2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00125

Automatically Generating Code Comment Using Heterogeneous Graph Neural Networks

Dun Jin*, Peiyu Liu*†, Zhenfang Zhu‡
*School of Information Science and Engineering, Shandong Normal University, Jinan, China
2020317087@stu.sdnu.edu.cn, liupy@sdnu.edu.cn
‡School of Information Science and Electrical Engineering, Shandong Jiao Tong University, Jinan, China
zhuzf@sdjtu.edu.cn
†corresponding author

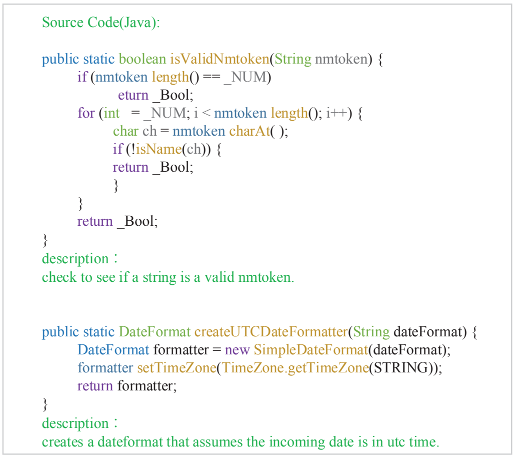
***Abstract*—Code summarization aims to generate readable summaries that describe the functionality of source code pieces. The main purpose of the code summarization is to help software developers understand the code and save their precious time. However, since programming languages are highly structured, it is challenging to generate high-quality code summaries. For this reason, this paper proposes a new approach named CCHG to automatically generate code comments. Compared to recent models that use additional information such as Abstract Syntax Trees as input, our proposed method only uses the most original code as input. We believe that programming languages are the same as natural languages. Each line of code is equivalent to a sentence, representing an independent meaning. Therefore, we split the entire code snippet into several sentence-level code. Coupled with token-level code, there are two types of code that need to be processed. So we propose heterogeneous graph networks to process the sentence-level and token-level code. Even though we do not introduce additional structural knowledge, the experimental results show that our model has a considerable performance, which indicates that our model can fully learn structural information and sequence information from code snippets.**

***Index Terms*—source code summarization, neural networks, heterogeneous graph networks, AI in SE**

I. Introduction

When we face a piece of code, it is undoubtedly an exciting thing if it is accompanied by clear comments like the code snippets in Figure 1. The advantage of accompanying code comments is that its functions can be clear to the people who use it. Especially for computer-related industry personnel, high-quality code comments can greatly save their precious time. However, writing a piece of code with high-quality comments is a costly task [1]. Therefore, only a small percentage of the code in the current code base is normatively annotated. It means that people need to read and try to understand the meaning of the code, which is obviously a time-consuming and laborious task. Therefore, automatically generating high-quality code comments is an urgent problem. Moreover, high-quality code comments can not only greatly improve the efficiency of software developers [2], but also very helpful for code retrieval and error debugging [1]. The research goal of this paper is to generate a concise natural language description for a method or function body as its comment.

As we all know, programming language is extremely strict on structure standardization [3]. Regardless of C++, Python,



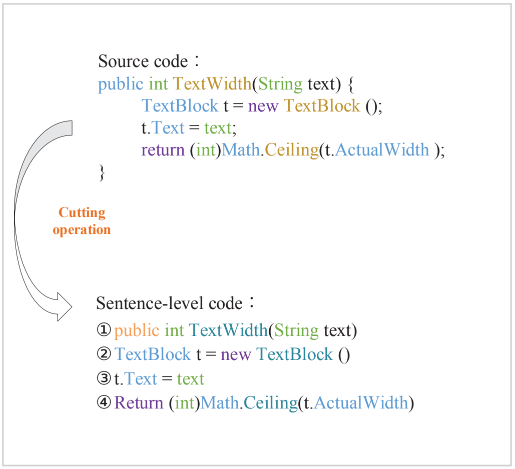
```
Source Code(Java):

public static boolean isValidNmtoken(String nmtoken) {
        if (nmtoken length() == _NUM)
                 eturn _Bool;
        for (int   = _NUM; i < nmtoken length(); i++) {
                char ch = nmtoken charAt( );
                if (!isName(ch)) {
                return _Bool;
                }
        }
        return _Bool;
}
description :
check to see if a string is a valid nmtoken.


public static DateFormat createUTCDateFormatter(String dateFormat) {
        DateFormat formatter = new SimpleDateFormat(dateFormat);
        formatter setTimeZone(TimeZone.getTimeZone(STRING));
        return formatter;
}
description :
creates a dateformat that assumes the incoming date is in utc time.
```

Fig. 1. An example of the source code with clear comment

Java or other, once the location of the code is written wrong, an error message will be reported when running. This shows that not only sequence information but also a lot of structural information is included in the source code. Therefore, when dealing with source code, we should not only pay attention to sequence information, but also pay more attention to their structural information [4].

From CODE-NN [1], which only uses code sequence information, to DeepCom [4], which uses AST as additional information, researchers have gradually realized the importance of structural information. An increasing number of models [3], [5] began to introduce additional data that can represent the structural information of code snippets, such as AST, SBT, etc. It is worth noting that although they were aware of the importance of structural information, they only use the extracted data with structural information as the input of the model instead of exercising the model's ability to learn the structure information of the source code.

Later, a growing number scholars have begun to pay attention to the problem of how to build a model to learn the structural information of the source code. Ahmad et al. [6] used a Transformer-based model to capture the structural information and sequence information of the unprocessed source code. Although their performance surpassed other models, we believe that most of this performance improvement comes from advantages of the Transformer [7] in processing long-distance sequence information. We believe that there will be a better model to capture the structural information from source code. An intuitive method is to use graph structure to capture structural information. Therefore, in this paper, we use heterogeneous graph networks [8] combined with sentence-level code to generate the code comments.


