TreeBERT: A Tree-Based Pre-Trained Model for Programming Language

Xue Jiang[1]   **Zhuoran Zheng**[2]   **Chen Lyu**[*1]   **Liang Li**[1]   **Lei Lv**[1]

[1]School of Information Science and Engineering, Shandong Normal University, China
[2]School of Computer Science and Engineering, Nanjing University of Science and Technology, China

Abstract

Source code can be parsed into the abstract syntax tree (AST) based on defined syntax rules. However, in pre-training, little work has considered the incorporation of tree structure into the learning process. In this paper, we present TreeBERT, a tree-based pre-trained model for improving programming language-oriented generation tasks. To utilize tree structure, TreeBERT represents the AST corresponding to the code as a set of composition paths and introduces node position embedding. The model is trained by tree masked language modeling (TMLM) and node order prediction (NOP) with a hybrid objective. TMLM uses a novel masking strategy designed according to the tree's characteristics to help the model understand the AST and infer the missing semantics of the AST. With NOP, TreeBERT extracts the syntactical structure by learning the order constraints of nodes in AST. We pre-trained TreeBERT on datasets covering multiple programming languages. On code summarization and code documentation tasks, TreeBERT outperforms other pre-trained models and state-of-the-art models designed for these tasks. Furthermore, TreeBERT performs well when transferred to the pre-trained unseen programming language.

1 INTRODUCTION

With the development of pre-trained models such as BERT [Devlin et al., 2019], state-of-the-art performance has been achieved on many natural language processing tasks. The use of pre-trained models reduces the budget of downstream tasks while achieving high accuracy and fast training speed. The success of pre-trained models in natural language processing (NLP) has also driven the emergence of

---

[*]Corresponding author (lvchen@sdnu.edu.cn)

pre-trained models for programming language (PL), such as CuBERT [Kanade et al., 2020] and CodeBERT [Feng et al., 2020], which learn generic code representations through pre-training tasks and then use them to enhance PL-oriented downstream tasks, such as code summarization and code classification.

However, existing PL-oriented pre-trained models face two main challenges. **1) Design of appropriate mechanism for learning program syntactical structure.** PL-oriented pre-trained models all adopt a natural language-like treatment, modeling the code as a sequence of words and considering only the linguistic nature of the code. However, the code is also strongly structured, and the semantics of the code relys on the combination of program statements and expressions with different syntactical structures to be represented. Therefore, the structural features of the program cannot be ignored. Typically, a code snippet can be parsed into a unique corresponding abstract syntax tree (AST) based on defined syntax rules. Previous works [Yin and Neubig, 2017, Rabinovich et al., 2017, Mou et al., 2016] have noted the key role of AST in program representation. However, the standard Transformer [Vaswani et al., 2017] architecture for pre-trained models cannot utilize the tree structure. How to use AST as input to the pre-trained model is a challenging problem. **2) Exploration of pre-training tasks for the tree structure.** The existing PL-oriented pre-trained models straightforwardly follow the NLP pre-training tasks. There is some inappropriate application of sequence-oriented tasks directly to non-sequential structured AST. Therefore, new pre-training tasks should be designed for tree so that the pre-trained model can extract both syntactic and semantic information from AST.

In this paper, we propose TreeBERT, *a tree-based pre-trained model for programming language.* TreeBERT follows the Transformer encoder-decoder architecture. To enable the Transformer to utilize the tree structure, we represent the AST corresponding to the code snippet as the set of root node to terminal node paths and then introduce node position embedding to obtain the position of the node

in the tree. We propose a hybrid objective applicable to AST to learn syntactic and semantic knowledge, i.e., tree masked language modeling (TMLM) and node order prediction (NOP). In TMLM, we design a novel masking strategy to mask the input of the encoder and decoder, input the set of AST paths with masked nodes at the encoder side, and predict the complete code snippet using the contextual information in AST at the decoder side. Since the order of nodes in the path expresses the program structure information, NOP improves the ability of the model to capture the syntactical structure information by predicting whether the nodes are out of order. As a result, TreeBERT can be applied to a wide range of PL-oriented generation tasks by means of fine-tuning and without extensive modifications to the task-specific architecture.

Our contributions are summarized as follows:

- We propose TreeBERT, a PL-oriented tree-based pre-trained model. TreeBERT is pre-trained on multiple programming language datasets and can be fine-tuned for improving the accuracy of PL-oriented generation tasks. Our source code and model are available at https://github.com/17385/TreeBERT.

- We represent the AST as a set of constituent paths and introduce node position embedding. We propose a hybrid objective for the tree structure, including TMLM and NOP. The former designs a novel masking strategy to help the model understand AST and infer semantics. The latter enables more program syntactical structure features to be extracted.

- We conduct extensive experiments on Python, Java, and C# datasets to verify that TreeBERT achieves state-of-the-art performance in code summarization and code documentation and generalizes better to another programming language that has not been seen in the pre-training phase.

