UTS UNIVERSITY OF TECHNOLOGY SYDNEY

Static Analysis-Guided Automatic Source Code Summarization via Deep Learning

by Wenhua Wang

Thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

under the supervision of Guandong Xu

University of Technology Sydney
Faculty of Engineering and Information Technology

December 2021

Certificate of Original Authorship

Required wording for the certificate of original authorship

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Wenhua Wang, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Software, Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.
**If applicable, the above statement must be replaced with the collaborative doctoral degree statement (see below).*

**If applicable, the Indigenous Cultural and Intellectual Property (ICIP) statement must be added (see below).*

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: 9/7/2022

ACKNOWLEDGMENTS

As a student of the Joint PhD Training Program offered by the University of Technology Sydney (UTS) and Southern University of Science and Technology (SUSTech), I would like to express my heartfelt gratitude to my supervisors, Guandong Xu in UTS and Yunquan Zhang in SUSTech, for their supervision as well as to UTS and SUSTech for their support for my studies in both universities. Besides, I also want to thank Yulei Sui in UTS and Shin Hwei Tan, Yepang Liu in SUSTech for their help when I encounter difficulties and bottlenecks and Yuanyuan Dou in SUSTech for her careful work to support our daily research work. Thank you all.

Thanks to the staff and teachers of the GRS in UTS and SUStech graduate school for their efforts to our daily life support. Thanks to teacher Xing Zhang in School of Engineering, SUSTech to offer help when I encounter in my life and he also organized various activities for example hiking and mountain climbing which relax us when we felt tired. Thanks to teachers in Department of Computer Science and Engineering, SUSTech and School of Computer Science at UTS who provides a comfortable scientific research environment. Thank you all.

Then, I want to thank the students I work with in both UTS and SUSTech. Thank you for your daily help in both my research and my life. Thank you for the warmhearted conversation when I feel confused. Thank you for going sightseeing and tasting delicious food together with me on holidays which let me know I am not lonely. Thank you for your warm solicitude when I am sick. Thank you for your talking with me about our research progress and solve the problems together. Thank you all.

Next, I want to thank all my lovely friends. When I feel sad and doubts about life and the future, it's your patience that enlightens and removes my doubts. When I encounter major difficulties in life, you give me great help. Thank you for being with me all the time, so that I can maintain my hope and longing for a better life in the future when I feel confused and helpless. Thank you all.

At last, I want to thank my family. I want to express my heartfelt gratitude to my parents who have supported me for all these thirty years, you are the source of my continuous progress in these years. Thanks to my sister, my grandmother and other members in my big family. Thank you all.

Thank you all!

List of Publications

Related to the Thesis :

1. Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. Reinforcement-Learning-Guided Source Code Summarization using Hierarchical Attention. IEEE Transaction on Software Engineering, 2021.

2. A Transformer-based Generative Adversarial Network Framework for Universal Code Summarization, submitted.

3. On Semantic-rich Code Summarization by Vital Code and Transformer-based Model, submitted.

Abstract

Abstract. Code summarization provides the main aim described in natural language of the given function, it can benefit many tasks in software engineering. As far as I known, the existing research on comment generation can be summarized as the template based approaches, the information retrieval based approaches and the deep learning based approaches. Nowadays, based on the proposal and wide utilization of deep learning, the research of neural machine translation has been introduced to the research of code summarization. Based on my study, The existing deep learning based code summarization approaches mainly utilize the seq2seq model in which the encoder translates the source code into hidden representation of the program code and then the decoder decodes the representation into comment. However, due to the special grammar and syntax structure of programming languages and various shortcomings of different deep neural networks, the accuracy of existing code summarization approaches is not good enough. These approaches mainly suffer from three major drawbacks: a) they regard the source code as plain text directly, which neglecting the syntax structure of the source code that is quite important for the comprehension of source code; b) they only consider the generation of the code's intent, while ignore the information of parameters etc which is also quite important for the understanding and usage of the source code; c) their adopted CNN/RNN model usually cause long-distance dependency and excessive computation cost problem. Considering these limitations, the main research work of this thesis are as follows: (1) The first work proposes to adopt the hierarchical attention mechanism to enable the code summarization framework to translate three representations of source code to the hidden spce and then it injects them into a deep reinforcement learning model to enhance the performance of code summarization. (2) While many existing approaches exploit inadequate power of statement-wise semantic contributions for augmenting their performance, the second work proposes the transformer-based generative adversarial network framework for universal code summarization which constructs a cross-language universal hierarchical semantic (UHS) model to classify statements according to their positions in the source code. (3) Considering that almost all approaches only consider to generate the general intent of the method without documenting their parameters, the third work proposes to generate both the method comment and the parameter comment to provide complete java documentation for the code snippets. Specifically, it designs a programming-analysis-based component to extract UseSet of parameter and the KeySet of the source code to obtain the main semantic information and discard the useless noise information and utilizes the copy-attention-integrated transformer based NMT

framework. Thought the completion of this thesis, I conduct a few of experiments, and the results of which prove that the proposed approaches can obtain better accuracy compared with the baseline approaches.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE **Page**

List of Tables

Table **Page**

CHAPTER 1

INTRODUCTION

1.1 Research Background and Significance

In recent years, with the development of computer science, many aspects of people's daily life, such as transport, entertainment, healthcare, energy, social security, depend more and more on software, while the development of software is a time and labor consuming task. In fact, almost 90% labor and time is spend on the maintenance of software during the software development life cycle (e.g., implementation, testing and maintenance). Furthermore, much of the effort is spend on the comprehension of source code and the target task [57]. Additionally, it is fact that meaningful source code comment can be helpful for all kinds of research in the software engineering area [27, 123], e.g., software testing [114, 121, 122], fault localization [56], program repair [29, 96].

Thus, it is quite important for the source code documentation to describe the intent of the source code for further maintenance of the software. Although many techniques have been developed to help developer for software implementation and testing, while writing meaningful comment for the source code is still labour consuming [4, 16, 95]. While in the real world projects, there are not plentiful code comment supplied for future maintenance [24, 45]. In fact, it is quite time-consuming and challenging for the programmer to supply enough good code comments when their time is tight. In general, The comments supplied by the developer or generated by the automatic code summarization approach should satisfy the following features: (1) *Correctness*. The comment must give the correct intent of the source code and how to utilize it. (2) *Fluency*. The comment must be fluent natural

language thus anyone reading it can comprehend them easily. In this way, the research on code summarization target on designing a method that can comprehend the intent of the program code and then produce the annotations directly according to the comprehension of the program code automatically.

The rapid development of machine learning and deep learning has made it widely used in all aspects of social life, especially in the area of natural language processing (NLP) [21]. For example, natural language translation (NMT) [11] using the deep learning technology has achieved great improvement in accuracy and precision. Recently, researchers have applied NLP models to the research about Program Language Processing (PLP). The focuses of PLP include automatic code summarization [71], semantic analysis [81], code generation [115], function summary [43] etc. Among them code summarization is a hot and crucial research direction.

Code summarization mainly generates the intent of a code snippet described by natural language performed by certain programming language (for example, Java, Python, etc.). Code summarization is important for the research of all fields. What will you do when you obtain a snippet of code and want to know its purpose? On the other hand, the researcher from other research areas also want to know whether a snippet of source code they obtain can achieve their special goals and they may also want to perform a simple change to satisfy their specific requirement. All of these need them to learn the special programming language which has its own grammar or syntactic and may be difficult for the researcher who has little knowledge about computer science, especially about computer programming. Assume that if the research of code summary generation can make great breakthrough and results. Through a certain model to generate the code annotation, we can easily obtain the description of the source code we obtained, and then the corresponding software development process will greatly simplified. Besides, the researcher from other research areas can understand their downloaded source code and know how to use them easily. Thus, this research can not only reduce the workload of software engineers but also help the researchers from other areas who want to use the computer to process their data and so on.