Source code：
```
public int TextWidth(String text) {
        TextBlock t = new TextBlock ();
        t.Text = text;
        return (int)Math.Ceiling(t.ActualWidth );
}
```

Sentence-level code：
```
①public int TextWidth(String text)
②TextBlock t = new TextBlock ()
③t.Text = text
④Return (int)Math.Ceiling(t.ActualWidth)
```

Fig. 2. Conversion of source code to sentence-level code

For the source code, we just divide it into sentence-level code to adapt to our heterogeneous graph networks instead of transforming it into AST or mAST, etc. We consider programming languages to be the same as natural language. Each line of code represents a specific meaning, and each line should be processed independently. Therefore, we divide the original code snippets based on specific symbols, and define these segmented code as sentence-level code. In order to experience this process more intuitively, Figure 2 shows the segmentation results of the sentence-level code. After that, we input the segmented sentence-level code and each token as two types of data into the model for training.

In the heterogeneous graph networks, our processing objects are not complete code sequences, but more fine-grained sentence-level code and token-level code. The token-level node acts as an intermediary connecting sentence-level nodes. All the tokens that exist in the sentence-level code will be connected to this sentence-level node. When the two types of nodes are updated iteratively, the information will be transmitted in the heterogeneous graph networks. When the nodes are updated iteratively, the graph attention network(GAT) [9] mechanism is used to update these nodes.

Using our heterogeneous graphs to process programming languages has the following advantages: (a) Different sentence-level code can interact with each other by the overlapping tokens. (b) Through the sufficient interaction and the graph structure, our model can fully learn the structure information of the code snippets. (c) When the nodes were updated iteratively, the two different granularities of sentence-level code and token-level code will be fully utilized.

Our main contributions can be summarized as follows:

- We propose the heterogeneous graph neural networks to model the relationship between sentence-level and token-level code. The experiment proves that our proposed model can better capture the structural information of the code on the basis of fully capturing the sequence information.
- We divide the source code into sentence-level code like natural language, and feed these sentence-level code and token-level code into our proposed heterogeneous graph networks for learning.
- Even if we do not introduce additional structural knowledge, such as AST and SBT, our model can better capture the structural information of the source code.

II. Relate work

A. *Code Summarization*

Code summarization aims to generate the readable and brief natural language summary that describe the functionality of the program. There have been many studies on automatic code summarization. These studies can be broadly divided into three categories: manually-crafted template [10], [11], information retrieval(IR) [12]–[14] and deep learning approaches [3], [4], [6], [15], [16].

Creating a manually-crafted template is the earliest method used to generate code comments. Haiduc et al. [17] used text retrieval techniques and latent semantic indexing (LSI) to extract keywords. And then they mainly used keyword information to generate code comments. Sridhara et al. [18] also used this method to generate code comment. In their work, Software Word Usage Model (SWUM) was employed to create a rule-based model which was used to generate code comment for Java. Morenoet al. [19] selected information from the original code by pre-defining some heuristic rules, and used this information to generate code summaries. This way of generating code comments is not flexible enough, so the quality of comments generated in this way is not high.

Later, IR-based methods were gradually used to generate code comments. These methods are mainly based on the principles of information theory, which generate comments according to similar code snippets. A typical example of using IR is that the Vector Space Model (VSM) and Latent Semantic Indexing (LSI) were used by Haiduc et al. [13] to generate natural language descriptions for Java code. Wong et al. [20] employed code detection technology to find the most similar code snippets and used them as a comparison to generate a new code summary. These IR-based methods rely heavily on similar code so that they cannot generate readable code comments without similar code.

With the development of deep learning, researchers began to employ deep learning to build language models to generate code comments. Their aim is to build language models to bridge the discrepancy between programming languages and natural languages. Iyer et al. [1] pioneered the use of deep learning techniques to generate code comments. They used RNN combined with the attention mechanism to generate natural language descriptions for C# and SQL. Hu et al. [21] built a multi-input neural network model, and introduced APIs as additional knowledge to assist in model training. Besides they trained a sequence-to-sequence model for API learning. They analogized the neural machine translation model, and introduced the structural information of the code snippets as input. Combine the methods mentioned above, a new code comment generation model [4] was built.

Later, the structural information of the code snippets was highly concerned by researchers [22], [23], and it was regarded as the important auxiliary information. Choi et al. [3] used mAST that integrates structural information as input. In their work, these additional information was processed by the graph convolutional networks. Then it was encoded by Transformer to generate the natural language descriptions. In order to better extract structural information, graph neural networks(GNN) was employed to encode the code snippets in [15], [24]. Similar to the previous work, they also used AST to assist in extracting structural information. Among them, Liu et al. [24] combined the retrieval based technology first used for source code summary with the current generative model based on deep learning, and proposed attention based graph to encode these source code information and retrieved information. The way they used GNN modeling code fragments greatly inspired us to use heterogeneous graphs to build the model. In addition, Wu et al. [5] proposed a novel model for extracting the structure information of the code fragments, named Structure-induced Transformer. Their method is based on the structure information of the code and made improvements on the basis of the transformer. These methods greatly improve the quality of the generated code comments. And at the same time, they are illuminating for the methods proposed in this paper.

To sum up, the source code processing methods of the source code summarization task can be divided into two categories. The first type is to directly use the source code token sequence for encoding. The other method is to use ASTs as auxiliary information to help the model learn the structure information of the source code. Our proposed method does not use AST information, but directly uses the source code token sequence for learning. Specifically, the more fine-grained source code token sequence is the learning object of our model, because we segment the source code token sequence according to certain rules

B. Text Summarization

The purpose of text summarization is to compress a tedious article into a short summary while retaining its important information [25]. Compared to code summarization, the input and the output of the text summarization are both natural languages. The most recent mainstream text summarization methods tend to use the deep learning method [26]–[29].