2 MOTIVATION

In this section, we describe our motivations for proposing TreeBERT, including the program syntactical structure and the pre-training task.

Program syntactical structure CuBERT [Kanade et al., 2020] represents the first attempt to pre-training a BERT contextual embedding of source code and shows its efficacy on five classification tasks. CodeBERT [Feng et al., 2020] is the first bimodal pre-trained model capable of handling programming language (PL) and natural language (NL). It is trained using a hybrid objective function, including standard masked language modeling [Devlin et al., 2019] and replacement token detection [Clark et al., 2020], to achieve state-of-the-art performance in NL-PL comprehension tasks (e.g., code retrieval) and generation tasks (e.g., code doc-

umentation). Nevertheless, both CuBERT and CodeBERT are based purely on the sequence. The knowledge they learn from the sequence is not sufficient because they do not take into account the learning of syntactical structural information in the program. Additionally, CodeBERT points out that using the code sequence causes a partial loss of accuracy compared to using AST. Hu et al. [2018] use a linear traversal approach to represent AST as the sequence. LeClair et al. [2019] combine tokens in code with code structure in AST. Alon et al. [2019] represent a given code snippet as the set of paths between k pairs of random terminal nodes in the AST, which makes their model outperform other state-of-the-art techniques in code summarization and code captioning tasks. Each of these task-specific, non-pre-trained models demonstrates that a tree-based representation can effectively extract information from source code than can a sequential representation.

The Transformer [Vaswani et al., 2017] has demonstrated strong feature learning capability on NLP tasks, and the strategy of using large amounts of data for pre-training has been successful. Addressing how to enable Transformer to utilize tree structure is the first step in enabling pre-trained models to learn syntactical structure. Sun et al. [2020] attempt to solve this problem by adding the structural convolution sublayers to the first few Transformer decoder blocks. Instead of changing the Transformer structure, TreeBERT represents the AST using the set of paths from the root to the terminal nodes to serialize the AST. This approach represents a tree uniquely compared to other traversal methods, such as pre-order traversal. We also introduce node position embedding, which obtains hierarchy information and the relative position information of the node's parent and sibling nodes. These operations ensure that the Transformer makes maximum use of all the information in the tree.

Pre-training Task Pre-training tasks are crucial for learning generic representations of language. In NLP, masked language modeling [Devlin et al., 2019], next sentence prediction [Devlin et al., 2019], sentence order prediction [Lan et al., 2020], replaced token detection [Clark et al., 2020], permuted language modeling [Yang et al., 2019], and other tasks have been used for pre-training. These specially designed pre-training tasks enable the model to learn contextually relevant representations of each member of the input sentence using a large amount of data, thus the model to implicitly learn generic knowledge in the language.

The pre-training tasks of CuBERT and CodeBERT follow the pre-training tasks of NLP. These pre-training tasks are designed for sequence, while non-sequential structured AST has more a complex structure than sequence. It is required to design pre-training tasks for AST to learn the code representation effectively. Our work is based on Seq2Seq MLM used in MASS [Song et al., 2019] and T5 [Raffel et al., 2020] and has been shown to benefit Seq2Seq-style down-
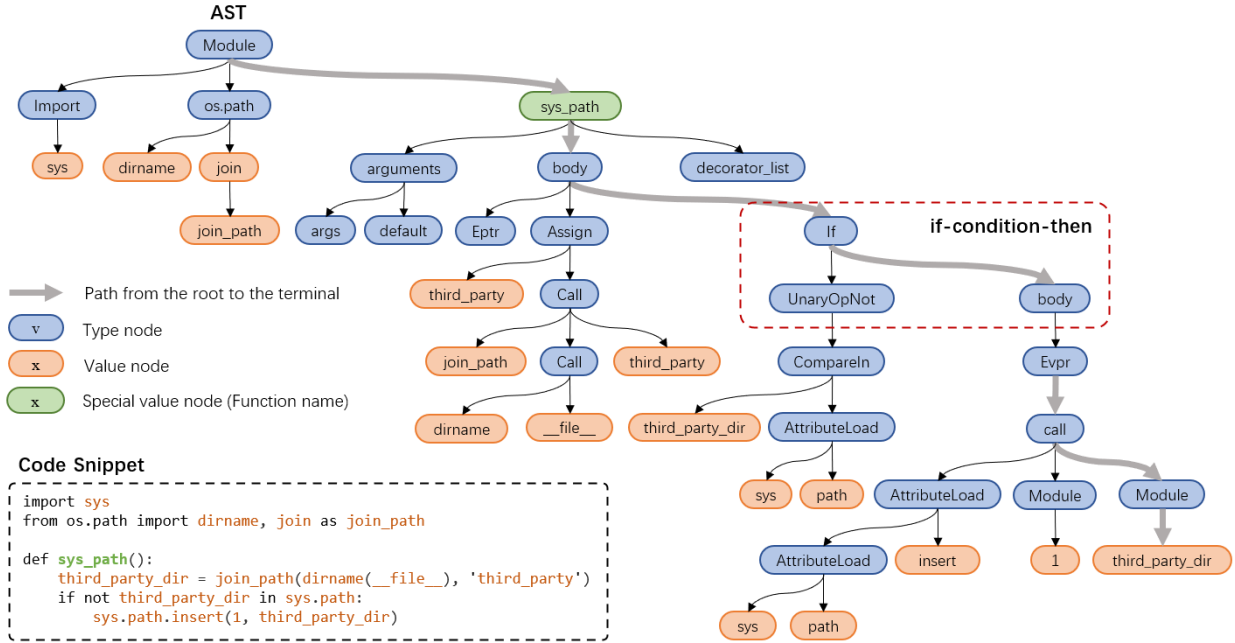
Figure 1: AST representation of the code snippet. When we represent AST, the terminal node is its corresponding value attribute, and the non-terminal node is its corresponding type attribute, except for the function name that acts as a non-terminal node but uses the value attribute.