At present, the research on comment generation can be summarized as template [91, 92], information retrieval(IR) [35, 36, 108] to learning-based approaches [8, 43, 71]. One of the most common comment generation approaches is to design different templates for certain source code type. Sridhara et al. create the rule-based model according to software Word Usage Model (SWUM) to generate the description of the Java method [91]. Moreno et al. grab the information and then produce annotations

by the predefined heuristic rules and then combining them [68]. Besides, some special information related to the source code, e.g., test cases [123] and code changes [19] are also adopted for the comment generation task. These approaches mainly deduce the annotations by grabbing important words from the supplied code snippets and filling in the human designed templates. The information retrieval based approaches are widely researched which usually grab the annotations from the similar program code or utilize the information retrieval algorithms to obtain the words with high frequency to compose the code intention. Fo instance, Haiduc et al. [36] combine different information retrieval models, i.e., VSM (Vector Space Model) and LSI (Latent Semantic Indexing), to deduce the comments of Java classes and functions. Eddy et al. replicate and exploit this work by adopting the hierarchical topic model [26]. On the other hand, Wong et al. [108] propose to utilize the code-annotation pairs to produce the description for the code snippet grabbed from the open-source projects. At the same time, they also utilize code clone to look for similar program code [107]. With the development of machine learning and deep learning, applying the deep learning method to solve the code summarization problem can be promising and effective. The deep learning based approaches mainly consider the programming language as a special natural language with complex syntax and grammar, and utilize the neural machine translation model, which follows the framework of encoder–decoder, to generate the summary of the source code. For example, Gu et al. [33] use the seq2seq neural network [94], which is followed for the statistical machine translation, to produce the middle representations of the query written in natural language which they used for the prediction of the API sequences.

Although source code summarization has achieved certain performance, it has not been widely used in practice. The main reasons are as follows:

- The template-based approach and the information-retrieval-based approach can only generate comment with limited word provided by the source code, thus, they can not generate comment which need the comprehension of the source code.

- Applied the deep learning method to solve the code summarization problem can be promising and effective, compared with natural language, program language has stricter semantic model and grammatical structure, however, existing work has not considered the intrinsic feature of source code, therefore, the representation of the source is not accurate enough.

- Existing work only consider the generation of the source code intent, while ignore the information of parameter etc which is also quite important for the understand-

ing and usage of source code.

- In addition, the adopted CNN/RNN model usually cause long-distance dependency and excessive computation cost problem.

Considering the limitations of the existing code summarization approaches described above, this thesis proposes different solutions accordingly, which are presented as follows:

- For the source code representation, different code features are introduced, I first utilize the sequenced abstract syntax tree and sequenced control flow graph of the code to compose the representation together with the plain text form.
- To exploit the power of statement-wise semantic contributions for augmenting their performance, this thesis constructs a cross-language universal hierarchical semantic (UHS) model to classify statements according to their positions in the source code.
- Consider the integrity of the generated comment, I propose to generate both the general intent and the parameter comment of the source code, for which I propose a statements selection approach to select the important statement and the usage statement for the general intent and the parameter comment generation respectively.
- For the limitation of the deep learning models, I propose to utilize different models and modify them considering the feature of different source code representations.

1.2 Research Contents and Innovations

In this thesis, I focus on the deep-learning-based code summarization. The work of this thesis mainly focus on the representation of the source code, the selection of statements and the utilization of different neural networks and their improvement. The main research contents of this thesis are as follows:

Research content 1: Different Source Code Representations and the Hierarchical Structure for Code Summarization.

Existing research regards code summarization as a simple encode-decoder work, and translates the representation of the program code and the comment written in natural language into vector representation to train the network. However, compared with the natural language, the grammatical structure of the programming language is

more complex, and most software engineering projects include many functions which have strict dependency relations. Therefore, applying the neural machine translation to code summarization simply is unreasonable. Most work regard the source code as simple text, while some hidden information may existing for the syntax grammar of the programming language. Therefore, apart from the simple plain source code, I will add the static analysis information, such as, abstract syntax tree and control flow, to excavate the hidden information and to obtain more comprehensive representation of the source code. To grab the hierarchical structure of the source code effectively, the hierarchical attention mechanism is introduced to encode these three representations respectively. The main part of the model is the multi-layer attention mechanism: they are at the token and statement levels respectively which enables the approach to pay attention to the statements and tokens differently when forming the representation for the program code.

To sum up, this work presents the code summarization approach which adopts the hierarchical attention mechanism by utilizing three representations of the source code (the plain code sequence, the sequenced abstract syntax tree and the sequenced control flow). Then, the reinforcement learning framework is adopted to produce source code annotation. When constructing the code representation, this approach pays "attention" to the statements and tokens differently to obtain the hierarchical representation of the source code according to different code features. The DRL enhances the code annotation result.

Research content 2: Statement-wise Semantic Information for Code Summarization.

Although some existing approaches attempt to exploit program semantics through modeling them via abstract syntax tree (AST) [51] or control flows [39], such models fail to establish straightforward statement-to-comment mappings for exploiting statement-wise semantic contributions to facilitate comment generation. Moreover, their adopted deep learning models tend to cause long-distance dependency issues which can seriously compromise the encoding effectiveness of their modeled program semantics [102]. To address such issues, my second work proposes the first transformer-based generative adversarial network framework that integrates statement-wise semantic contributions for universal code summarization. Specifically, it first constructs a cross-language universal hierarchical semantic (UHS) model to splits code blocks within functions/methods upon language-specific code block delimiters, e.g., "{ }" in Java, to classify statements for collecting the underlying method/function-oriented hierarchical semantics, i.e., statement-wise

semantic contributions. Next, by integrating the UHS model with the transformer features, it proposes a tree-transformer-based generator for augmenting its generative effectiveness. At last, the approach is further enhanced by injecting the tree-transformer-based generator into the generative adversarial network.

Research content 3: Statements Selection for Integral Comment Generation

Although existing code summarization techniques show promising results in generating summary for a code snippet [43, 51, 91, 113], the automatically generated summaries by these techniques have some limitations that may reduce their usefulness for developers in code understanding. Firstly, the generated summaries may be *redundant comments* [60] that do not add much value to code understanding. For example, most of the code comment generation techniques only produce one sentence as a description for a given method. Such method summaries may be *incomplete* as they only explain the general intent of the methods without documenting their parameters. A complete comment should not only contain the method comment which shows its intent but also the parameter comment which indicates its usage. Besides, to grab the semantic information of this code snippet, not all statements in the code snippet contribute equally to the comment generation as some statements do not include the words in the comment at all. To generate complete API documentation for Java programs, prior work suggested augmenting method summaries with parameter comments [92]. However, as the parameter comments are only loosely integrated with the corresponding method summaries, the entire generated code comment may not precisely capture the semantics of the method. Besides, the information-retrieval-based approaches are argued to be data dependent, while the deep-neural-network-based approaches tend to be restrained from long-distance dependency, excessive computation cost, and/or inadequate semantic delivery.

To address such issues, in this work, I propose the transformer-based neural network model for the tasks of method comment and parameter comment generation. Specifically, it first designs a programming-analysis-based component to extract usage statements set of parameter and the important statements set in the code snippet to obtain the main semantic information and discard the useless noise information for the dual tasks. Next, the copy-attention-integrated transformer based NMT (Neural Machine Translation) framework is utilized for both the method comment and parameter comment generation to provide complete java documentation for the code snippets.

To evaluate the effectiveness of the proposed approaches, a few of experiments upon

the collected dataset are performed. The results prove that the designed approach can achieve better performance compared with the baseline approaches.

1.3 Thesis Structure

The organizational structure of this thesis is as follows:

Chapter 1: This chapter first introduces the research background and significance of code summarization, then illustrates the research contents and innovations.

Chapter 2: This chapter first demonstrates the literature review of code summarization approaches by categorising them and analysing the mainly advantages and disadvantages of different categories. Then, this chapter describes the techniques related to the research, such as code representation, deep learning.

Chapter 3: This chapter proposes a code summarization approach utilizing the hierarchical attention by merging three code representations. Then to generate the code annotations, the vector deduced from the three code representations is input to the reinforcement learning framework.

Chapter 4: Considering that many existing approaches exploit inadequate power of statement-wise semantic contributions for augmenting their performance, this chapter proposes the first transformer-based generative adversarial network framework that integrates statement-wise semantic contributions for universal code summarization. It mainly constructs a cross-language universal hierarchical semantic (UHS) model to classify statements according to their positions in source code and integrates it with the transformer features to augment its generative effectiveness.

Chapter 5: This chapter proposes a transformer-based neural network model for the dual tasks of method comment and parameter comment generation. Specifically, it first designs a programming-analysis-based component to extract usage statements set of parameter and the important statements set in the code snippet to obtain the main semantic information and discard the useless noise information for the dual tasks. Then the copy-attention-integrated transformer based NMT (Neural Machine Translation) framework is utilized for both the method comment and parameter comment generation to provide complete documentation for the code snippets.

Chapter 6: This chapter summarizes the main work of this thesis, and presents the prospect of future work.

Finally, the acknowledgement and the main research achievements are given.