Li et al. [27] extracted keywords of the document and then used these keywords as key reference information for the model to assist in generating the summary of the document. Li et al. [28] employed double attention pointer networks to obtain text summarization. Compared with the baseline model, the results of their model improve significantly. Huang et al. [29] employed heterogeneous graph networks as the main architecture of the model and introduced Elementary Discourse Units (EDU) as a special node to select important sentences from the document. In these methods, they feed each sentence of a document into the model separately, and modeled the representation of each sentence. They learned each sentence relationship to extract key sentences or generate general summary. It is worth noting that when they process a document, individual sentences will be sent to the model for processing in turn. However, the current method of code summarization is to input a whole piece of code into the model for learning. Combining the methods of Wang et al. [30] and Huang et al. [29], we divide the whole code snippet into sentence-level code and sent sentence-level code into the model for learning. A code snippet is divided into sentence-level and token-level granularities. These granularities are also just suitable for the heterogeneous graph model we proposed.

C. Neural Machine Translation

Neural Machine Translation (NMT) refers to the use of neural networks for automatic translation between different languages [4]. In recent years, the encoder-decoder architecture proposed by Bahdanau et al. [19] has been adopted by almost NMT tasks. The encoder is generally used to automatically extract features and vectorize the input of the model. The function of Decoder is to decode the intermediate tensor generated by the model. In fact, a programming language can be regarded as one language, and the natural language description of these code can be regarded as another language. So that the traditional NMT method can be transferred to translate these two special languages. Adhering to this idea, Hu et al. [4] explored the use of NMT methods for code summarization. Researchers [5], [31] have also made corresponding improvements to the traditional NMT model to adapt to these differences. But their experimental results are not very satisfactory. In this paper, we carry out research from programming language to natural language based on this idea.

III. Proposed Model

In this section, we describe the proposed model CCHG in detail. In order to better connect sentence-level codes and capture the relationship between them. We use heterogeneous graphs with graph attention mechanism to model the relationship between them, and use token-level nodes as intermediate nodes to connect different sentence-level codes. Figure 3 provides an overview of our proposed model. It is worth noting that before sending the source code to the model, we split the code snippets into sentence-level and token-level code. Then we respectively encode these sentence-level and token-level code into vector representations. Then the vectors are input into the heterogeneous graph networks to learn their sequence and structural information. After the iterative update, these sentence-level nodes full of structural information and sequence information will be fed into the decoder to generate code comments.

Suppose there are n sentence-level code in a code snippet $C = \{s1, s2, \cdots, sn\}$. Our goal is to convert this programming language snippet into a natural language that can describe the function of the source code. There are two kinds of nodes in our heterogeneous graph, namely token-level node and sentence-level node. The token-level nodes are composed of each token in the code snippets, and the sentence-level nodes are composed of sentence-level code. Each sentence node is connected to the token nodes it contains. In this way, we construct the code snippet into a heterogeneous graph.

A. Data Preparation

Since in the heterogeneous graph we need two different granularities nodes, but we do not find any sentence-level code in the datasets of code summarization. So we simply process the dataset to adapt it to our model. We divide a code snippet into sentence-level code according to the definition of a program statement. For Java, suppose there is a code snippet with a standard format, which has corrected indentation and correct line breaks and other correct formats. We regard the content before symbol "{" of each line as a sentence-level code, which is mainly a rule for the function name line and the first line of the loop body. Secondly, the content before the ";" was also regarded as a sentence-level code in each line of code. For Python, due to its particularity, we only regard each line of the formatted Python code as a sentence-level code. In addition, we removed a few data that are not suitable for building heterogeneous graphs

B. Heterogeneous Graph Construction

Given a code snippet, in order to fully learn its structure information and sequence information, we use a heterogeneous graph G to represent the code snippet, where $G = \{V, E\}$. The V represents the nodes in our graph and E represents the edges. In addition, the node set $V = \{V_t, V_s\}$ can be further divided, where $V_t = \{t1, t2, \cdots, tn\}$ stands for the token-level node set, $V_s = \{s1, s2, \cdots, sm\}$ stands for the sentence-level code node set. If a code fragment is divided into n sentence-level code, then there will be n sentence-level nodes in the corresponding heterogeneous graph. For token-level nodes, the number depends on how many unique tokens are included in the code snippet. The E in G can also be expressed as $E = \{e_{11}, \cdots, e_{mn}\}$. If the sentence-level code s_i contains token t_j, there will be an edge e_{ij} connecting them for information transfer. Since in the programming language, if a variable is defined above, this variable will definitely be used by the following code. Therefore, the same token will appear in different sentence-level code. Moreover, this way of using heterogeneous graphs to represent sentences and words separately and update them has been proven effective in the field of natural language processing. And this is obviously a natural advantage for the heterogeneous graph networks we constructed. After constructing the graph, a token node will be connected to at least one sentence-level code node.

In order to understand the structure of our heterogeneous graph more intuitively, in Figure 4 we construct a simple code snippet into a heterogeneous graph to help understand. Due to display limitations, we only show how to connect and transfer information between two sentence-level nodes. As depicted in the figure, two sentence-level nodes can be connected through the token-level node "text". In this way, the "text" node acts as an intermediate bridge. Through this bridge, a connection will be established between different sentence-level nodes. Then through the iterative update between the nodes, the structure information and sequence information will be transmitted and captured.

C. Graph Nodes Initialization

We use $\mathbf{X}_t \in R^{m \times d_t}$ to represent the input feature matrix of token, where d_t is the dimension of the token embedding. $\mathbf{X}_s \in R^{n \times d_s}$is used to represent the sentence-level code nodes, where d_s is the dimension of each sentence-level representation vector. We follow Ahmad et al. [6] to use an improved version of the transformer encoder to obtain the matrix of these sentence-level code and token-level code. When coding the source code, absolute position coding is no longer used. Instead, we use relative position coding here. Because many semantics in programming languages no longer depend on the absolute position of the token. For example, the execution results and semantics of $x + y$ and $y + x$ are the same [6].

In the transformer encoder, stacked multi-head attention and parameterized linear transformation layers are used to vectorize the code. In these stacked layers, multi-head attention and residual connections that implement the self-attention mechanism are used to improve model performance. The sequence information obtained by using this encoding method is more sufficient, which is very critical for the generation of final code comments. With such a wealth of sequence information, and coupled with our heterogeneous graph neural networks to capture structural information, it will be easy to generate readable and highly generalized code comments.