stream tasks. Specifically, we design TMLM. Considering the repetition of nodes in AST paths and the importance of terminal nodes, we adopt a novel masking strategy to mask the nodes in AST and tokens in code snippet. The input of the encoder is the set of masked AST paths, and the output of the decoder is the predicted complete code snippet, which facilitates TreeBERT to understand AST and infer the missing semantics. In addition, to learn node order constraints in the AST, e.g., the "if" node must be followed by the "body" node, we develop NOP to capture more information about the syntactical structure of the program.

3 PROPOSED MODEL

In this section, we describe the details of TreeBERT, including the model architecture, the input representation, and the pre-training tasks employed by TreeBERT.

3.1 MODEL ARCHITECTURE

The model architecture of TreeBERT is based on the original implementation of the Transformer-based encoder-decoder described by Vaswani et al. [2017]. We omit the detailed description of the Transformer architecture since the use of the Transformer is already widespread [Dong et al., 2019, Lan et al., 2020, Liu et al., 2019]. Our model modifies the encoder side of the Transformer, adding only a fully connected layer to adjust the dimensionality of the input.

3.2 INPUT REPRESENTATION

In the pre-training phase, we set the input as a set of constituent paths in the corresponding AST given a code snippet. All node embeddings in the path are concatenated to represent the path after adding the node position embedding. The input at the decoder side is the sequence of tokens obtained from the code snippet after tokenization. In particular, the value nodes in the AST and tokens in the code snippet are represented by summing the subword embeddings.

AST Representation Each code snippet is converted into an AST, as shown in Figure 1, which exhibits the syntactical structure of the program in the form of a tree. Each node in the tree represents a structure in the code. For example, a conditional jump statement such as if-condition-then can be represented by a node with two branches.

AST nodes are divided into two categories: type nodes, such as "argument" and "Eptr", denoted as v, and value nodes (i.e., the tokens in the code), such as "sys_path" and "join_path", denoted as x. Note that the value nodes are almost always terminal nodes. We use the set of paths from the root node to the terminal nodes to represent the AST, $\mathbf{A} = \{\mathbf{p_1}, \mathbf{p_2}, \cdots, \mathbf{p_N}\}$, where N is the number of paths contained in the AST.

Code Representation The code snippet corresponding to AST is tokenizing into a sequence of tokens, and tokens $[LT]$ and $[CLS]$ are added at the beginning and end, respec-

tively, i.e., $\mathbf{C} = [\mathbf{LT}], \mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_M}, [\mathbf{CLS}]$ where $[\mathbf{LT}]$ is the representation vector of $[LT]$, $[\mathbf{CLS}]$ is the representation vector of $[CLS]$, and M is the length of the code snippet. $\mathbf{C}$ will be used as the input to the decoder, which is shown in Figure 2. $[EOS]$ is the end-of-sentence identifier at the decoder side. Not only does $[LT]$ serve as a start-of-sentence identifier, but its value represents the type of the target programming language, e.g., $[LT] = [PLT]$ when the language type is Python, $[LT] = [JLT]$ when the language type is Java and $[LT] = [UNK]$ when the language generated by the decoder is a language not seen in the pretraining phase. This is because the implementation details of the different types of languages are hidden when the code snippet is converted to AST. We need to hint at the language type for the model to learn the differences between different programming languages. $[CLS]$ is used as the aggregate representation for NOP.

Path Representation   Each path is a sequence of nodes, $p_i = v_1^i v_2^i \cdots v_{L-1}^i x_t^i$, where the terminal node x_t^i of the path is a token in the corresponding code snippet, and L is the length of the path. We concatenate the vector of nodes in the path to represent the path.

$$\mathbf{p_i} = Concat[\mathbf{v_1^i}; \mathbf{v_2^i}; \cdots ; \mathbf{v_{L-1}^i}; \mathbf{x_t^i}].$$

Note that there is no ordering relationship between path representation vectors in the set of paths. Therefore, unlike the standard Transformer, our model encoder side does not add position encoding to assign position information to the path vectors but adds the position information of the nodes in the tree using node position embedding when forming the node representations.