CHAPTER 2

RELATED WORK

The research on code summary generation has always been. According to the investigation into the existing code summarization approaches, they vary from manually-crafted-template-based approach[91], IR-based approach[71] to the deep-learning-based approach [8, 37, 41]. The template-based approaches usually design a template for certain type of the source code, and then extract keywords from the code to synthesize the corresponding comment. The information-retrieval-based approaches mainly utilize the information retrieval model [44] to obtain the word with high frequency to compose the code intention. The deep-learning-based approaches often regard the programming language as a special language and then the neural machine translation (NMT) model is adopted to produce the code annotation. With the development of deep learning technique, there are more and more research focus on the direction of the deep-learning-based code summarization. The main research of this direction focus on the representation of the source code and the improvement of the neural machine translation model. This chapter will first categorize the existing code summarization approaches in Section 2.1, and then Section 2.2 presents the technique related to the deep-learning-based approach as my research is focus on this direction.

2.1 Literature Review of Code Summarization

In recent years, code summarization has been proposed to generated code annotation automatically to realize the developer's labour and great attention has been paid to this

research topic recently. The existing research of comment generation can be summarized as template based [91], information retrieval based [71] and deep learning based approaches [8, 37, 41]. For the template based approaches, template for each type of function is usually designed and then pad it with the keywords extracted from the source code to obtain the code comment. In the information retrieval based approaches, the information retrieval algorithms [44] are often utilized to obtain the more important word (usually the word with high frequency) in the source code to form the corresponding code annotation. The deep-learning-based approaches often utilize the neural machine translation (NMT) model and take the programming language as a language with specifical syntax/grammar to generate the code comment. This section mainly gives a detailed literature review of existing code summariztion approaches by different categories and analyzes the advantages and disadvantages of them.

2.1.1 Template-based approaches

In the template-based approaches, the researchers often design a template for certain type of source code, and then extract keywords to fill it to generate the code annotation. Sridhara et al. [91] design heuristic rules to select important statements from the code snippet, and use the Software Word Usage Model (SWUM) to identify keywords from those statements and create summaries through manually-crafted templates. Combined with this work, they proposed to generate parameter description and integrate with function comment in [92]. Moreno et al. [68] first determine the stereotypes of the class which is used to select the information and then generates the class summary by following a set of predefined rules. To generate the commit messages, Cortes-Coy propose ChangeScribe [22] which utilize method stereotypes to identify the method responsibilities in a class, and use template-based summarization to generate the commit message. To automatic generate summarization of c++ methods, two steps are performed: identification of method stereotype and template generation for documentation purposed [3]. Wang et al. [104] Utilize high-quality source code projects to train templates for learning the behavior of related objects, and automatically generated a summarization for the related objects with the source code method. To generate the release notes, Moreno et al. propose ARENA [69], which extracts changes from the source code, summarizes them according to the pre-designed templates and integrates them with informatiom from versioning systems and issue trackers. Dawood et al. [23] and Hammad et al. [38] both fill in the predefined template with program structural information such as the number of interfaces in a package or what kind of parameter does a method use. Rai

et al. [83] identify methods' or classes' stereotype based on their property and choose templates accordingly to generate code summary by filling them with messages such as method signature and return type. To generate the abstractive summaries, Badihi et al [10] propose CrowdSummarizer which utilizes crowdsourcing, gamification and natural language processing to learn a set of weights and sentence templates for summaries genenration. Zhai et al. [119] propose CPC which first classifies comments based on different perspectives and code entities based on which each comment is attributed to a code element and becomes a first-class object just like other classic objects in program analysis. Then it leverages program analysis techniques to propagate comments from one code entity to another and to update, infer, and associate comments with code entities.

2.1.2 Information-retrieval-based approaches

The information-retrieval-based approaches mainly utilize special information retrieval algorithm to abstract the key words from the code snippet to generate the intent of the program. Considering the generation of code fragment summaries, Ying [116] proposed a supervised machine learning approach which selects lines contained syntactic features or related to the given query. McBurney et al. [62] represent the software as a call graph and then process it for the topic model to extract the key information to generate the summary. Besides, they want to generate not only the method behavior but also why the method exists or what role it plays [63], they utilize PageRank to explore the importance of the methods and then use SWUM [91] to extract the keywords to generate the summarization. Wang et al. [101] use part-of-speech tagging to identify keywords that best represented the features of the source code. Through noise reduction, a number of keywords with the highest weights are selected to form the code summarizaation. Rigby et al. [85] adopt three feature decision tree classifiers to extract the code elements for StackOverflow Site Discussions. A heuristic-based technique is proposed for mining insightful comments in [82], which utilizes topic model and PageRank algorithm to collect five heuristics and combines them for ranking, then selects the top ranked comments as the insightful comments. Similarly, Latifa et al. generate the code element summary from StackOverflow site discussions by extracting the natural language text from developer's discussions [34]. In [117], latent semantic indexing and hierarchical clustering are utilized to derive artifacts from source code and group source files sharing similar vocabulary to summarize the Java packages. SVM and Naive Bayesian classifiers are adopted in [73] to create code fragment summaries from online FAQs. Fowkes et al. [28] propose TASSAL, in which the Topic Model is utilized for autofolding source code that is

less informative code areas considering on the file specific tokens. Then the summary of the source code is generated based on the file-specific tokens. Considering the summary generation of Java classes, Malhotra et al. [61] propose to tag different micro patterns utilize micro patterns which helps identifying various design structures employed in the class. Wong et al. [108] propose to utilize the code-annotation pairs to produce the description for the code snippet grabbed from the open-source projects. At the same time, they also utilize code clone to look for similar program code [107]. Movshovitz-Attias et al. [71] produce code annotations according to the topic model and n-grams from the Java projects. Haiduc et al. [36] combine different information retrieval models, i.e., VSM (Vector Space Model) and LSI (Latent Semantic Indexing), to deduce the comments of Java classes and functions. The functions are regarded as documents and they search on the resulted documents by LSI and VSM, then they compute the cosine distances among them and the most similar fragments are chosen as the code annotation. Rodeghero et al. [86] improve the process of selecting fragments by eye-tracking and adjust the weights of VSM for better comment generation.

2.1.3 Deep-learning-based approaches

Typically, the programming language are taken as a special natural language and the neural machine machine translation model, which is a encoder-decoder model, is utilized to produce code annotation in the deep learning based comment generation approach. These type of comment generation approaches mainly combine leverage Convolution neural networks (CNNs) or Recurrent Neural Networks (RNNs) with the attention mechanism. For example, RNN with the attention mechanism is proposed to utilized in [43] to produce annotations for C# code snippets and SQL queries. An attentional CNN is adopted to translate the input tokens to detect the topical attention features in long range to sketch program code to short name-like summaries. Comment generation is also taken as machine translation, and the models Seq2Seq [94] and Seq2Seq with attention [11] are employed to realize this task. Based on the abstract syntax tree structure of the source code, Alon et al. propose to utilize the compositional paths to represent the source code and then in the decoding process, the attention mechanism is utilized to select the relevant paths for comment generation [9]. Liu et al. [59] propose Nearest Neighbor Generator (NNGen), in which the `diffs` information of the code is utilized to generate concise commit messages. Hu et al. [42] propose to grabbing the semantic of the source code to produce code annotations with the assistance of API knowledge. In [20], the author propose to utilize C-VAE and L-VAE for source code and

natural language encoding respectively, based on which the semantic representations of both the program code and description can be deduced by the framework and which can help to generate completely annotations for the source code. In addition to such seq2seq based comment generation approaches, the reinforcement learning model is adopted in [100] to ease the exposure bias issue. Hu et al. [41] propose to traverse the abstract syntax tree of the source code and which is input to the NMT model for comment generation. Leclair et al. [51] propose the learn code structure independently from the text in source code by combining words in source code with code structure in the abstract syntax tree. CoaCor [113] adopts the plain sequence of the program code and realizes code summarization by using the seq2seq framework based on LSTM. Consider both NMT-based and retrieval-based approaches, *Rencos* [120] is proposed. Given the program code, *Rencos* first retrieves the dataset to find the most similar code, then the input and the two source code retrieved are encoded, and eventually fuses them during decoding to generate the summary. In [50], a graph-based neural architecture is proposed to better matches the default structure of the AST, together with source code sequence as separate inputs to better matches present a new neural model architecture that utilizes a sequence of source code tokens along with ConvGNNs to encode the AST of a Java method and generate natural language summaries. Considering project-level code summarization, Bansal et al. [13] propose the related source code files, such as the word, subroutine, file, and project embedding as the input the encoder-decoder framework for comment generation. Li et al. propose that code summaries are composed of patternized words and keywords [54], thus propose EDITSUM which retrieve the summary of the similar code snippet as the templates and then generate the comment based on the source code, the similar code snippet and their summaries. Different from these approaches, Moore et al. [67] propose a character-based approach which takes all the characters of the source code as input of the encoder-decoder framework and introduces a vocabulary creation method that allows the model to overcome the long-tailed nature of terms.

