GraphPLBART: Code Summarization Based on Graph Embedding and Pre-Trained Model

Jie Li
School of Computer Science and Technology
Soochow University
Jiangsu, China
2027407061@stu.suda.edu.cn

Lixuan Li
School of Computer Science and Technology
Soochow University
Jiangsu, China
li_lixuan@outlook.com

Hao Zhu
School of Computer Science and Technology
Soochow University
Jiangsu, China
2027406032@stu.suda.edu.cn

Xiaofang Zhang✉
School of Computer Science and Technology
Soochow University
Jiangsu, China
xfzhang@suda.edu.cn

***Abstract*—Code summarization is a task that aims at automatically producing descriptions of source code. Recently many deep-learning-based approaches have been proposed to generate accurate code summaries, among which pre-trained models for programming languages have achieved promising results. It is well-known that source code written in programming languages is highly structured and unambiguous. Though previous work pre-trained the model with well-design tasks to learn universal representation from a large scale of data, they haven't considered structure information during the fine-tuning stage. To make full use of both the pre-trained programming language model and the structure information of source code, we utilize Flow-Augmented Abstract Syntax Tree (FA-AST) of source code for structure information and propose GraphPLBART – Graph-augmented Programming Language and Bi-directional Auto-Regressive Transformer, which can effectively introduce structure information to a well pre-trained model through a cross attention layer. Experimental results show that our approach outperforms the baseline models in some metrics.**

***Index Terms*—code summarization, pre-trained model, code structure, deep learning**

I. Introduction

Code summary refers to a natural language description of a code segment which can facilitate code comprehension. However, writing high-quality code summaries is a very time-consuming activity, and due to the rapid update of software, many human-written code summaries can be outdated fast. Thus with the development of software engineering, the code summarization task, which aims at automatically generating natural language descriptions for code segments, has received increasing interest in recent years.

Many approaches have been proposed to generate code summaries automatically, among which the deep-learning-based approaches have been proven to be the most effective. For example, Iyer et al. [1] first utilized Long Short Term Memory (LSTM) networks with attention mechanism to produce comments for C# and SQL, which outperformed the traditional retrieval-based method. Ahmad et al. also used a transformer model [2] with copy mechanism [3] to capture the long-range dependencies in source code [4], which proved the transformer model better on the task.

However, in contrast to natural language, source codes are unambiguous and structured [5]. From one aspect, programming languages are formal languages so the structure information is essentially important for deep learning models to learn the representation of source code better. To take advantage of rich and unambiguous structure information of source code, Hu et al. [5] proposed a special structure-based traversal technique to linearize the Abstract Syntax Tree (AST), the linearized ASTs are then fed into an LSTM model to generate summaries. From another aspect, though Transformer has been proven better on code summarization task, it needs more training data as its multi-head attention is purely data-driven [6]. SG-Trans [7] and SiT [8] utilize structure information such as AST and Data Flow Diagram(DFD) as inductive bias to overcome attention collapse or attention redundancy in the Transformer model, which is the main issue that hinders the transformer model's representation ability [9]. Another solution to the attention collapse problem is pre-training. For example, CodeBERT is a Roberta-based pre-trained model for programming languages and natural language (PL-NL) [10]. The model is pre-trained with the task of masked language modeling [11] and replaced token detection [12] on a large scale of training data. Pre-trained PL-NL model can be easily fine-tuned on a limited dataset for downstream tasks because the pre-training procedure enables the model to capture important dependencies from a large scale of data and thus overcome the attention collapse problem.