Token Representation   A large number of user-defined identifiers in the program often lead to out-of-vocabulary tokens or sparse vector representation. To alleviate this problem, we use bytes-pair-encoding (BPE) [Sennrich et al., 2016] to learn the most frequent subtoken from value nodes in the AST and tokens in the code snippet and split them, e.g., "third_party" might be split into "third", "_", "party". Following Alon et al. [2019], we use the vector sum of all subtokens that constitute each token to represent the complete token.

$$\mathbf{x_t} = \sum_{s \in BPE\_split(x_t)} E_s^{subtokens}.$$

where $E^{subtokens}$ is a learned embedding matrix to represent each subtoken. The number of type nodes in AST is fixed and small, so it is represented as a real-valued vector by embedding.

To train a more general pre-trained model, we share vocabulary and weights for Python and Java. This approach can reduce the overall vocabulary size and maximize the token overlap between languages, thereby improving the cross-language performance of the model [Conneau and Lample, 2019, Devlin et al., 2019, Lample et al., 2018].

Node Position Embedding   To obtain the relative position information of the nodes in the AST, we add node position embedding to the node embedding on the encoder side.

A node's position embedding is a linear combination of its parent node position embedding and its corresponding level embedding. There are $H+1$ level embeddings as parameters, namely, W^{level}, where H is the height of the tree. We use W_0^{level} as the parent position embedding of the root node. If there is a node on level j whose position embedding is W^{parent} and has c child nodes, then the position embedding of its i-th child node is

$$E_i^{position} = \frac{c-i+1}{c+1} W^{parent} + \frac{i}{c+1} W_{j+1}^{level}, c \geq 1 \ and \ 1 \leq i \leq c.$$

where W^{parent}, W^{level} are learnable weight matrices. Node position embedding can obtain hierarchy information and the relative position information of the node's parent and sibling nodes.

The pros of node position embedding are fewer parameters (approximately 1/200 of learned position embedding [Gehring et al., 2017] in the experiment) and extrapolability. If n parameters are used, learned position embedding can encode n nodes, while node position embedding is all nodes in a tree of height n-1, the number much larger than n. Learned position embedding, which allocates a different weight matrix as parameters for each position, fails to scale up. Compared to learned position embedding, node position embedding allows the model to extrapolate to a larger number of nodes than the ones encountered during training.

3.3   PRE-TRAINING TASKS

We design two tasks, TMLM and NOP, for training Tree-BERT. Seq2Seq MLM has been shown to be effective in previous studies [Song et al., 2019, Raffel et al., 2020]. We design the TMLM by extending the Seq2Seq MLM to the AST. NOP considers the node order constraints in AST to further extract program structure information. The two pre-training tasks are illustrated in Figure 2.

3.3.1   Tree Masked Language Modeling (TMLM)

Given an AST-code snippet pair (A, C), we propose a strategy for masking the nodes in the AST and the tokens in the code snippet.

At the encoder side, we first sample the nodes in path p_i according to the distribution of probability $\{q_n^i\}_{n=1 \cdots L}$ and use the $TOPK()$ operation to select the top k nodes m_i^A with the higher probability and then replace these nodes in
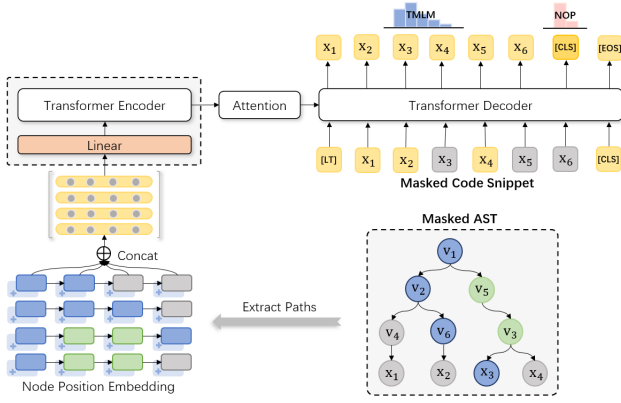
Figure 2: **Overview of TreeBERT.** The gray nodes indicate that the nodes (or tokens) are masked, and green nodes mean that the nodes (e.g., v_3 and v_5) exchange their positions.

path p_i with a special token $[mask]$ to obtain p_i^{masked}.

$$q_n^i = \frac{e^{(l-L)}}{\sum_{j=1}^{L} e^{(j-L)}},$$
$$m_i^A = TOPK\left(p_i, k, \{q_n^i\}\right),$$
$$p_i^{masked} = REPLACE\left(p_i, m_i^A, [mask]\right),$$
$$A^{masked} = \{p_1^{masked}, p_2^{masked}, \cdots, p_N^{masked}\}.$$

where A^{masked} denotes the set of masked paths, l is the current node level, L is the maximum node level in the path, N is the number of paths contained in an AST, and $i = 1 \cdots N$. Note that L is subtracted to prevent numerical overflow. This ensures that nodes with larger levels in the path have a higher probability of being masked.