Although these code summarization approaches have achieved certain performance, while they still have all kinds of limitations which result in their low accuracy, thus these approaches still have not been widely used in practice. Firstly, thought the template-based approaches and the information-retrieval-based approaches is quite simple and the implementation of them is very easy, while they can only generate comment with limited words provided by the source code. Therefore, they cannot generate comment which need the comprehension of the source code. Besides, the template-based approaches may can only be effective in certain source code type, and the predefined template

or rules may be invalid for other types, which make them with low scalability. The deep-learning-based approaches solve these problems, while the existing approaches still have different disadvantages. Firstly, compared with natural language, program language has stricter semantic model and grammatical structure, but existing work has not considered the intrinsic feature of source code, therefore, the representation of the source is not accurate enough. Secondly, existing work only consider the generation of the source code intent, while ignore the information of parameter etc which is also important for the understanding and usage of the source code. Last but not least, the existing researches mainly adopt the CNN/RNN-based neural machine translation framework, while the CNN/RNN usually cause long-distance dependency and excessive computation cost problem which result in that the encoded source code is not accurate enough and the encoding of the input is time consuming.

2.2 Deep Learning Technique

2.2.1 Deep learning network

Recently, with the development of neural network, it has been widely used for its performance in capturing the hidden information inside the data. For the sequence information process task, RNN (Recurrent Neural Network) is proposed [32], which is recurrent network that keeps the history information of all the previous input sequences to the current state. Fig 2.1 presents the structure of RNN, at the current time t, it takes both the outputted hidden state in previous time step $t-1$ namely s_{t-1} and the current input x_t as inputs and outputs the current state s_t, namely, $s_t = tanh(Ux_t + Ws_{t-1} + b)$, where W, U, and b are the parameters of the network and are updated while training and $tanh$ is the activation function. However, when the input sequence is very long, RNN will cause the long distance dependency problem which is resulted by gradient vanishing and gradient exploding. The gradient vanishing caused by small error gradients accumulate and they shrink exponentially until they vanish and make it impossible for the model to learn, while the gradient exploding is caused by large error gradients and result in very large updates in the norm of the gradient during the back propagation. Therefore, RNN is argued that it is unable to grasp the nonlinear relationship over a long sequence span.

To ease the long distance dependency problem, the LSTM (Long short-term Memory) technology [40] is designed and it realizes the function by adding a gate mechanism to ascertain which tokens' information to be keep in the result. Three gates, namely the

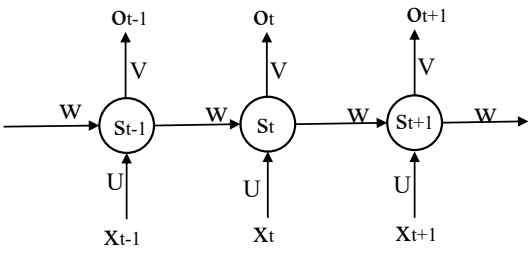


Figure 2.1: The structure of RNN unit.

input gate, forget gate and output gate, compose the gate mechanism which control the information in the input sequence to contain in the final output through weights and activation function. In this way, compared to vanilla RNN, LSTM can handle the long span dependency more effectively, which make it widely utilized to solve semantically related tasks and it has also obtained promising results. However, as the state of step $t-1$ is used in the computing of the current step t, thus the calculation must conduct step by step which may causing the excessive compute time problem.

To solve the above long-distance dependency problem and the excessive computing problem, Transformer [98] is proposed to obtain better representation of the sequence by adopting the self-attention mechanism which calculates the relationships among the tokens in the input sequence to obtain the intermediate representation of the input sequence. There are N identical layers in transformer, and each layer includes two sub-layers. The mechanism of self-attention is realized in the first sub-layer, and the second sub-layer is just a straightforward feed-forward neural network.

2.2.2 Neural machine translation

NMT (Neural Machine Translation) [48] relays on the encoder-decoder framework in which the encoder encodes the input into a hidden space to obtain the vector representation of the input sequence and the decoder decodes the vector to the target language space to generate the output. Fig. 2.2 shows the main framework of NMT. According to the figure we can see that the framework mainly include two parts: encoder and decoder. Given the input tokens $x_1, x_t, ..., x_m$, the encoder, a neural network, encodes the embedded input tokens into hidden space one by one and obtains the hidden states $s_1, s_2, ..., s_m$ which compose the hidden representation of the input sequence. Then, another neural network, namely the decoder, decodes the hidden states into the target space to obtain the target sequence $y_1, y_2, ..., y_n$. Besides, it is often accompanied by the attention mechanism that aligns target with the source tokens [11]. Compared with those

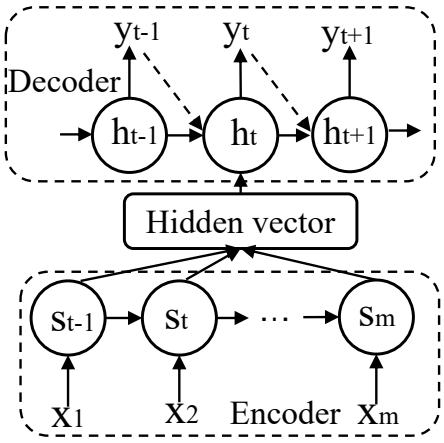


Figure 2.2: The Neural Machine Translation framework.

of phrase-based approaches, NMT has shown impression performance as the previous need hand engineered features.

NMT can effectively bridge the gap between different natural languages. As code summarization is also a variant of machine translation problem in which the code snippet are written in special programming language, thus we can utilize the neural machine translation framework to tackle the code summarization task and the research focus on the representation of the source code as the programming languages have stricter semantic/syntax structure compared to the natural language.

2.3 Conclusion

This chapter mainly gives a literature review of code summarization by classifying them into three categories: template-based code summarization, information-retrieval-based code summarization and deep-learning-based code summarization and then analyzes the advantages and disadvantages of different categories. Besides, the related techniques utilized in the research such as deep learning, neural machine translation are also described simply in this chapter.

CHAPTER 3

THE COMMENT GENERATION BASED ON DRL AND HIERARCHICAL ATTENTION

Code summariztion provides the main aim of the source code described in natural language, which can benefit the development and maintenance of software. As far as I know, the existing deep learning based comment generation approaches mainly utilize the seq2seq model in which the program code is encoded into hidden space first and then decode it to produce the target comment. However, these kind of approaches have the following drawbacks: (1) they mainly take the source code as plain text and ignore the hierarchical structure of the source code; (2) most of the approaches only consider simple features, such as, tokens, which overlooking the hidden information that can help grab the relationships between source code and comments; (3) they typically train the decoder to produce the code annotation by calculating and maximizing the odds based on the subsequent natural language words, however in fact, they mainly produce the code annotation from scratch. Therefore, these drawbacks result in inferior comment generation accuracy and inconsistent of the generated comment.

To solve the limitations described above, this chapter proposes to utilize the hierarchical attention mechanism to combine several source code features for code summarization. The representation of these features are input to the deep reinforcement learning framework for the comment generation task. To present the hierarchical structure of the source code under different contexts considering the code features, the proposed approach allocate weights to different statements and tokens respectively when forming

the representation of the source code. Furthermore, the reinforcement learning model improves the generated comment through the actor and critic network, in which given the present state the actor furnishes the confidence of generating the next words, and then the critic network calculates the reward values of the potential next word to provide clues for the generation explorations. At last, I train the framework based on the reward and perform experiments bases on the dataset collected from real projects in github.

This chapter is organized as follows. The introduction of this work is given in Section 3.1. The background preliminary is presented in Section 3.2. Section 3.3 demonstrates the detailed information of the proposed approach. The experimental results and analysis are elaborated in Section 3.4. Section 3.5 presents the threats to validity. At last, Section 3.6 concludes the main contribution of this work.

3.1 Introduction

In recent years, the research on automatic code summarization has made great progress to generate natural language descriptions of software. According to my observation, statistical language models is the mostly adopted approach to learn the semantic representation of the source code in the most of the existing comment generation approaches [71, 76], and then the templates/rules [91] or information retrieval models [107, 108] are utilized to generate comments. Nowadays, with the research progress of deep learning, researchers propose to introduced the neural machine translation models for the comment generation task [8, 37, 43], in which the encoder-decoder framework is utilized. In these approaches, two recurrent neural networks (RNNs) [88] are included, the first RNN encodes the input source code into a hidden space and another RNN decodes the result to generate the targeted comment. According to the generated words and the ground truth, the likelihood of the next words is typically maximized by these models.