It is worth noting that the degree of a token-level node (the number of edges connecting this node) represents the number of times this token appears in the code snippet. A token-level node with a high degree means that there are more edges connecting it to the sentence-level node, and they can collect more information from more neighbor sentence-level nodes. This means that the higher the degree of the token-level nodes, the more information they can obtain from the node iteration process.

D. Graph Attention Layer

In the above section, we constructed the code snippet as a heterogeneous graph, and use vectorized sentence-level and token-level code to initialize our heterogeneous graph nodes. In order to fully capture the sequence information and structure information of the code snippets, in this section,

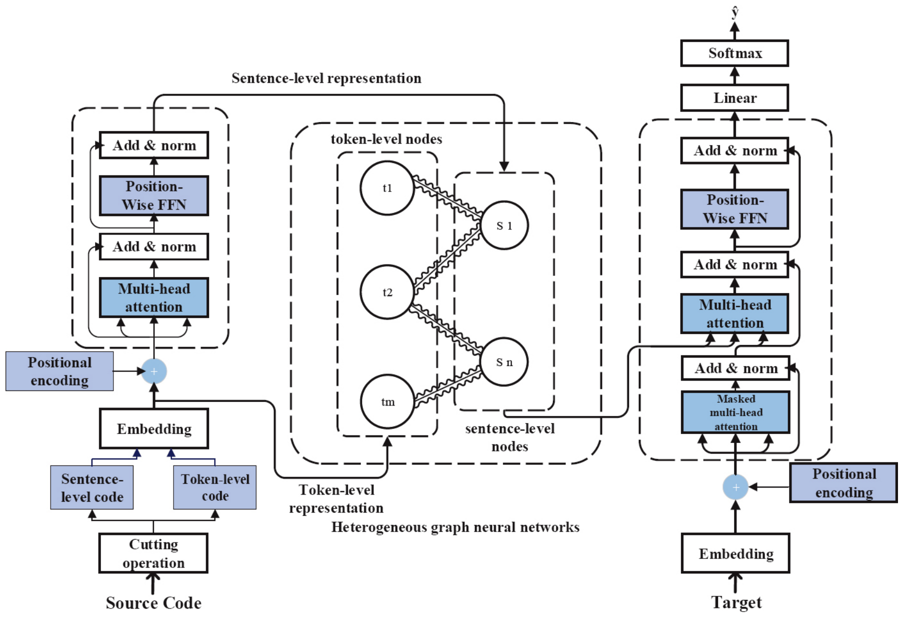
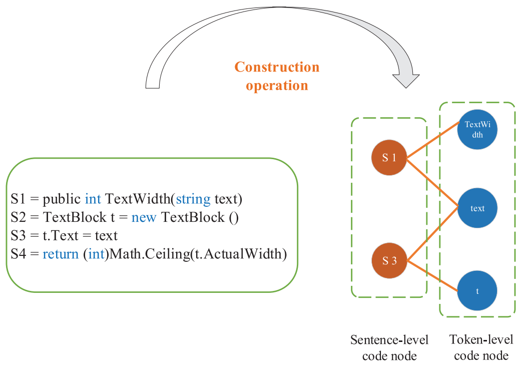


Fig. 3. The framework of our proposed model. In the graph attention network layer, s represents sentence-level code, and t represents token-level code. The waves in the graph represent the multi-head attention networks



Construction operation
S1 = public int TextWidth(string text)
S2 = TextBlock t = new TextBlock ()
S3 = t.Text = text
S4 = return (int)Math.Ceiling(t.ActualWidth)
S 1
S 3
TextWidth
text
t
Sentence-level code node
Token-level code node

Fig. 4. An example of heterogeneous graph construction. It clearly shows how the two sentence-level nodes are connected

we iteratively update the nodes, and continuously integrate the node information. When updating the nodes, the graph attention network(GAT) [9], an improved version of the graph convolutional network, is used to update the representations of our sentence-level and token-level code nodes.

In each round of iterative updates, sentence-level code nodes are updated by their neighbor nodes. We define h_i as the hidden layer state of the node i, where $i \in \{1, \cdots, m+n\}$. The graph attention mechanism and the adjacent nodes are employed to update the hidden layer state h_i of the i^{th} node. After each round of update, each node in the heterogeneous graph will get the important information from its neighbor nodes. Due to the repetition of tokens, important information of indirectly connected sentence-level nodes will be transmitted. The update formulas of the Graph Attention network are as follows:

$$e_{ij} = a\left([Wh_i \| Wh_j]\right), j \in N_i \tag{1}$$

$$\alpha_{ij} = \frac{\exp\left(\text{Leak ReLU}\left(e_{ij}\right)\right)}{\sum_{k \in N_i} \exp\left(\text{ LeakyReLU }\left(e_{ik}\right)\right)} \tag{2}$$

$$h_i' = \sigma\left(\sum_{j \in N_i} \alpha_{ij} W h_j\right) \tag{3}$$

In the above formula, i and j represent the i^{th} and j^{th} graph node, and α_{ij} represents the attention weight between the i^{th} node and the j^{th} node. In Equation 1, where $[\cdot\|\cdot]$ is to concatenate the two tensors, $\alpha\left(\right)$ means to map a tensor to a real number. Then SoftMax are employed to normalize the correlation coefficient e_{ij} as shown in Equation 2. As shown in Equation 3, after obtaining the attention coefficient, we add the attention weights of all neighbor nodes of node i, where N_i is the set of all neighbor nodes of node i, and σ is the sigmoid function. As an enhanced version of the attention mechanism, the multi-head attention mechanism is used to capture richer information from neighbor nodes. Its formula is defined as follows:

$$h_i'(k) = \|_{k=1}^{K} \sigma\left(\sum_{j \in N_i} \alpha_{ij}^{k} W^k h_j\right) \tag{4}$$

where K represents the number of heads in the multi-head attention layer. Then a position-wise feed-forward (FFN) layer containing two linear transformations will be used to process the results of the graph attention processing.