TMLM masks the nodes closer to the terminal in the path with higher probability. The main reasons are 1) since each path is from the root node to the terminal node, the closer the node is to the root node, the more times it is repeated in the set of paths. If we use the standard MLM masking strategy, many same type nodes are masked. Repeatedly learning the representation of such nodes will harm the performance of the model. 2) The terminal nodes of the AST usually refer to user-defined values that represent identifiers and names in the code with rich meanings. Therefore, masking such nodes more frequently can force our model to learn their representations.

At the decoder side, the input C^{masked} of the decoder is obtained by masking the tokens in the code snippet according to the following equations.

$$m^C = \{x | x \in C \cap x \notin m^A\},$$
$$C^{masked} = REPLACE\left(C, m^C, [mask]\right).$$

where $m^A = m_1^A \cup m_2^A \cup \cdots \cup m_N^A$ and x is the element of the set m^C that needs to be masked in the code snippet.

We keep the tokens corresponding to the value nodes in m^A and mask the other tokens in the code snippet C. In this way, by making the next token prediction, TMLM can force the decoder to rely on the feature representation of the AST instead of the previous tokens in the code snippet.

An example is shown in Figure 2. In the AST, according to the above strategy, the set of nodes to be masked for the four paths are $m_1^A = \{v_4, x_1\}, m_2^A = \{x_2\}, m_3^A = \{\}$, and $m_4^A = \{x_4\}$. In the code snippet, the masked value nodes x_1, x_2, x_4 in the path are given, and the other nodes are masked, i.e., $m^C = \{x_3, x_5\}$. The decoder needs to predict the complete code snippet $x_1, x_2, x_3, x_4, x_5, x_6$.

In TMLM, the encoder reads the set of masked AST paths, and then the decoder infers the code snippet corresponding to the AST. Notably, when the code is converted to AST, some semantic information is hidden, such as "+", ">", "<=" and other binary operators are represented in AST using the "BinOpSub" node. In this case, if the decoder is designed to predict AST, the above semantic information will be ignored. Therefore, we design the decoder to predict the code snippet to encourage the model to infer such semantic information, thus enhancing its generalization ability in downstream tasks. The objective function of TMLM is given as follows.

$$\mathcal{L}_{TMLM}(\theta) = \frac{1}{M} \sum_{x \in C} -log \prod_{t=1}^{M} P\left(x_t | x_{<t}, A^{masked}\right).$$

where M is the length of the code snippet and x is the token in the code snippet.

In summary, TMLM can force the encoder to understand the AST and infer the semantic information hidden in the AST. It also encourages the decoder to extract information from the encoder to help with code snippet prediction, enabling joint training of the encoder-decoder framework.

3.3.2 Node Order Prediction (NOP)

TMLM is capable of learning rich semantic knowledge. To further improve the ability to extract syntactical structure information from program, we designed the binarized pre-training task NOP.

There are some implicit constraints on the order of nodes in the AST. Taking the path "Module ->$\cdots$-> if -> body -> Expr ->$\cdots$-> third_party_dir" in Figure 1 as an example, there must be a "body" node below the "if" node and a "body" node must have an "Expr" node underneath it. To capture this syntactical structure information, we decide with a certain probability whether to randomly exchange the positions of some nodes in the path, and then train the model to distinguish whether the order of nodes in the AST is correct or not. As shown in Figure 2, we swap the positions of nodes v_3 and v_5. The hidden vector of $[CLS]$ will be

compressed to 1 dimension by a fully connected layer, and then the probability $\bar{y}$ of the existence of nodes out of order in the AST path is obtained via the Sigmoid function. The objective function of NOP is as follows.

$$\mathcal{L}_{NOP}\left(\theta\right) = -\left(ylog\bar{y} + \left(1 - y\right)log\left(1 - \bar{y}\right)\right).$$

where $\bar{y}$ takes a value of 1 when AST has nodes out of order and is 0 otherwise.

Since TMLM and NOP play different roles, we adopt the hyperparameter α to adjust the loss function.

$$\mathcal{L} = \min_{\theta}\left(\alpha\mathcal{L}_{TMLM}\left(\theta\right) + \left(1 - \alpha\right)\mathcal{L}_{NOP}\left(\theta\right)\right).$$

4 EXPERIMENTS

In this section, we first present the pre-training setup for TreeBERT. Then, we provide the results of fine-tuned Tree-BERT on some generation tasks. Finally, an ablation study is performed to evaluate the effectiveness of various components of TreeBERT.