Though pre-trained models learn representations from a large scale of training data and have achieved great success on code summarization tasks, they haven't considered structure information during the fine-tuning stage. To address this limitation and take full advantage of both code structure information and pre-trained models, in this paper, we propose

DOI reference number: 10.18293/SEKE2023-192

GraphPLBART: a graph-augmented code summarization approach based on the PLBART pre-trained model [13]. We first parse source codes into ASTs and then construct Flow-Augmented Abstract Syntax Trees (FA-ASTs) [14] by adding additional semantic edges to each AST. Then we use Gated Graph Neural Network (GG-NN) [15] to learn the structure information of source codes. The structure representations are fused into the model with an additional cross-attention layer added after every self-attention layer of the PLBART encoder. The main contributions of this article are as follows:

- A novel graph-augmented code summarization approach is proposed to explore the integration of structure information of source code and PL-NL pre-trained model during the fine-tuning stage.
- Extensive experiments are conducted to prove the effectiveness of introducing structure information during the fine-tuning stage. The experiment results show that our GraphPLBART outperforms the baseline models.

The rest of this paper is organized as follows. Section II introduces the background and our motivation. Our approach is introduced in Section III. The experimental setup and results are presented in Section IV and V respectively. Section VI is the conclusion and the future work.

II. Background

A. Pre-trained Programming Language Model

Pre-trained programming language models, which are trained on large scale of data in a self-supervised manner to learn universal programming language representations, can avoid training a new model from scratch and have shown advantages on various downstream tasks. For example, Code T5 [16] makes use of both PL-only and NL-PL bimodal data to pre-train the model, which yields better results on five downstream tasks. Other pre-trained programming language models e.g., CodeBERT [10] and GraphCodeBERT [17] also demonstrate their promise on program understanding and generation. Besides, PLBART [13] is a sequence-to-sequence model capable of performing various programming and language understanding and generation tasks. It has undergone pre-training of denoising autoencoders on a large number of Java and Python functions, as well as related natural language texts.

B. Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language [18]. Every node of the tree represents a structure of source code. Developers can get the declaration statements, assignment statements, operation statements and realize operations by analyzing the tree structures [19]. Nowadays, AST has been widely used for various program understanding and generation tasks [7] [8] since it contains unambiguous structure information of source code which is essentially important to program understanding.

C. Motivation

Pre-trained language models for programming have shown impressive results in several programming understanding tasks, including code summarization. GraphCodeBERT has implemented an edge prediction pre-training task to learn representations from Data Flow Diagrams, which contain semantic structural information about source code. Although these models learn universal representations from well-designed pre-training tasks on large-scale data, it is essential to include structural information as an inductive bias during the fine-tuning stage. SG-Trans [7] and SiT [8] have attempted to introduce structural information into their approach, but their models are trained from scratch, which can be very time-consuming due to the purely data-driven nature of multi-head attention [6].

To address these limitations, we propose GraphPLBART, a model that effectively incorporates structure information extracted from FA-AST into pre-trained PLBART. During the fine-tuning stage, we add an additional cross-attention layer to our model to incorporate the structural information. This approach allows us to leverage the benefits of pre-training while also accounting for the structural properties of the input code. By doing so, we expect to achieve better performance in programming understanding tasks such as code summarization.

III. Approach

GraphPLBART contains a graph reader, an encoder, and a decoder. The graph reader reads grammatical information from the FA-AST and passes it on to the encoder via the attention mechanism. In response, we add another cross-attention layer to the encoder after the original self-attention layer to receive grammatical information. Besides, the weights of the encoder embedding layer, as well as the decoder, are derived from the pre-trained PLBART. The framework of the model is shown in Figure 1.

A. Graph Reader

We first parse the code into an abstract syntax tree(AST) and add data flow and control flow to extend AST into FA-AST(Flow-Augmented Abstract Syntax Tree). Then GraphPLBART extracts information using GG-NN. As with the Transformer, we perform a non-linear transformation of the graph reader with a Feed Forward layer at the end of the graph reader.