In order to understand the update process of nodes more intuitively, Figure 5 shows the update process of nodes with different granularities. In this example, we use the code snippet used in Figure 4 to build a heterogeneous graph. The same token-level node may connect multiple sentence-level nodes, which enhances the connection between different sentence-level nodes. Before the node update process, we initialize the token-level node and sentence-level node with the methods mentioned above. Then, we use token-level nodes to update sentence-level nodes through the GAT. As shown in Figure 5, a sentence-level node is updated by the token-level nodes it contains. In turn, this sentence-level node will update the token-level nodes it contains in the next process. The "text" token-level node in Figure 5 appears in the first sentence-level code and the second sentence-level code at the same time. Therefore, the S_1 node and S_3 node are connected by the "text" token node for information transmission. As shown in part b of Figure 5, the "text" node receives the information from S_1 and S_3. And since our iterative update process will carry out multiple times, so the information will be fused between different sentence-level code nodes during the next node update process. For example, the "text" node that integrates the information of S_3 will pass the information to S_1 in the next round of updates.

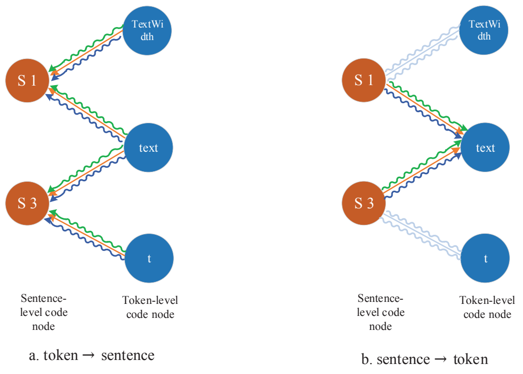


Fig. 5. Schematic diagram of iterative update of nodes in a heterogeneous graph

The above process not only explains how the information is transmitted in the graph, but also how the two sentence-level code nodes without edges convey the information. It can be seen that each iteration of the node in this process consists of the process of token-level to sentence-level and sentence-level to token-level. The i^{th} iterative update process can be expressed as:

$$\mathbf{N}_{t\to s}^{h+1} = \mathrm{GAT}\left(\mathbf{H}_t^h, \mathbf{H}_s^h, \mathbf{H}_s^h\right) \tag{5}$$

$$\mathbf{N}_{s\to t}^{h+1} = \mathrm{GAT}\left(\mathbf{H}_s^h, \mathbf{H}_t^h, \mathbf{H}_t^h\right) \tag{6}$$

The iterative update process of nodes in a certain round can be expressed by Equation 5 and Equation 6. Where t represents a token-level node, and s represents a sentence-level node, and h represents time step. $t \to s$ means that the sentence-level node is updated by the token-level node, and $s \to t$ means that we use the sentence node to update the token-level node.

E. Prediction Layer

After the iterative update in the heterogeneous graph network, the sentence-level nodes obtain a wealth of information by interacting with token-level nodes.So we select these sentence-level nodes from the heterogeneous graph and use them as intermediate vectors for decoding. Then our task is to decode these vectors to generate the natural language description of the source code. Since we use the encoder of the transformer to vectorize the original code when initializing the node, thus it is a good choice to use the decoder of the transformer when decoding. It is worth noting that different from the relative position coding in the encoder, we use absolute position coding when embedding the target sentence in the decoder. So in the decoding stage, we follow Ahmad et al. [6] to use the decoder of transformer. The cross-entropy loss is used to update the weights during the training process.

IV. Experiment

A. Dataset

Since we want to divide the code snippets into sentence-level code, we choose Java that is well recognizable in format as the experimental dataset. This is because the Java has obvious sentence ending characters ";", and every code block is wrapped by "{}". This coding style is convenient for us to divide the code snippets into sentence-level code. In addition, we also choose python as the experimental dataset. Due to its conciseness and powerful functions, Python has been admired by many people in recent years. So in view of its popularity, we choose it as another dataset for our experiment. But because Python pursues brevity, there are no sentence ending characters and some structural symbols in its encoding rules. So we simply select each line as the sentence-level code. And the python and java datasets used in our experiments come from TransRel [6]. The statistics of these two datasets are shown in Table I, and the way of segmentation of code snippets is as introduced in section III-A.

TABLE I
Statistics of Java and Python Dataset

Dataset	Python	Java
train	55538	69708
valid	18505	8714
test	18502	8714
Avg. sentence-level code[a]	11.39	9.65

[a]It represents the average number of sentence-level code that are segmented

B. Metrics

1) Bleu: The Bleu [32] metrics was first used in the evaluation of machine translation. Its main calculation formula is as follows:

$$BLEU = bp \cdot \exp\left(\sum_{n=1}^{N} w_n \log c_n\right) \tag{7}$$

Where n represents n-gram, w represents the weight of n-gram, c_n represents the coverage of n-gram, and bp represents brevity penalty, which is short sentence penalty factor.

2) METEOR: Meteor [33] considers the case where the meaning of the sentence generated by the model is correct but the expression does not match the reference sentence.Its formula is shown in Equation 8. Where p is the penalty factor. The penalty here comes from the difference between the word order in the candidate translation and the word order in the reference translation.

$$\text{METEOR } = (1 - \text{ p }) \times F_{\text{means}} \tag{8}$$

3) ROUGE: Rouge [34] and Bleu are very similar. It calculates the length of the longest common subsequence between the generated sentence and the reference sentence. The longer the length, the higher the score. Its main formula is as follows:

$$ROUGE - L = \frac{\left(1+\beta^2\right) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{9}$$

$$R_{lcs} = \frac{\text{LCS}(\hat{y}, y)}{m} \tag{10}$$

$$P_{lcs} = \frac{\text{LCS}(\hat{y}, y)}{n} \tag{11}$$

Where $\hat{y}$ represents the generated sentence, y represents the reference sentence, $LCS(\hat{y}, y)$ represents the length of the longest common subsequence between the generated sentence and the reference sentence, m represents the length of the reference sentence, and n represents the length of the generation.