4.1 PRE-TRAINING

Pre-training Data The pre-training datasets we used are the Python and Java corpus published by CuBERT. Duplicate data from the fine-tuned datasets were removed. The Python dataset consists of 7.2 million Python files containing over 2 billion tokens, which are processed into ASTs with approximately 500 million paths and 7 billion nodes. The Java dataset consists of 14.1 million Java files containing approximately 4.5 billion tokens, which are processed into ASTs containing approximately 1.1 billion paths and 16.5 billion nodes.

Training Details TreeBERT sets the number of encoder layers (i.e., Transformer blocks) to 6, the hidden size to 1024, and the number of self-attention headers to 8. The decoder uses the same settings as the encoder. For pre-training, the max path number, max node number and max code length are set to 100, 20 and 200, respectively, based on the statistical information about the dataset. We took 32K BPE merge operations. We use Adam [Kingma and Ba, 2015] with a learning rate of {1e-3, 1e-4, 1e-5}, $\beta1 = 0.9$, $\beta2 = 0.999$, $L2$ weight decay of 0.01, learning rate warm up during the first 10% of steps, and linear decay afterwards. We use a GELU activation, and the batch size is {2048, 4096, 8129}. We also applied 0.1 dropout [Srivastava et al., 2014] on all layers. In TMLM, each path randomly masks 15% of the nodes. In the NOP, each AST has a 50% chance of exchanging a pair of nodes.

We train TreeBERT with different values of weight α (0.25, 0.5, 0.75, 1) and test the performance of TreeBERT in two downstream tasks, code summarization and code documentation. The results show that TreeBERT's performance improves as the value of α increases and performs best at 0.75, but the model performance starts to deteriorate when the value of α increases further. Therefore, we set the weight α in the loss function to 0.75.

4.2 FINE-TUNING

To verify the effectiveness of TreeBERT, TreeBERT was fine-tuned for two generation tasks and was compared with the baselines. The generation tasks are code summarization and code documentation. We also evaluate the performance of TreeBERT on the C# dataset and experimentally demonstrate that TreeBERT generalizes well to programming language not seen in the pre-training phase.

4.2.1 Code Summarization

Code summarization refers to predicting the function name of a given function body, which can roughly describe the function and help programmers name the function.

Datasets and Training Details In the code summarization task, we evaluate TreeBERT on Python and Java datasets, where the Python dataset uses ETH Py150 [Raychev et al., 2016] and the Java datasets [Allamanis et al., 2016] use Java-small, Java-med and Java-large. The statistical information of the datasets is provided in the supplementary material.

We set the learning rate and batch size to 1e-5 and 64, respectively. We use the Adam update parameter, and the number of epochs is set to 100. If the model performance does not improve over several epochs, we end the training early and select the model with the lowest loss in the validation set for subsequent evaluation.

Model Comparison The selected comparison baselines are the sequence-based Transformer [Vaswani et al., 2017], graph-based Graph2Seq [Xu et al., 2018], AST-based Code2Seq [Alon et al., 2019], and both AST and sequence-based Code+Gnn+GRU [LeClair et al., 2020]. Transformer has achieved state-of-the-art performance for translation tasks. Graph2Seq introduces a novel graph-to-sequence neural encoder-decoder model that maps input graphs to vector sequences and decodes target sequences from these vectors using an attention-based LSTM. Code2Seq represents a code snippet as k paths between terminal nodes in its AST and uses attention to select the relevant paths when decoding. Code+Gnn+GRU processes AST using ConvGNN and combines the output of the ConvGNN encoder with the output of the code token encoder.

We also compared TreeBERT with other pre-trained models, namely, NL-oriented MASS ($L_e = 6, L_d = 6, H =$

Table 1: Experimental results of code summarization on datasets of different sizes in Java and Python.

Methods	ETH Py150			Java-small			Java-med			Java-large		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
Methods without pre-training												
Transformer	22.18	12.67	16.13	28.13	26.7	31.41	50.11	35.01	41.22	59.13	40.58	48.13
Graph2Seq	35.74	24.77	29.26	50.44	37.15	42.79	61.07	46.89	53.05	62.89	54.07	58.15
Code2Seq	36.45	25.59	30.07	50.64	37.40	43.02	61.24	47.07	53.23	64.03	55.02	59.18
Code+Gnn+GRU	**40.66**	**29.89**	**34.45**	**55.87**	**42.32**	**48.16**	**67.10**	**52.35**	**58.81**	**71.66**	**60.56**	**65.64**
Methods with pre-training												
MASS(Pre-training with code)	38.05	28.14	32.35	53.06	40.15	45.71	62.96	48.56	54.83	67.27	55.45	60.79
CuBERT	33.48	22.61	26.99	46.15	32.62	38.22	54.58	39.73	45.99	57.55	45.07	50.55
CodeBERT	35.97	25.12	29.58	49.39	35.19	41.10	58.41	43.16	49.64	61.76	49.18	54.76
TreeBERT	**45.81**	**34.01**	**39.04**	**60.33**	**45.68**	**51.99**	**69.95**	**54.42**	**61.22**	**73.43**	**62.03**	**67.25**
Absolute gain over Code+Gnn+GRU	+5.15	+4.12	+4.58	+4.46	+3.36	+3.83	+2.85	+2.07	+2.40	+1.77	+1.47	+1.61