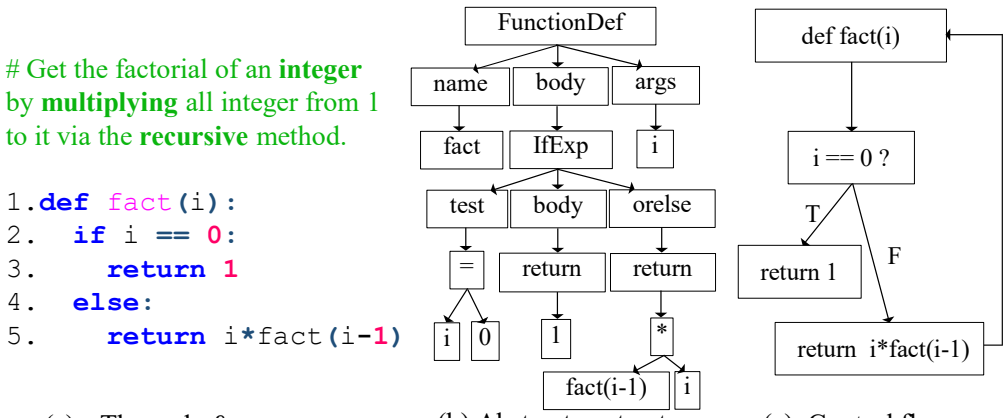
According to my investigation, there are three shortcomings in the most existing code summarization approaches: (1) The input source code are regarded as plain sequences which are composed of tokens, such as character, symbol etc. directly which has ignored the hierarchical structure of the source code (for example, the tokens making up the statements and the function is consisted of different statements) by which more comprehensive information can be provided for the generation of comment. (2) Most of these approaches [8, 42, 71] only consider the token sequence to obtain the representation of the source code, while some hidden code information contained in the other features, such as Abstract Syntax Tree and control flow graphs which can grab the relationships

between the code snippet and comments are not explored well. (3) The “teacher-forcing” model adopted by the training of most existing approaches has the exposure bias problem which caused by the fact that the ground truth is unreachable during the testing state and the next work is produced according to the words generated previously [84], therefore the model is trained according to the sequence in the ground truth [106].

In this work, I aim to address these three mentioned issues. To effectively capture the hierarchical structure information, I adopt the hierarchical attention mechanism to encode these three representations respectively. As shown in 3.4, the attention network includes two layer compose the main model, and the two layer attentions are at the token and statement level respectively, and the details will be described in section 3.3.2. This hierarchical attention mechanism allow the model to allocate different attention to each token and statement differently when building the code snippet representation. Finally, a hybrid attention layer is adopted to combine these three code representations for further process to generate the target comment.

To represent the the structural (or syntactic) information of source code, three comprehensive representations of the source code are utilized. The first one takes the source code as plain text which is just as most existing research do. The second one sequences the abstract syntax tree (AST), which is a data structure that is widely utilized in compilers, to a sequence by traversing the tree to represent the structure of program code. The third representation of the source code is the sequenced control flow graph of the code snippet which is also quite pivotal for the comprehension of the source code. The three representations of an Python source code example is shown in Figure 3.1 (a-c), the function of this code snippet is to obtain the factorial of an integer through the recursive method. The source code snippet and the corresponding comment is presented in Figure 3.1 (a), the plain text representation read the code line by line to obtain the sequence. The abstract syntax tree is shown in Figure 3.1 (b), and its sequence representation is obtained based on the module of ast for Python. The control flow of the code snippet is demonstrated in Figure 3.1 (c). The control flow can represent the execution order of the source code. For example, when we get the factorial of integer 3, the code execute “else: return i*fact(i-1)” in lines 4&5 instead of “if i ==0: return 1” in lines 2&3 when we want to get the factorial of integer 1. Three LSTM models are utilized to encode the three sequential information of code: plain source code sequence, sequenced abstract syntax tree, and sequenced control flow.

To surmount the exposure bias problem, I introduce the deep reinforcement learning framework, in which the exploration and exploitation are all included in one framework.



(a) The code & summary (b) Abstract syntax tree (c) Control flow

Figure 3.1: The example of different representations of the source code.

Rather than looking for the subsequent correct word greedily by learning a sequential recurrent model, I adopt the reinforcement learning (actor-critic network) to define the best next word at every step. The actor network plays the part of local guidance to supply the probability of generating the next word to form the comment. While the critic network acts as the global guidance to calculate the award value of the potential words based on the present state. Through these mechanism, the better words with lower possibility can also be included for generating comment. To train the framework more efficiently, the standard supervised learning with cross entropy loss is adopted to train the actor network, and then mean square loss is utilized to train the critic network. At last, based on the BLEU value the framework is trained simultaneously via policy gradient.

The framework of the proposed code summarization approach is illustrated in Figure 3.2, which is mainly based on the two-layer attention network. The dataset which is consisted of a corpus of $< code, comment >$ pairs is prepared first. In specific, I first represent the source code consider both the structural and the unstructured information by three sequences: x^{TXT}, x^{AST} and x^{CFG}, as shown in Figure 3.2(a); then, these three representations of the source code are encoded by the hierarchical attention network and then integrate them. Finally, the obtained information is injected into the proposed deep reinforcement learning framework for further training, as shown in Figure 3.2(c). At last, given the trained actor network and a source code whose comment we want to generate, we can just obtain the target comment by injecting the code snippet to the trained network.

The contributions in this work can be mainly summarized as follows:

- To represent the granularity information of the source code, a LSTM-based hierar-

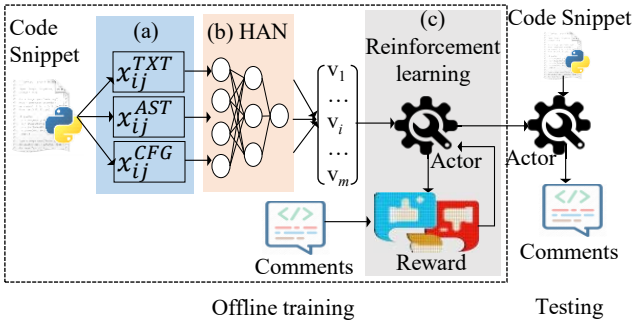


Figure 3.2: The overview framework of the proposed code summarization approach.

chical attention mechanism is introduced, and two layers are included which are at the token level and the statement level respectively. This mechanism enables the approach to pay different attention to statements and tokens differentially when structuring the representation of the source code.

- three comprehensive representations of the input code snippet: plain source code sequence, the sequenced abstract syntax tree and the sequenced control flow are utilized.

- To estimate the performance of the proposed approach, a few of experiments are performed based on a dataset collected from real projects in github which contains about 108,000 Python source code. The results indicate the effectiveness of the proposed approach comparing to the existing approaches.

3.2 Preliminaries

This section describes some background techniques that adopted in my work, for example, language model, reinforcement learning and so on. Assume that the sequence of one function is signed as $\mathbf{x} = (x_1, x_2, \ldots, x_t, \ldots, x_{|\mathbf{x}|})$, where x_t means the tokens in the source code, such as, "*def*", "*fact*", and "*i*" in the Python statement "*def fact(i):*". The list of sequence for the generated comments is presented as $\mathbf{y} = (y_1, y_2, \ldots, y_{|\mathbf{y}|})$, where $|\mathbf{y}|$ is the tokens number in the sequence. The max number decoding step in the framework is signed as T. $y_{l\ldots m}$ denotes $y_l, \ldots, y_m$ is the comment subsequence and $\mathscr{D} = \{(\mathbf{x}_N, \mathbf{y}_N)\}$ denotes the dataset, and N means the training set size.

3.2.1 Language model

Given a certain sequence, the language model calculate the occurrence probability of the words sequence [31]. Assume a sequence containing T words $\mathbf{y}_{1...T}$, let $p(y_{1:T})$ represent its occurrence probability, which is calculated according to the conditional probability by the sequence of n words before, aka n-gram [103], and the calculation is shown in Equation 3.1.

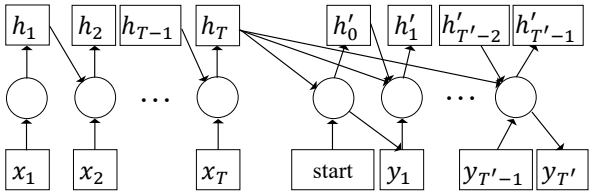
$$p(y_{1:T}) = \prod_{i=1}^{i=T} p(y_i|y_{1:i-1}) \approx \prod_{i=1}^{i=T} p(y_i|y_{i-(n-1):i-1}) \tag{3.1}$$

This kind of n-gram approach has obvious problems [66, 87]. For instance, the n-gram model counts the frequency of different fragments in the sequence which results in poor performance when the sequence contains many words that have not appeared before.