C. Experimental Settings

The model we implement using the Pytorch[1] framework. We follow Wei et al. [35] to limit the size of the dictionary and the length of the code snippets. The relevant parameters of the model are as follows: 1)we train the CCHG models using Adam optimizer [36] with an initial learning rate of 10^{-4}; 2)the batch size and dropout rate is 32 and 0.2, respectively; 3)the dimension of the sentence-level code nodes are initialized to 512; 4)the number of heads of multi-head attention in GAT is 8; 5)the hidden size of FFN is 512; 6)with reference to the performance of the model on the validation set, we set the number of iterative updates of the nodes in GAT to 2; 7)We train the CCHG models for a maximum of 200 epochs and an early stop is performed if the validation performance does not improve for 20 continuous iterations.

[1]https://pytorch.org/

D. Baselines

RNN-based Approaches This category includes related models that use RNN, including those that use sequence information and non-sequence information [1], [4], [35], [37]. The method of generating natural language comments from source code based on RNN was first proposed by Iyer et al. [1]. They use RNN combined with the attention mechanism to generate natural language descriptions for source code. Then many researchers extended their models by using additional knowledge such as AST, API, etc.

Transformer-based Approaches Since the Transformer model has excellent performance on many generation tasks, Ahmad et al. [6] used it to generate natural language descriptions of source code. And they used relative position encoding (RPE) [38] and copy attention [39] to enhance model performance. Later in [3], [5], the Transformer-based model that paid more attention to the structural information of the source code was proposed for code comment generation.

Pre-training Approaches CodeBERT [40], a pre-trained language model in the field of code processing, is also used as a comparison between our proposed models. This model is trained from a huge code corpus and has achieved good performance on downstream tasks such as code summarization, code retrieval and code generation.

Graph-based Approaches We also compare our model with other approaches [15], [41] that use graphs to generate code summarization. These methods are also the most similar to the method we proposed. For example, LeClair et al. [15] input AST into ConvGNN, and the output of the graph is processed by GRU. Finally, a context vector is generated through a double attention.

V. Results and Analysis

A. Results on Java

Left part of Table II shows the results on Java. We can see that the transformer-based model proposed by Ahmad et al. [6] is very powerful, which greatly surpassing the performance of other baseline models. In view of its superior performance, our model also takes advantage of the transformer. Nevertheless, our proposed CCHG model using sentence-level code and heterogeneous graph has been further improved on this basis. Our model achieves higher scores, reaching 45.69, 27.32 and 54.98 on BLEU, METEOR and Rouge-L respectively. It can be seen from the performance in Table II that the way we process sentence-level code and token-level code separately through heterogeneous graphs is very effective. Compared with the GCN model, the performance of our GAT model has made great progress. We believe that compared to the same graph-based models (such as GCN-based models), our model performs better for two reasons. The first is that we split the original lengthy source code sequence into subsets, which allows our encoder to encode the source code better. The other is that the multi-head attention mechanism used in our heterogeneous graph network can better aggregate information between different codes than the GCN aggregation mechanism.

TABLE II
PERFORMANCE OF OUR PROPOSED MODELS AGAINST RECENTLY RELEASED BASELINE MODEL ON JAVA AND PYTHON

	Java			Python		
Models	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
CODE-NN	27.60	12.61	41.10	17.36	09.29	37.81
Tree2Seq	37.88	22.55	51.50	20.07	08.96	35.64
RL+Hybrid2Seq	38.22	22.75	51.91	19.28	09.75	39.34
DeepCom	39.75	23.06	52.67	20.78	09.98	37.35
API+CODE	41.31	23.73	52.25	15.36	08.57	33.65
Dual Model	42.39	25.77	53.61	21.80	11.14	39.45
CodeBERT	43.33	26.20	54.64	33.47	21.69	49.35
TransRel	44.58	26.43	54.76	32.52	19.77	46.73
mAST+GCN	45.49	27.17	54.82	32.82	20.12	46.81
CCHG-RNN	45.51	27.10	54.83	32.75	20.45	46.92
CCHG-base	45.69	27.32	54.98	33.01	20.53	47.26

Note: The results of the baseline methods are directly reported from Ahmad et al. [6] and Choi et al. [3]. CCHG is the full model we proposed, and CCHG-RNN refers to the model that uses RNN.

B. Results on Python

The performance of CCHG on the python dataset is shown in the right part of Table II. The transformer-based method proposed by Ahm ad et al. [6] greatly improves the performance of the previous baseline model on the Python dataset. Compared to the Dual Model's [35] 21.80 on BlEU, it rises to 32.52, an increase of 10.72. Our model also takes advantage of the transformer. We use its encoder to encode sentence-level code snippets and then uses heterogeneous graphs to iteratively update the nodes information. On the Python dataset, CCHG is improved by 0.49, 0.76, 0.53 on BLEU, METEOR and Rouge-L respectively than the transformer-based model [6]. Compared with other graph-based models, such as mAST+GCN, our model has also improved by 0.19, 0.41 and 0.45 on BLEU, METEOR and Rouge-L respectively.

C. Analysis

From Table II, we can see that the score of the Python dataset is obviously much lower than that of the Java dataset. It is particularly obvious in the Bleu metric; the gap is as high as 12.68. We think there are two reasons for this result. Firstly, when we divide the code fragments into sentence-level code, structural symbols such as "{}" and ";" exist in the Java language. Therefore, we are able to divide the code snippets into sentence-level code in a more detailed and accurate manner. There are no such symbols in python, so we can only simply treat each line of a multi-line code block as a sentence-level code. Such a segmentation method may cause our model to fail to adequately capture the structural information and sequence information of the code fragments. Secondly, according to our statistics, the number of different tokens in the python dataset is 56189, while the number of different tokens in Java is 46895. Python has nearly 10,000 more tokens than the Java dataset, which greatly increases the difficulty of our model learning from the Python dataset. We think this is also one of the reasons for the degradation of model performance.