1024, $A = 8$) and PL-oriented CuBERT ($L_e = 24, H = 1024, A = 16$) and CodeBERT ($L_e = 12, H = 768, A = 12$), where L_e is the number of encoder layers, L_d is the number of decoder layers, H is the hidden size, and A is the number of self-attention heads. We retrained MASS using the same parameter settings and pre-training data as those used for TreeBERT. In addition, we used the parameters of CuBERT and CodeBERT to initialize Transformer's encoder. The other baseline models were set according to the parameters given in their papers. In this task, we evaluate the performance of each model using three metrics: precision, recall, and F1 score, which refer to supplementary material for the calculation methods.

Table 1 lists the results of the code summarization task, and visualizations of the data are shown in the supplementary material. TreeBERT significantly outperformed other models, including both with pre-trained and without pre-trained on all four datasets. Graph2Seq, Code2Seq, and Code+Gnn+GRU outperform Transformer, and these results suggest that tree-based or graph-based representations are more effective than sequence in extracting information from code. MASS, after retraining using the code dataset, outperforms the pre-trained models CuBERT and CodeBERT, suggesting that joint encoder-decoder training improves the generative ability. However, MASS does not perform as well as Code+Gnn+GRU, which was specifically designed for this task. TreeBERT makes more use of the syntactical structure of the code than MASS, and it outperforms Code+Gnn+GRU.

When compared to the best baseline, Code+Gnn+GRU, in terms of the F1 score, TreeBERT performs better on small datasets, where it achieves a relative improvement of 7.95% on the Java-small dataset. On the smaller ETH Py150 dataset, TreeBERT achieves a relative improvement of 13.29%. The relative difference becomes smaller as the dataset becomes larger, but it still achieves a relative improvement of 2.45% on the Java-large dataset. These results indicate that TreeBERT performs better in the absence of annotated data, and it is more robust as the data size increases.

4.2.2 Code Documentation

Code documentation refers to generating a comment for the code to understand what the code does by reading the comment rather than the code itself.

Datasets and Training Details We use the Java dataset provided by DeepCom [Hu et al., 2018] to fine-tune Tree-BERT. The size of the original dataset is reduced due to the presence of some code that cannot be converted to AST. The statistics of the processed dataset are provided in the supplementary material.

In this experiment, we set the learning rate and batch size to 3e-5 and 64, respectively, and the Adam optimizer and early stopping are adopted. Finally, we select the best-performing model on the validation set.

Table 2: Evaluation results of code documentation on Java dataset using BLEU.

Methods	BLEU
Methods without pre-training	
Transformer	13.32
DeepCom	14.81
Graph2Seq	18.39
Code2Seq	18.48
Code+Gnn+GRU	**19.73**
Methods with pre-training	
MASS(Pre-training with code)	18.92
CuBERT	17.41
CodeBERT	17.87
TreeBERT	**20.49**

Model Comparison In the code documentation task, we use the same baseline as for code summarization but add a comparison with DeepCom, which proposes an AST traversal method that converts the AST into the sequence to train a seq2seq model for the code documentation task. As with DeepCom, we adopt BLEU as the metric in this experi-

ment, and its calculation is described in the supplementary material.

Table 2 shows the results of the code documentation task. TreeBERT achieves a BLEU score of 20.49, which is 7.17 points higher than that of Transformer. In this task, Transformer produces a very poor result because it stays at the shallow textual token-level compared to other methods without pre-training, thereby overlooking information that is important to the program. Transformer is trained from scratch and does not perform as well as methods with pre-training, which proves that pre-trained models learn valuable information. The performance of TreeBERT is significantly improved compared to other baselines because of the joint training of the encoder-decoder using AST.

4.2.3 Generalization to Programming Language not Seen during Pre-training

We now evaluate the performance of TreeBERT on programming language not seen in the pre-training phase. We use CodeNN's [Iyer et al., 2016] C# dataset which contains programming questions and answers about the C# language obtained from StackOverflow. The dataset statistics are presented in the supplementary material. In this task, we compare TreeBERT with CodeNN, MASS, CuBERT, and CodeBERT. In addition, we evaluate the model using the BLEU metric and the same hyperparameters as those used in code documentation.

Table 3: Performance of TreeBERT and Baseline models on language not seen in the pre-training phase.