As only a certain number of predecessor words are utilized to predict the current word in the n-gram model which is quite limited, the neural language model enable all the generated words to be referred to predict the present word. The neural network mainly contains three layers, namely, the input layer takes the input sequence x_t and generates the corresponding vector, the recurrent hidden layer calculates and renews the present hidden state h_t by adding the input token x_t, and based on the present hidden state, the output layer calculates the odds of the next word. According to this structure, the network takes the tokens in the input sequence successively, and then produce the potential word one by one. $p(y_{t+1}|y_{1:t})$, namely the odds of the next word, is calculated according to the following steps: (1) firstly, the the input layer maps the present input token y_t to a vector; (2) then, based on the previous hidden state $\mathbf{h}_{t-1}$ and the present input x_t, the hidden state $\mathbf{h}_t$ is calculated according to $\mathbf{h}_t = f(\mathbf{h}_{t-1}, w(x_t))$, where w means the parameters of the networks; (3) the probability of the comment $p(y_{t+1}|y_{1:t})$ is calculated based on the present hidden state $\mathbf{h}_t$: $p(y_{t+1}|y_{1:t}) = g(\mathbf{h}_t)$, and g is the stochastic output layer by which the target words can be generated.

3.2.2 RNN encoder-decoder model

Figure 3.3 presents the framework of encoder-decoder based on Recurrent Neural Network (RNN), in which two recurrent neural networks are included. Given the input source code $\mathbf{x}$, the encoder transforms its sequence into the hidden states $(\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{|\mathbf{x}|})$, while the decoder decodes these results to obtain the target comment by generating one word y_t in each time step.



(a) The architecture of encoder in RNN

(b) The architecture of decoder in RNN

Figure 3.3: The framework of encoder-decoder based on RNN.

3.2.2.1 Encoder

As shown in Figure 3.3 (a), the hidden state of the encoder is a fixed-dimension vector. At time step t, $\mathbf{h}_t$, namely the hidden state, is calculated according to Equation 3.2.

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, w(\mathbf{x}_t)) \tag{3.2}$$

where $\mathbf{h}_{t-1}$ means the state in time step $t-1$, $\mathbf{x}_t$ represents the token input at step t, and the hidden layer is signed as f. The last input token $< eos >$, namely the symbol indicating the end of the input sequence to the encoder and generate the hidden state $\mathbf{h}_T$, which is the final vector representation of the input sequence $\mathbf{x}_{1:T}$. However, RNN suffers from two problems: gradient vanishing and gradient exploding when the input sequence is very long. The gradient vanishing caused by small error gradients accumulate and they shrink exponentially until they vanish and make it impossible for the model to learn, while the gradient exploding is caused by large error gradients and result in very large updates in the gradient during the back propagation. To solve this issue, researchers propose long short-term Memory (LSTM) [40] which tries to ascertain the information accumulation by a gate mechanism, where three gates, namely the input gate, forget gate and output gate, compose the gate mechanism which control the information in the input sequence to contain in the final output through weights and activation function.

At a certain time step t, the hidden state is calculated according to the Equation 3.3:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}^{(i)}\mathbf{x}_t + \mathbf{U}^{(i)}\mathbf{h}_{t-1} + \mathbf{b}^{(i)}) \\
\mathbf{f}_t &= \sigma(\mathbf{W}^{(f)}\mathbf{x}_t + \mathbf{U}^{(f)}\mathbf{h}_{t-1} + \mathbf{b}^{(f)}) \\
\mathbf{a}_t &= \tanh(\mathbf{W}^{(a)}\mathbf{x}_t + \mathbf{U}^{(a)}\mathbf{h}_{t-1} + \mathbf{b}^{(a)}) \\
\mathbf{c}_t &= \mathbf{i}_t \odot \mathbf{a}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1} \\
\mathbf{o}_t &= \sigma(\mathbf{W}^{(o)}\mathbf{x}_t + \mathbf{U}^{(o)}\mathbf{h}_{t-1} + \mathbf{b}^{(o)}) \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned} \tag{3.3}$$

where $\mathbf{i}_t$, $\mathbf{f}_t$, and $\mathbf{o}_t$ represent the input gate, forget gate, and output gate respectively for the update of memory cell c_t. The weight matrices are signed as $\mathbf{W}^{(\cdot)}$ and $\mathbf{U}^{(\cdot)}$, and $\mathbf{b}^{(\cdot)}$ means the bias vector, and $\mathbf{x}_t$ is the vector representation of the tth input node. $\odot$ means the operation of element-wise multiplication between vectors and $\sigma(\cdot)$ denotes the logistic function.

3.2.2.2 Decoder

The structure of RNN-based decoder is shown in Figure 3.3 (b), and the output compose the demanded sequence, namely $\mathbf{y} = (y_1, \cdots, y_{T'})$. Firstly, it is initialized by the start symbol of the input sequence which indicates the beginning of the aimed sequence. At the certain time t, given the present hidden state $\mathbf{h}_t$, the conditional distribution of the next token y_{t+1} is calculated according to $p(y_{t+1}|y_t) = g(\mathbf{h}_t)$.

3.2.2.3 Training Goal

During the training process, the encoder and decoder are trained simultaneously to optimize the below objective,

$$\max_{\theta} \mathscr{L}(\theta) = \max_{\theta} \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim\mathscr{D}} \log p(\mathbf{y}|\mathbf{x};\theta) \tag{3.4}$$

where θ represents the parameter set in the model.

3.2.3 Reinforcement learning

Reinforcement learning [97] enables the framework to obtain the optimal policy from the reward signal by interacting with the true environment, by which the exposure bias problem can be potentially solved. In specific, during the generation, the next word are

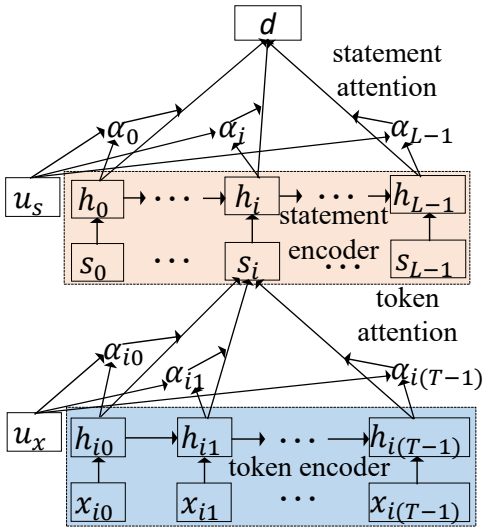


Figure 3.4: The hierarchical attention network.

produced by the RNN model iteratively according to all the produced tokens previously that may have not been occurred in the training data [118]. While when the approach adopts reinforcement learning, it computes the reward value by interacting with the environment to supply guidance to train the model for solving this problem. As presented in Equation 4.7, the target of the sequence generation task is to obtain a policy that can optimize the reward of the generated sequence sampled from the model's policy.

$$\max_{\theta} \mathscr{L}(\theta) = \max_{\theta} \mathbb{E}_{\substack{\mathbf{x}\sim\mathscr{D} \\ \hat{\mathbf{y}}\sim P_\theta(\cdot|\mathbf{x})}} [R(\hat{\mathbf{y}}, \mathbf{x})] \tag{3.5}$$

here θ denotes the parameter set of policy, $\mathscr{D}$ represents the set to be trained, $\hat{\mathbf{y}}$ means actions/words that target to be generated, and the reward function is signed as R. Therefore, the comment generation problem can be formulated as:

- Assume that the sequence $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathbf{x}|})$ represents the program code, I aim to find a policy based on which the target comment sequence $\mathbf{y} = (y_1, y_2, \ldots, y_{|\mathbf{y}|})$ can be generated.