In addition, in Figure 6 and Figure 7, we show the trend of the performance of our model on the java and python datasets. In the figure, we respectively depict the trends of Bleu, Rouge-L and F-1 during training. Due to the particularity of the Meteor metric, we only use it as the evaluation standard on the test set. In Figure 6 and Figure 7, we use F-1 instead of it to draw the curve. As for stability, since the neural networks have not been fully trained at the beginning of training, there are some fluctuations in the performance of the model. With the training of the model, the parameters of the neural network are constantly updated. The fluctuation of model performance is also decreasing. Them stabilize at around 150 rounds.

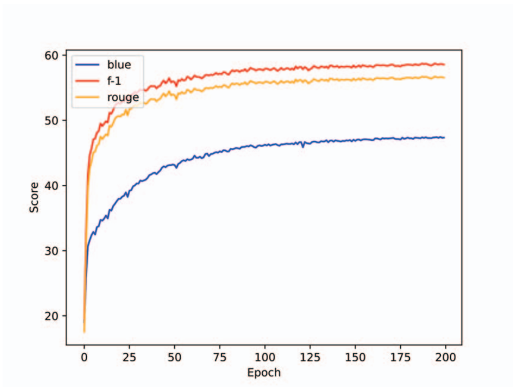


Fig. 6. The performance of CCGH on Java

D. Iterative update

In the iterative update process of the heterogeneous graph nodes, the information stored in the node will change after each round of updates. The next round of updates to the same node will have a different effect. Therefore, the number of node updates will have a certain impact on the final result. Based on the above considerations, we conduct experiments on the Java dataset with different iteration times. Note that

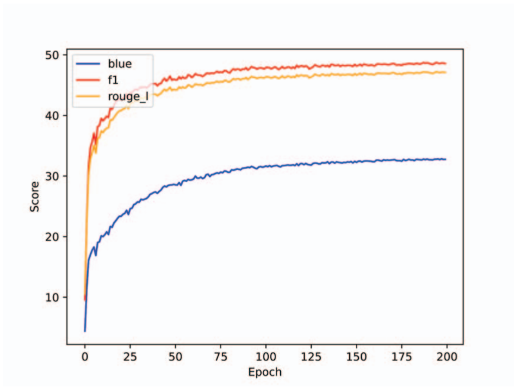


Fig. 7. The performance of CCHG on Python

the number of iterations here refers to the number of repeated updates between token-level nodes and sentence-level nodes.

In order to choose the best iteration round, we carry out experiments of repeated iteration updates from 1 to 5 times on the test set of Java. When we set the number of iterations to 2, our CCHG achieve the best performance. Therefore, in our experiment, we finally fix the number of updates of the nodes in the heterogeneous graph to 2 times. Although the performance of the model may be improved when the number of iterations is higher than 5, considering that the cost of training time will increase exponentially, we do not continue to increase the number of iterations.

E. Generated example

In order to experience our work more intuitively, we respectively enumerate two examples of Java and Python in Figure 8. As can be seen from the figure, the code comments generated by our model are very similar to the reference comments for both Java code and Python code. Although some words in the predicted are different from the reference summarization, the meanings expressed by our predicted and referenced summarization are almost the same. Just like the second example of Java, although the predicted and referenced sentences look different, it does not hinder our understanding of the meaning of the code snippets.

F. Ablation study

Because the data in our model is processed by the encoder of the transformer before passing through the GAT layer. The transformer is proved to achieve good performance when processing code summarization tasks by Ahmad et al. [6]. In order to prove the effectiveness of decomposing code fragments into sentence-level code and using heterogeneous graphs to process these sentence-level code, we do the following ablation experiments. We use the traditional RNN structure as the encoder instead of the encoder of transformer. As shown in Table III, we show the change in model performance after changing the transformer to RNN. On the Java dataset, the Bleu, Meteor, and Rouge-L metrics dropped to 45.51, 27.10, and 54.83, down 0.18, 0.22, and 0.15, respectively. For the python dataset, Bleu, Meteor and Rouge-L dropped by 0.26, 0.08 and 0.34 respectively. From these data, we can see that even though we use traditional RNN instead of transformer to process the sentence-level code and token-level code, our model can achieve considerable results too. This shows that the iterative update between different types of nodes in heterogeneous graph neural networks is indeed very helpful for capturing the structural information and sequence information of the code snippets. At the same time, it also proves that the performance of our model does not depend on the performance of transformer but mainly comes from the use of heterogeneous graphs to process the sentence-level and token-level code.

We compare the Bleu scores of the two models which based on RNN and Transformer. It can be seen from Figure 9 that after switching from Transformer's encoder to traditional RNN, the performance of the model does have some decline. But these declines are small, only about 0.2. These faint drops show that despite the powerful performance of Transformer, our model still has excellent performance after using RNN instead of Transformer. At the same time, these faint drops prove that our heterogeneous graph model can fully capture the structural information and sequence information of the code snippets.

In addition, we set the number of node updates in the graph to 0, which means that the node data has not passed through the GAT layer. Therefore, the token-level nodes and sentence-level nodes in the heterogeneous graph are only initialized but not updated through the graph attention networks. From Table III, we can see that without node update, the performance of our model drops significantly. Compared with our CCHG(base) on the Java dataset, without node update operation will cause the scores of Bleu, Meteor, and Rouge-L to drop by 1.19, 0.87, 0.43, respectively. There are also 0.41, 0.73 and 0.38 drops on the Python dataset. As shown in Table III, the performance degradation rate caused by not using node update is more than twice that of not using transformer as the encoder, which confirms the importance of our heterogeneous graph networks and using sentence-level code.

VI. Conclusion and Future Work

In this paper, we propose a code summarization model CCHG. We do not use the entire code snippet for processing, nor do we use additional knowledge such as AST to aid training. Instead, we divide a piece of code into sentence-level code and token-level code. In addition, we use heterogeneous graph networks to process these different granularities of code. Experiments show that our model can fully capture structural information and sequence information, and use this information to generate high-quality code comments. In future work, we will add other types of nodes to the heterogeneous graph, such as using keywords as new nodes. We will also train our model in combination with pre-training.