1) Flow-Augmented Abstract Syntax Tree: We build the graph representation for programs as follows. To parse ASTs from Java programs, we use a python package *javalang*[1]. On the other hand, we use a built-in Python package *ast*[2] to parse from Python programs. It is important to note that although both tools can extract ASTs, they differ in naming the nodes. Therefore, we need to find the corresponding relationship between these two naming methods, such as *WhileStatement*

[1]https://github.com/c2nes/javalang
[2]https://github.com/att/ast

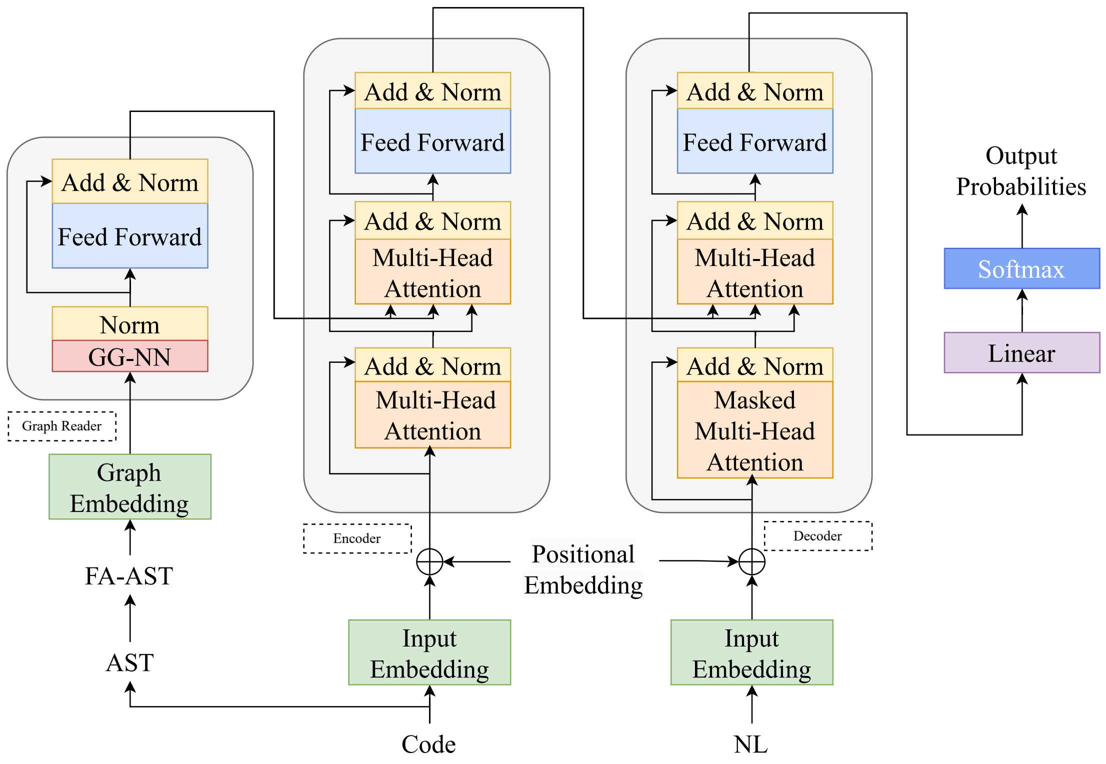

Fig. 1: Details of Model. We use a graph reader to extract grammar information from flow-augment AST, then deliver the grammatical message to the encoder by attention mechanism.

in *javalang*, which becomes *While* in *ast*, and there is no direct corresponding naming for *BlockStatement* in *ast*. During the code augmentation process, in terms of data flow, we connect non-leaf nodes with their sibling nodes, nodes with their next usage, and leaf nodes with their next nodes. In terms of control flow, we connect the body and condition of *while* and *for* statements, and for *if* statements, we connect the condition to both the true and false branches. Following [14], we add an additional backward edge for each edge that does not have one to increase the frequency of message passing. Figure 2 is an example of extracting AST from Java code and transforming it into FA-AST. The code provided is used to calculate the absolute value of an integer, including function definition, variable calculation, and *if* conditional statement. The black-directed edges in the figure are edges in the AST, and the other colored edges are edges generated during the flow augmentation process. The meaning of each edge is shown in the legend.