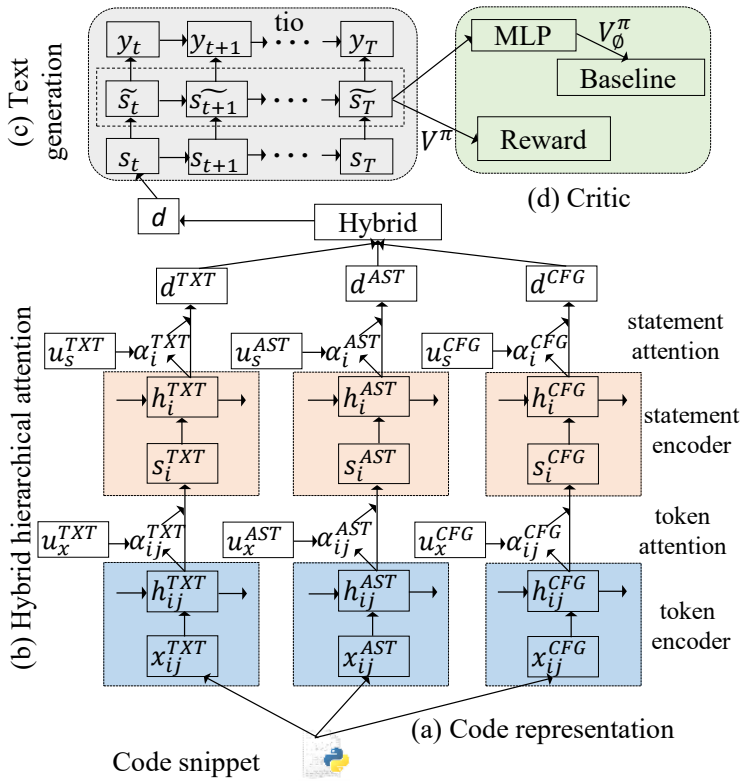


Figure 3.5: The overview of the proposed code summarization via hierarchical attention mechanism: (a) the three representations of source code are signed as x_{ij}^{TXT}, x_{ij}^{AST} and x_{ij}^{CFG} where i and j represent the jth token in the ith statement; (b) d_t^{TXT}, d_t^{AST} and d_t^{CFG} denote the vectors encoded of the three code representations by the hierarchical attention mechanism respectively; (c) the annotation of the source code is produced by the LSTM-based decoder; (d) Given the state s_t, the critic network estimates its value according to $|baseline - reward|$.

3.3 The Comment Generation Based on DRL and Hierarchical Attention

Figure 3.5 presents the overview framework of the proposed comment generation approach through the hierarchical attention mechanism, which mainly follows the actor-critic framework [49]. In specific, four submodules are included in the proposed framework, namely (a) the representation module of the source code which is used to explain the structural and unstructured syntax of the code snippet; (b) the module of hierarchical attention is utilized to translate the representations of the source code into vectors; (c) the LSTM-based text generation module which used to generate the next word according to the words before; and (d) the critic module is adopted to estimate the accuracy of the

generated annotation and supply feedback the the above modules.

3.3.1 Representations of the source code

Considering the characteristic of the source code, I adopt a set of symbols, i.e., {. , " ’ _ ( ) { } : ! - (space) } to tokenize and split the code into different identifiers and then translate them to lowercase letter. Next, the gotten words are embedded into vectors according to the module *genism* in Python. The tokens have not appeared are regarded as unknown words which is similar to [9, 43, 113].

In this chapter, I utilize the following three representations of source code: Plain text sequence, the sequenced abstract syntax tree, and the sequenced control flow graph.

3.3.1.1 Plain Text representation

Consider that the comments are usually generated by choosing the lexical tokens of the program code, for example the function name, operator name etc., thus one main representation of the source code is the plain text.

3.3.1.2 structural representation

When executing the program, the compiler transforms the source code into intermediate code by constructing the abstract syntax tree [6] and then translating to control flow graph which reflects the vital information of the source code. Therefore, the abstract syntax tree and control flow graph are also adopted as the structural representations of the source code. Based on the *ast* module [1] in Python, the sequenced abstract syntax tree can be obtained. For the control flow graph representation of the source code, each node includes a sequence of tokens which composes the statement and each edge which connects two nodes represents the flow of statements in the source code. Following the *ast* module [1] and [72], the control flow graph can be obtained and then it is transformed to get the control flow sequence in depth-first order.

3.3.2 Hybrid hierarchical attention network

Different part of the source code contributes differently to the final output of the comment. In specific, in different code snippet, the same token or statement is differentially important. Besides, the source code has the hierarchical structure essentially, for example statements are consisted of tokens and the functions are consisted of statements

respectively. Thus, I introduce the hierarchical attention mechanism [112], which is proposed in natural language processing, to enable the proposed comment generation approach to pay attention differentially to different statements and tokens respectively when forming the representations of the source code.

As shown in Figure 3.5 (b), a two-layer attention network are utilized in the proposed approach, they are ate the token layer and the statement layer respectively. The utilized network is consisted of four parts: the token encoder, the token-level attention, the statement encoder, and the statement-level attention. Assumed that the vectors of the the three code representations are represented as d^{TXT}, d^{AST}, and d^{CFG} respectively. Then, they are integrated into one vector d to obtain the final representation of the source code. The detailed information of this network are described as follows.

The Token Level encoding. Assuming that a sequence of tokens $x_{i0},...,x_{iT_i-1}$ consists a statement s_i, and T_i means the length of the tokens in the statement sequence. Firstly, the tokens are embedded into vectors by the embedding matrix W_i, namely, $v_{it} = W_i x_{it}$. Then, as shown in Equation 3.6, the LSTM model is utilized to encode the tokens from x_{i0} to x_{iT_i-1} in the statement.

$$\begin{aligned} \mathbf{v}_{it} &= \mathbf{W}_i \mathbf{x}_{it}, t \in [0, T_i) \\ \mathbf{h}_{it} &= lstm(\mathbf{v}_{it}), t \in [0, T_i) \end{aligned} \tag{3.6}$$

Consider that different tokens in the statement supply different semantic information and contribute differently to the generated comment. For example, in Figure 3.6 which shows the representation of the source code in the editor, as the words “numbers” and “string” are contained in the given comment, thus the tokens “number” and “str” can supply more information than token “def” in statement “def check_number_exist(str):” essentially. Therefore, the attention mechanism is adopted by the proposed approach to collect the tokens that are more important for the generating of the comment and the extracted words are merged to generate the representation of the corresponding statement as shown in Equation 3.7.

$$\begin{aligned} \mathbf{u}_{it} &= tanh(\mathbf{W}_x \mathbf{h}_{it} + \mathbf{b}_x) \\ \alpha_{it} &= \frac{exp(\mathbf{u}_{it}^T \mathbf{u}_x)}{\Sigma_T exp(\mathbf{u}_{it}^T \mathbf{u}_x)} \\ \mathbf{s}_i &= \Sigma_T \alpha_{it} \mathbf{h}_{it} \end{aligned} \tag{3.7}$$

```
# Check if there are numbers in a string.

def check_number_exist(str):
    has_number = False
    for c in str:
        if c.isnumeric():
            has_number = True
            break
    return has_number
```

Figure 3.6: The source code example that tokens supply information differently for comment generation.

W_x represents the weight matrix, b_x denotes the bias vector, the attention from token x_{it} to the statement s_i is signed as α_{it}, and u_x means the sequence vector in the token level. Particularly, u_x is initialized randomly and optimized during the training process gradually.

The Statement Level Encoding. As the statement vector s_i has been obtained, similar to the encoding of the tokens, the function vector can be generated. Firstly, the statements are encoded by LSTM according to Equation 3.8.

$$\mathbf{h}_i = lstm(\mathbf{s}_i), i \in [0, L) \tag{3.8}$$

where L means the statements number contained in the code snippet.

To extract the statements which contain more important information to the corresponding code snippet for the comment generation task, the attention mechanism is adopted again to generate the function vector u_s in the statement level which enables the framework to estimate the importance of different statements.

$$\begin{aligned} \mathbf{u}_i &= tanh(\mathbf{W}_s \mathbf{h}_i + \mathbf{b}_s) \\ \alpha_i &= \frac{exp(\mathbf{u}_i^T \mathbf{u}_s)}{\Sigma_L exp(\mathbf{u}_i^T \mathbf{u}_s)} \\ \mathbf{d}^c &= \Sigma_L \alpha_i \mathbf{h}_i \end{aligned} \tag{3.9}$$

Where, the weight matrix is signed as W_s, b_s represents the bias vector, and the attention from each statement s_i to the final representation vector $\mathbf{d}^c$ is signed α_i.

As the vectors of the three representations of the source code, namely the vector of the plain text sequence, the vector of the sequenced abstract syntax tree and the vector of the sequenced control flow graph have been generated, they are concatenated firstly and then are feed into the linear network: $\mathbf{d} = \mathbf{W_d}[\mathbf{d}^{TXT};\mathbf{d}^{AST};\mathbf{d}^{CFG}] + \mathbf{b_d}$, where $\mathbf{d}$ denotes the final representation of the source code, $[\mathbf{d}^{TXT};\mathbf{d}^{AST};\mathbf{d}^{CFG}]$ represents the concatenation of the three representations of the source code. At last, an additional hidden layer is utilized for the comment generation: $\widetilde{\mathbf{s}_t} = \tanh(\mathbf{W}_c\mathbf{s}_t + \mathbf{b}_d)$, where s_t is hidden state and s_0 is initialled as $\mathbf{d}$.