Methods	BLEU
TreeBERT	**17.94**
CodeNN	14.18
MASS(Pre-training with code)	**16.84**
CuBERT	14.95
CodeBERT	15.31

Table 3 shows the performance of TreeBERT on C# dataset. The experimental results indicate that TreeBERT performs generalization surprisingly well on language not seen in the pre-training phase. Specifically, TreeBERT achieves a BLEU of 17.94 by fine-tuning, which is 3.76 points better than that of CodeNN, whose authors introduced the dataset. We suppose that TreeBERT can be well generalized to the new language because of the high similarity between the AST of the new language and the AST used for pre-training.

4.3 ABLATION STUDY

Extensive ablation studies were conducted to better understand the role of the different components and masking strategy in the model. We vary the model in different ways

and measure the performance changes for each variant. The experiment is performed on a Java dataset of code documentation task in the following scenarios.

- *No TMLM* — We do not use TMLM.
- *No NOP* — We do not use NOP.
- *No Node Position Embedding* — We use learned position embedding instead of node position embedding.
- *Randomly Masking Nodes* — In TMLM, we do not use our proposed masking strategy and randomly mask nodes in the AST according to the standard MLM.
- *Only Masking Value Nodes* — Since value nodes express rich semantics, we try not to mask type nodes and mask only value nodes.
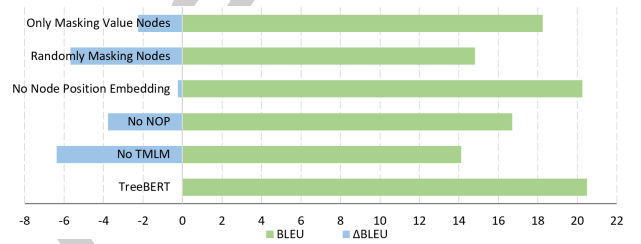

Figure 3: **Results of ablation study.** The left side shows the loss in BLEU values, and the right side shows the total BLEU values.

The detailed results of the ablation study are presented in the supplementary material. The experimental results in Figure 3 show that the model lost the most BLEU when the TMLM is missing, which can prove its important role in TreeBERT. Although the loss is smaller when NOP is missing, the syntactical structure information captured by the NOP is indispensable. When we use node position embedding instead of learning position embedding, the results indicate that we achieve comparable or even higher results than learning position embedding. It is worth mentioning the great role of the masking strategy in TMLM. If the random masking strategy is used, the model will perform poorly, and the result is almost similar to that without TMLM because most of the nodes masked are repeated AST non-terminal type nodes, resulting in a large number of code tokens not being involved in the pre-training, which is detrimental to the extraction of key semantic information in the code. Value nodes represent rich semantics, but some loss can be caused if all masked nodes are value nodes. This suggests that masking a small number of type nodes is useful, and we suppose that this approach can be used to extract structural information and help improve the performance of the model.

5 CONCLUSION

In this paper, we propose a PL-oriented tree-based pre-trained model called TreeBERT. To learn code represen-

tation using AST, TreeBERT uses the set of constituent paths of AST as input, introduces node position embedding and then trains the model using a hybrid objective, including TMLM and NOP. The experimental results indicate that TreeBERT achieves state-of-the-art performance in code summarization and code documentation tasks and performs well when transferred to the pre-training unseen programming language. To further investigate the role of each part of TreeBERT, we conducted an extensive ablation study.

TreeBERT still has considerable room for improvement. First, TreeBERT can be used not only for tasks such as code summarization and code documentation, and it can be used in any task where the source language can be constructed as an AST. We will continue to explore the possibility of applying TreeBERT to more PL tasks. Second, we will further improve TreeBERT, for example, by adding more program information to the AST or by using multimodal forms such as AST, graph and sequence simultaneously, thus extracting information about the program from different perspectives so that TreeBERT can better solve PL downstream tasks.

Acknowledgements

References

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*, 2016.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR*, 2019.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: pre-training text encoders as discriminators rather than generators. In *8th International Conference on Learning Representations, ICLR*, 2020.

Alexis Conneau and Guillaume Lample. Cross-lingual language model pretraining. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, 2019.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP*, 2020.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML*, 2017.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC*, 2018.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL*, 2016.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. 2020. URL http://arxiv.org/abs/2001.00059.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR*, 2015.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. In *6th International Conference on Learning Representations, ICLR*, 2018.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *8th International Conference on Learning Representations, ICLR*, 2020.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE*, 2019.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph

neural network. In *ICPC '20: 28th International Conference on Program Comprehension*, 2020.

Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL*, 2019.

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

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL*, 2017.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

Veselin Raychev, Pavol Bielik, and Martin T. Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2016.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL*, 2016.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. MASS: masked sequence to sequence pre-training for language generation. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, 2019.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15:1929–1958, 2014.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2017.

Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. 2018. URL http://arxiv.org/abs/1804.00823.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL*, 2017.