2) Gated Graph Neural Network: We use GG-NN only to learn embeddings of each node in FA-AST, so we do not need a readout function to read the information of the entire graph. The calculation formulas of GG-NN are as follows:

$$\begin{aligned} m_{j\to i} &= \mathrm{MLP}(h_i^{(t)}, h_j^{(t)}, e_{j,i}), \forall (j,i) \in \mathrm{E} \\ \mathbf{m}_i &= \sum_j m_{j\to i} \\ \mathbf{h}_i^{(t+1)} &= \mathrm{GRU}(\mathbf{m_i}, \mathbf{h}_i^{(t)}) \end{aligned} \tag{1}$$

where $m_{j\to i}$ represents passing message from $node_j$ to $node_i$, $e_{j,i}$ represents weight of the edge connecting $node_j$ and $node_i$, E is the set of edges, $h_i^{(t)}$ represents hidden state of $node_i$ at time t, and m_i represents message of $node_i$. MLP and GRU represents multilayer preceptron [20] and gated recurrent unit [21] respectively. To keep more of the original information in the embeddings, we only use one layer of graph network to avoid interference from the embeddings of other nodes [22]. Since there is only one layer of graph network, we do not need to apply residual connections but only normalize the embeddings.

It should be noted that since the tokenizer used by PLBART is sentencepiece [23], it may break down the original words and thereby damage the graph structure. Therefore, the graph reader uses a separate vocabulary instead of sharing the same vocabulary with the pre-trained PLBART. The embedding size of the graph reader is the same as that of PLBART.

B. Encoder

Our encoder has a structure similar to that of Transformer's encoder, except that we added another cross-attention layer after the existing self-attention layer to read grammatical information from the graph reader. This makes the structure of the GraphPLBART encoder and decoder very similar, except that the first multi-head attention layer in the encoder does not use the masking mechanism. The output of the encoder is a code embedding that has been augmented with syntactic information.

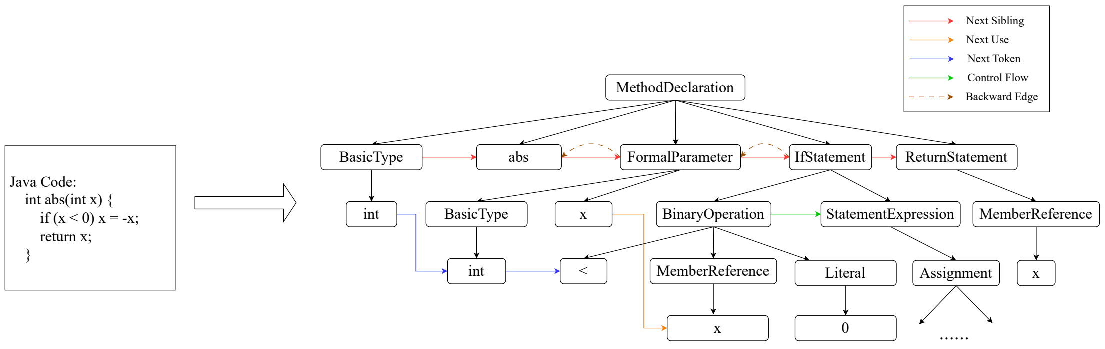

Fig. 2: Extract FA-AST from a sample Java code, only some of the added edges are shown in the graph.

C. Decoder

Our decoder is fully based on the pre-trained PLBART decoder. Moreover, because PLBART shares the same vocabulary for encoder and decoder, we do not need to use a separate copy mechanism [3] to solve the OOV (Out Of Vocabulary) problem, which simplifies the architecture and reduces computational overhead.

IV. Experimental Settings

A. Datasets and Evaluation Metrics

In our experiments, we utilized two public datasets that are commonly used. The first dataset is the Java dataset [5], which contains 87,136 Java code snippets with accompanying comments written by developers. The second dataset is the Python dataset [24], which encompasses 87,226 Python code snippets along with comments. To ensure fairness during comparison, we employed the same pre-divided datasets as [4]. We utilize three commonly used metrics to evaluate our model: BLEU [25], METEOR [26], and ROUGE-L [27].