Java

```
public void addHandler(String columnName, SQLDataHandler
handler)
{
   if(m_overrides == null)
        m_overrides = new HashMap(NUM);
   m_overrides.put(columnName, handler);
}
```

Reference：add a custom data handler for a given column name.
Prediction：add a custom data handler for a column name.

```
public double dist(MathVector other)
{
   double dist_X = this.x - other.x;
   double dist_Y = this.y - other.y;
   return Math.sqrt(dist_X * dist_X + dist_Y * dist_Y);
}
```

Reference：calculate each vector's distance.
Prediction：returns the distance from this vector and another.

Python

```
def _DeepCopySomeKeys(in_dict, keys):
    d = {}
    for key in keys:
        if key not in in_dict:
            continue
        d[key] = copy.deepcopy(in_dict[key])
    return d
```

Reference：performs a partial deep-copy on |in_dict|.
Prediction：returns a new dict whose keys are copied.

```
def extract_description(texts, document):
    for text in texts:
        try:
            document + text['description']
        except KeyError as e:
            print('KeyError %s\n%s' % e, text)
    return document
```

Reference：returns all the text in text annotations as a single string.
Prediction：returns text annotations as a single string.

Fig. 8. Examples of code comments generated by our model

TABLE III
COMPARISON OF THE RESULTS OF USING DIFFERENT ENCODERS

	Java			Python		
Models	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
Base	45.69	27.32	54.98	33.01	20.53	47.26
RNN-based	45.51	27.10	54.83	32.75	20.45	46.92
t=0[a]	44.50	26.45	54.55	32.60	19.80	46.88

[a]Here t=0 means that the nodes are only initialized but not updated. In this way, there is no interaction between nodes, which is equivalent to removing our heterogeneous graph neural networks

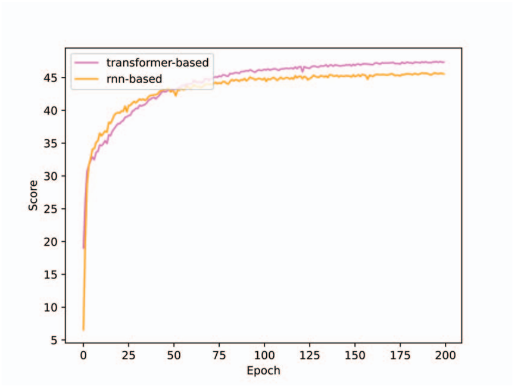


Fig. 9. Comparison of BLEU scores of RNN-based and Transformer-based models

ACKNOWLEDGMENT

This work was supported in part by the National Social Science Foundation under Award 19BYY076, in part Key R & D project of Shandong Province 2019JZZY010129, and in part by the Shandong Provincial Social Science Planning Project under Award 19BJCJ51, Award 18CXWJ01, and Award 18BJYJ04.

REFERENCES

[1] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "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)*, 2016, pp. 2073–2083.

[2] S. Haque, A. Bansal, L. Wu, and C. McMillan, "Action word prediction for neural source code summarization," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 330–341.

[3] Y. S. Choi, J. Y. Bak, C. W. Na, and J. H. Lee, "Learning sequential and structural information for source code summarization," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 2842—-2851.

[4] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension*, 2018, pp. 200–20 010.

[5] H. Wu, H. Zhao, and M. Zhang, "Code summarization with structure-induced transformer," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2020, pp. 1078–1090.

[6] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[8] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 793–803.

[9] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[10] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[11] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering*, 2011, pp. 101–110.

[12] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *2010 acm/ieee 32nd international conference on software engineering*, 2010, pp. 223–226.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*, 2010, pp. 35–44.

[14] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 562–567.

[15] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.

[16] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, pp. 2179–2217, 2020.

[17] S. Haiduc and A. Marcus, "On the use of domain terms in source code," in *2008 16th IEEE International Conference on Program Comprehension*, 2008, pp. 113–122.

[18] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[19] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations*, 2015.

[20] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 380–389.

[21] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 2269–2275.

[22] Y. Huang, S. Huang, H. Chen, X. Chen, Z. Zheng, X. Luo, N. Jia, X. Hu, and X. Zhou, "Towards automatically generating block comments for code snippets," *Information and Software Technology*, p. 106373, 2020.

[23] R. Shahbazi, R. Sharma, and F. H. Fard, "Api2com: On the improvement of automatically generated code comments using api documentations," *arXiv preprint arXiv:2103.10668*, 2021.

[24] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," in *International Conference on Learning Representations*, 2020.

[25] Z. Liang, J. Du, Y. Shao, and H. Ji, "Gated graph neural attention networks for abstractive summarization," *Neurocomputing*, vol. 431, pp. 128–136, 2021.

[26] F. Mohsen, J. Wang, and K. Al-Sabahi, "A hierarchical self-attentive neural extractive summarizer via reinforcement learning (hsasrl)," *Applied Intelligence*, pp. 1–14, 2020.

[27] H. Li, J. Zhu, J. Zhang, C. Zong, and X. He, "Keywords-guided abstractive sentence summarization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8196–8203.

[28] Z. Li, Z. Peng, S. Tang, C. Zhang, and H. Ma, "Text summarization method based on double attention pointer network," *IEEE Access*, vol. 8, pp. 11 279–11 288, 2020.

[29] Y. J. Huang and S. Kurohashi, "Extractive summarization considering discourse and coreference relations based on heterogeneous graph," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics*, 2021, pp. 3046–3052.

[30] D. Wang, P. Liu, Y. Zheng, X. Qiu, and X.-J. Huang, "Heterogeneous graph neural networks for extractive document summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 6209–6219.

[31] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *2021 IEEE/ACM 29th International Conference on Program Comprehension*, 2021, pp. 184–195.

[32] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[33] S. Banerjee and A. Lavie, "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*, 2005, pp. 65–72.

[34] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[35] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Advances in Neural Information Processing Systems*, 2019, pp. 6563–6573.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[37] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[38] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018, pp. 464–468.

[39] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 1073–1083.

[40] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 1536–1547.

[41] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *International Conference on Learning Representations*, 2018.