baselines, showing that our attention-based neural network unit is effective. Moreover, modeling fraternal dependencies and sibling positions also contribute to both tasks. However, the effect of position encoding in Java250 is less significant than in WrongOp: removing position encodings in Java250 hardly affects the classification accuracy. This may indicate that in our program classification datasets, the sibling order information is not essential for distinguishing programs of different classes. On the contrary, sibling orders in wrong operator localization are more important because locating wrong operators requires reasoning on relationships between operators and operands, and sibling dependency is a key part of these relationships.

An interesting finding is that only removing position encodings (keep the fraternal attention) results in worse accuracies than jointly removing position encodings and fraternal attention on the WrongOp dataset. Because the fraternal attention alone cannot learn from the sibling order information, adding this attention without given position information will deepen the model and make Tree-Transformer harder to train. If we keep the position encoding and only remove the fraternal attention, the results are still lower than the complete model. This suggests that position encoding works better when integrated with fraternal attention in Tree-Transformer.

Table 4: Ablation study on program classification (Java250) and wrong operator localization and repair. “-” means removing a certain component from Tree-Transformer. The experiment of removing top-down propagation on wrong operator localization is omitted because leaf nodes cannot receive the information from their parent nodes only through the bottom-up propagation.

| Model                          | Java250 | WrongOp |
|-------------------------------|---------|---------|
|                               | Loc     | Loc & Rep |
| Tree-Transformer              | 95.34   | 88.26   | 68.58 |
| -position encoding            | 95.33   | 85.78   | 63.32 |
| -fraternal attention          | 94.66   | 86.26   | 63.91 |
| -fraternal attention + position encoding | 94.95 | 87.18   | 65.64 |
| -top-down propagation         | 94.01   | N/A     | N/A   |

5 Conclusion and Future Work

In this paper, we propose a novel recursive-structured network Tree-Transformer for program representation learning. Tree-Transformer leverages the powerful multi-head attention in two dimensions: fraternal and parental, to capture the dependencies between siblings and ancestors/predecessors on trees. Motivated at the neglect of the global dependency among nodes in current recursive-structured neural networks or GNNs, we propose bidirectional information propagation along trees and extend existing recursive neural network architecture with a novel top-down unit. Extensive experiments on two different tasks: program classification (tree-level prediction) and wrong operator localization & repair (node-level prediction), have demonstrated the effectiveness of our proposed approach. In the future, we would like to further explore the potential of our Tree-Transformer with pre-training techniques and apply Tree-Transformer in different fields, such as natural language dependency trees.

References

Mahtab Ahmed, Muhammad Rifayat Samee, and Robert E Mercer. You only need attention to traverse trees. In Proceedings of the 57th Annual Meeting of the Association for Computational
Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 30–38, 2021.

Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 2019.

Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN’96)*, volume 1, pages 347–352. IEEE, 1996.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2020.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations*, 2016.

Zhongli Li, Qingyu Zhou, Chao Li, Ke Xu, and Yunbo Cao. Improving BERT with syntax-aware local attention. In *Findings of the Association for Computational Linguistics*, pages 645–653, 2021.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. Retrieval-augmented generation for code summarization via hybrid gnn. In *International Conference on Learning Representations*, 2021.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. Integrating tree path in transformer for code representation. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. *Advances in Neural Information Processing Systems*, 32:12081–12091, 2019.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, 2015.

Zhiyang Teng and Yue Zhang. Head-lexicalized bidirectional tree lstms. *Transactions of the Association for Computational Linguistics*, 5:163–177, 2017.
Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. Modular tree network for source code representation learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–23, 2020.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Ziyu Yao, Frank Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations*, 2021.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, pages 783–794. IEEE, 2019.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*, 2021.
Appendix

Analysis on Learned Code Representations

To analyze the program representation learned by Tree-Transformer, we make a visualization study on the Java250 code classification dataset. The visualization result is shown in Figure 3. For clarity, we choose the first ten classes from the total 250 classes in our visualization study. We can see that the embeddings from the same class are similar, which is consistent with our classification accuracy.

Figure 3: A visualization of program representations learned by Tree-Transformer on Java250 code classification.