B. Baseline

We compared the commonly used model and the proposed GraphPLBART model in this paper using data from [28].

- Hybrid2Seq [24] is a deep reinforcement learning framework that utilizes an LSTM-based encoder to learn from code snippets and binary trees generated from ASTs.
- DeepCom [5] uses an LSTM-based architecture to enhance the quality of comments by analyzing their structure and converting ASTs into token sequences.
- API+Code [29] learns API knowledge from source code API sequences and applies the learned features to enhance code summarization performance.
- Dual Model [30] uses a dual training framework to train code generation and code summarization tasks together. It employs two sequence-to-sequence networks with attention to improve the performance of both tasks.
- Transformer [4] utilizes relative positional encoding to capture pairwise relationships between tokens in the source code text.
- mAST+GCN [31] combines sequential and structural features of code using AST, graph convolution, and Transformer layers for code summarization.
- SiT [8] utilizes a structure-induced Transformer that preserves the structural relationships and self-attention mechanism for encoding.
- CodeT5 [16] is a pre-trained encoder-decoder model based on T5 [32] for programming and natural language, directly tuned with summarization datasets.
- CodeBERT [10] is a pre-trained encoder model for programming and natural language based on Roberta [33].
- M2TS [17] uses a multi-scale approach to extract features from ASTs, enabling more comprehensive extraction of structural information at local and global levels.

C. Hyper-parameter Setting

We build our model based on the Hugging Face *transformers* module[3], with the number of layers for GG-NN in the graph reader set to 4 while the other parameters of the model were the same as PLBART-base. During training, we set the batch size to 32, use the AdamW optimizer with an initial learning rate of 0.00005, and set the dropout to 0.1. While generating, we set the parameter num_beams to 6.

V. Experimental Results

This section will focus on the following questions:

RQ1: How does GraphPLBART perform compared to other baselines?

RQ2: What impact does FA-AST have on the performance of GraphPLBART?

RQ3: How does GraphPLBART perform in specific examples?

A. RQ1: How does GraphPLBART perform compared to other baselines?

We present the results of our experiments, including BLEU, METEOR, and ROUGE-L scores, in Table I, where GraphPLBART and other baselines are evaluated on the testing set. The results indicate that GraphPLBART outperforms non-pre-trained models by a significant margin. In comparison to other pre-trained methods such as CodeBERT and CodeT5, GraphPLBART demonstrates superior performance in generating code summaries. For example, on the Java dataset, GraphPLBART achieves a top BLEU score of 47.56, while

[3]https://huggingface.co/docs/transformers/index

TABLE I: Results on code summarization

Methods	Java			Python		
	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
RL+HybridSeq (2018) [24]	38.22	22.75	51.91	19.28	9.75	39.24
DeepCom (2018) [5]	39.75	23.06	52.67	20.78	9.98	37.35
API+CODE (2018) [29]	41.31	23.73	52.25	15.36	8.57	33.65
Dual (2019) [30]	42.39	25.77	53.61	21.80	11.14	39.45
Transformer (2020) [4]	44.58	26.43	54.76	32.52	19.77	46.73
mAST+GCN (2021) [31]	45.49	27.17	54.82	32.82	20.12	46.81
SiT (2021) [8]	45.19	27.52	55.87	34.31	22.09	49.71
CodeT5 (2021) [16]	46.01	28.55	56.49	34.31	22.09	49.25
CodeBERT (2020) [10]	46.64	28.84	56.23	34.34	21.99	49.73
M2TS (2022) [17]	46.84	28.93	**57.87**	-	-	-
GraphPLBART w/o graph	47.26	30.71	56.91	**34.94**	**23.29**	**50.55**
GraphPLBART	**47.56**	**30.95**	57.57	33.81	21.56	46.26

M2TS and CodeBERT achieve slightly lower scores at 46.84 and 46.64, respectively. Additionally, GraphPLBART outperforms other baseline models in the Meteor metric. However, its Rouge-L score is slightly lower than M2TS, which may be attributed to the fact that the graph network ignores the order of nodes and thus has a certain impact on the summary's order.

It is worth noting that the performance of GraphPLBART differs significantly from other baseline models on the Python dataset. This discrepancy can be attributed to the small size of the dataset, which hinders GraphPLBART's ability to fully converge on the pre-trained parameters. Nonetheless, these findings confirm the effectiveness of GraphPLBART in code summarization tasks and highlight the importance of pre-training for optimal performance.

B. RQ2: What impact does FA-AST have on the performance of GraphPLBART?

To investigate the role of graphs in code summarization, we conducted an experiment where we removed the graph reader from GraphPLBART as well as the multi-head self-attention layer that was added in the encoder. This effectively restored GraphPLBART to its original form, PLBART. The results of this experiment are shown in Table I. We found that GraphPLBART outperformed PLBART (GraphPLBART w/o graph) in all three metrics for Java datasets. This suggests that the fine-grained structure and semantic information provided by FA-AST is crucial during the inference process. However, for Python datasets, we observed that the non-graph network performed significantly better than GraphPLBART and other baselines. We suspect that this is because all parameters in PLBART without a graph were pre-trained, which greatly reduced the requirement for additional training data. On the other hand, the Python dataset was too small for the graph reader to fully converge. This finding confirms the importance of pre-training in code summarization. In summary, our experiment highlights the important role of graphs in code summarization, particularly for Java datasets.

C. RQ3: How does GraphPLBART perform in specific examples?

To further investigate the performance of GraphPLBART, we examine the summaries generated by it for some given codes, as shown in Fig 3. As we can see, the model without a graph is unable to pay attention to the *IF* control statement, resulting in the lack of premise conditions in the generated summary. However, GraphPLBART can notice it. In the Python example, *isdigit* is a method of the *str* class in Python. GraphPLBART successfully captures this syntax information and provides the *string* in the summary, while PLBART can only provide the variable name *text* while inferencing.

Java
```
public synchronized boolean removeLast ( K obj )
{
    if ( peekLast ( ) != obj ) { return _BOOL; }
    array = Arrays.copyOf ( array, array . length - _NUM );
    return _BOOL ;
}
```

Ground truth: remove the last element , if it matches .
GraphPLBART w/o graph: remove the last element at the end .
GraphPLBART: remove the last element, if it matches .

Python
```
def _number(text):
    if text.isdigit():
        return int(text)
    try:
        return float(text)
    except ValueError:
        return text
```

Ground truth: convert a string to a number .
GraphPLBART w/o graph: convert a text to a number .
GraphPLBART: convert a string to a number .

Fig. 3: The output summary of GraphPLBART in a specific input code.

VI. Conclusion and Future Work

In this paper, we introduce a novel graph augmentation approach for code summarization, which effectively leverages structural information in a pre-trained model. We conducted experiments to evaluate the effectiveness of our GraphPLBART model and found that it outperforms baseline models in certain metrics. Our work highlights the potential of pre-trained models for programming languages in code summarization tasks. In future work, we aim to explore more advanced pre-training tasks to capitalize on the large-scale training data, particularly for pre-training graph models.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (62172202), Collaborative Innovation Center of Novel Software Technology and Industrialization, the Major Program of the Natural Science Foundation of Jiangsu Higher Education Institutions of China under Grant Nos.22KJA520008, the Priority Academic Program Development of Jiangsu Higher Education Institutions, and the Undergraduate Training Program for Innovation and Entrepreneurship, Soochow University(202210285197H).

References

[1] Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L. (2016, August). Summarizing source code using a neural attention model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 2073-2083).

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

[3] See, A., Liu, P. J., Manning, C. D. (2017). Get to the point: Summarization with pointer-generator networks. arXiv preprint arXiv:1704.04368.

[4] Ahmad, W., Chakraborty, S., Ray, B., Chang, K. W. (2020, July). A Transformer-based Approach for Source Code Summarization. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (pp. 4998-5007).

[5] Hu, X., Li, G., Xia, X., Lo, D., Jin, Z. (2018, May). Deep code comment generation. In Proceedings of the 26th conference on program comprehension (pp. 200-210).

[6] Guo, Q., Qiu, X., Liu, P., Xue, X., Zhang, Z. (2020, April). Multi-scale self-attention for text classification. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 05, pp. 7847-7854).

[7] Gao, S., Gao, C., He, Y., Zeng, J., Nie, L., Xia, X., Lyu, M. (2023). Code Structure–Guided Transformer for Source Code Summarization. ACM Transactions on Software Engineering and Methodology, 32(1), 1-32.

[8] Wu, H., Zhao, H., Zhang, M. (2021, August). Code Summarization with Structure-induced Transformer. In Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021 (pp. 1078-1090).

[9] An, B., Lyu, J., Wang, Z., Li, C., Hu, C., Tan, F., ... Chen, C. (2020, November). Repulsive Attention: Rethinking Multi-head Attention as Bayesian Inference. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 236-255).

[10] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... Zhou, M. (2020, November). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Findings of the Association for Computational Linguistics: EMNLP 2020 (pp. 1536-1547).

[11] Kenton, J. D. M. W. C., Toutanova, L. K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of NAACL-HLT (pp. 4171-4186).

[12] Clark, K., Luong, M. T., Le, Q. V., Manning, C. D. (2020). Electra: Pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555.

[13] Ahmad, W., Chakraborty, S., Ray, B., Chang, K. W. (2021, June). Unified Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 2655-2668).

[14] Wang, W., Li, G., Ma, B., Xia, X., Jin, Z. (2020, February). Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 261-271). IEEE.

[15] Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R. (2015). Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

[16] Wang, Y., Wang, W., Joty, S., Hoi, S. C. (2021, November). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (pp. 8696-8708).

[17] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

[18] Fang, C., Liu, Z., Shi, Y., Huang, J., Shi, Q. (2020, July). Functional code clone detection with syntax and semantics fusion learning. In Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis (pp. 516-527).

[19] Alon, U., Zilberstein, M., Levy, O., Yahav, E. (2019). code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), 1-29.

[20] Taud, H., Mas, J. F. (2018). Multilayer perceptron (MLP). Geomatic approaches for modeling land change scenarios, 451-455.

[21] Dey, R., Salem, F. M. (2017, August). Gate-variants of gated recurrent unit (GRU) neural networks. In 2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS) (pp. 1597-1600). IEEE.

[22] Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L. (2020, April). Treegen: A tree-based transformer architecture for code generation. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 05, pp. 8984-8991).

[23] Kudo, T., Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. arXiv preprint arXiv:1808.06226.

[24] Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P. S. (2018, September). Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (pp. 397-407).

[25] Papineni, K., Roukos, S., Ward, T., Zhu, W. J. (2002, July). Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics (pp. 311-318).

[26] Banerjee, S., Lavie, A. (2005, June). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization (pp. 65-72).

[27] Lin, C. Y. (2004, July). Rouge: A package for automatic evaluation of summaries. In Text summarization branches out (pp. 74-81).

[28] Wang, Y., Dong, Y., Lu, X., Zhou, A. (2022, May). GypSum: learning hybrid representations for code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (pp. 12-23).

[29] Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z. (2018, July). Summarizing source code with transferred API knowledge. In Proceedings of the 27th International Joint Conference on Artificial Intelligence (pp. 2269-2275).

[30] Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z. (2019). Code generation as a dual task of code summarization. Advances in neural information processing systems, 32.

[31] Choi, Y., Bak, J., Na, C., Lee, J. H. (2021, August). Learning sequential and structural information for source code summarization. In Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021 (pp. 2842-2851).

[32] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

[33] Gao, Y., Lyu, C. (2022, May). M2ts: Multi-scale multi-modal approach based on transformer for source code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (pp. 24-35).