3.3.3 Text generation

As the representation of the source code has been deduced, then then comment can be generated by a softmax function. Assume that the policy π defined from the actor network is signed as p_π, and the distribution of the probability of the tth word y_t is signed as $p_\pi(y_t|\mathbf{s}_t)$, we can get the comment generation equation:

$$p_\pi(y_t|\mathbf{s}_t) = softmax(\mathbf{W}_s\widetilde{\mathbf{s}_t} + \mathbf{b}_s) \tag{3.10}$$

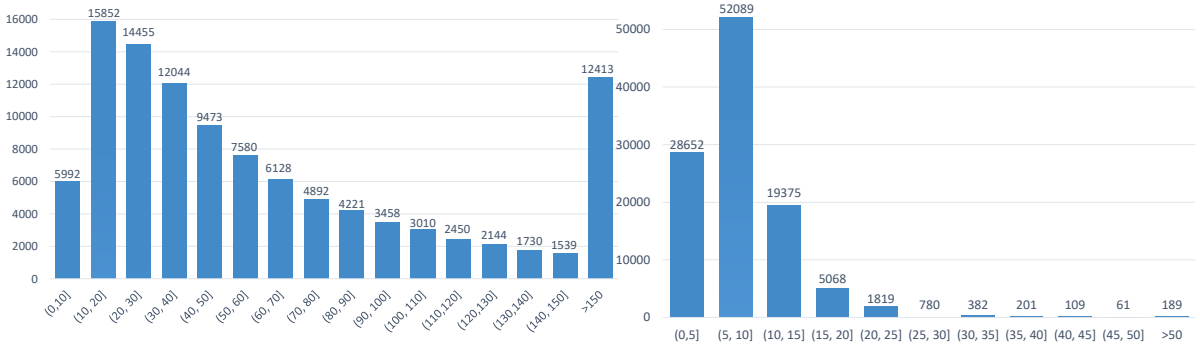
3.3.4 Critic network

Unlike the traditional code summarization approaches which generate comments by optimizing the probability of next words based on the ground truth, while the comments are produced by optimizing the reward iteratively in the proposed approach which is realized by reinforcement learning. In specific, the critic network is introduced to calculate the reward of the summary action to supply a feedback to train the network iteratively.

3.4 Experiments and Analysis

To estimate the performance of the proposed approach, sufficient experiments based on real-world dataset is performed. The following research questions are designed for the estimation in the experiment.

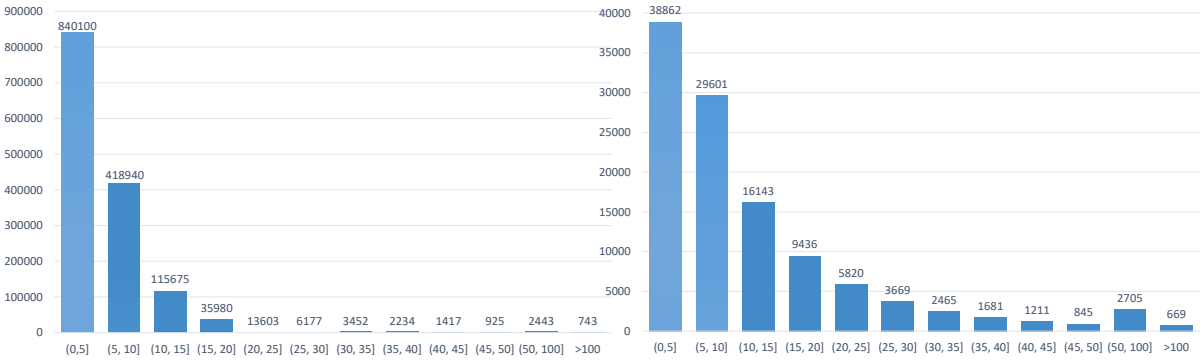
- **RQ1.** What is the performance of different components in the proposed model? For instance, whether the representations of the source code, the hierarchical attention network and the reinforcement learning can improve the performance respectively?



(a) Distribution of code length.

(b) Distribution of comment length.

Figure 3.7: The distribution of the code and comment length in the data.



(a) Distribution of token number.

(b) Distribution of statement number.

Figure 3.8: The statistic analyses of the program code.

- **RQ2.** What is the performance of the proposed code summarization approaches considering different code or comment length?
- **RQ3.** What is the time consumption of different parts?

The research question 1 aims to estimate the performance of each component in the proposed approach compared with the state-of-the-art baselines. The research question 2 aims to estimate the proposed approach consider the range of the code and comment length. The research question 3 is proposed to estimate the time complexity of the proposed approach.

3.4.1 Dataset preparation

To estimate the performance of the proposed approach, the dataset collected in [14], which is collected from GitHub [2] which is the most popular open source projects hosting

platform, is utilized and it is a commonly used dataset these years. There are mainly about 108,700 <code, comment> pairs included in the dataset, and about 50,400 code tokens and 31,300 comment tokens are contained respectively in the datset. The dataset is split into different part in a random way, the first part contains 60% for training, the second part contains 20% for validation and the last 20% is used for testing. Firstly, to learn the characters the adopted source code and comments, statistics analysis is performed and the results are shown in Figures 3.7 and 3.8. Figure 3.7 (a) and (b) present the distributions of the code and comment length respectively. According to Figure 3.7 (a), we can observe that the lengths of the source code range from 10 to 80. Figure 3.7 (b) shows that the tokens in each comment ranges from 5 to 40. This result suggests that the comment target to generate is not too long. Besides, Figure 3.8 (a) and (b) present the distributions of the number of tokens and statements in each source code respectively. The results show that the lengths of the statement are located between 1 to 15 tokens and the lengths of the function are between 2 and 25.

3.4.2 Evaluation metrics

In this work, similar as [8, 100, 113**?**], I estimate the accuracy of the produced annotations from the aspect of their similarity with the corresponding ground-truth comments. In specific, the three widely-used evaluation metrics adopted in NLP area particularly in the the NMT task are utilized: BLEU [77], METEOR [12], and ROUGE [58]. These metrics mainly calculate the similarity degree between the generated natural language text and the ground-truth by measuring the frequency of the tokens occurrence in both of them from different aspects. As comment generation is also a text generation task in which the natural language words composed the out put, thus they are adopted to assess the accuracy of the produced annotations.

Particularly, BLEU is the most common evaluate metric adopted in the natural language generation task [46, 55, 74, 78–80, 90] which estimates the n-gram accuracy by comparing with a few reference sentences. It is computed according to the following equation:

$$BLEU-N = BP * exp(\sum_{n=1}^{N} \frac{1}{N} * logp_n)$$
$$BP = \begin{cases} 1, & len(c) > len(r) \\ e^{(1-len(r)/len(c))}, & len(c) \leq len(r) \end{cases} \tag{3.11}$$

where $p_n = \frac{\sum_{n-gramec} count_{clip}(n-gram)}{\sum_{n-gram'er} count(n-gram')}$, c denotes the generated comment, r denotes the ground truth, and N denotes the gram number of the associated phrase. It first computes the n-gram match level and sums the clipped n-gram counts to prevent the repeated calculation of the multiple n-gram occurrences. Next, it divides the collected clipped count by the number of reference n-grams to compute a modified precision score p_n. At last, it derives and multiplies an brevity penalty factor BP to prevent the length penalization when the generated comment length exceeds the ground-truth comment length. Note that I adopt a BLEU-N family, i.e., BLEU1 to BLEU4, to approach convincing evaluations in terms of the BLEU metrics.

METEOR, on the other hand, considers the stemming and synonymy matching and estimates the extent to which the results grab the content based on the references. In particular, recall is computed as the proportion of the matched n-grams out of the total number of n-grams in the reference comment to reflect to what degree the translation covers the entire content of the translated sentence. Accordingly, METEOR is derived upon a combination of unigram-precision, unigram-recall, and a measure of fragmentation that is designed to directly capture how well-ordered the matched words in the machine translation are compared to the reference comment. It is computed as:

$$METEOR = (1 - Pen)F_{mean}, \tag{3.12}$$

where $Pen = \gamma(\frac{ch}{m})^\theta$, $F_{mean} = \frac{P_m R_m}{\alpha P_m + (1-\alpha)R_m}$, γ, ch means the token number, m represents the matched tokens number, P_m denotes the unigram precision which is the ratio of the number of matched unigrams to the total unigram number, and R_m denotes the unigram recall which is calculated as as the ratio of the matched unigram number in the system translation to the total unigram number in the reference translation.

ROUGE-L calculates the similarity in the sentence-level structure and recognizes the longest co-occurrence consider the sequential n-grams. It is calculated as:

$$\begin{aligned} ROUGE - L &= \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \\ R_{lcs} &= \frac{LCS(X,Y)}{m} \\ P_{lcs} &= \frac{LCS(X,Y)}{n} \end{aligned} \tag{3.13}$$

where R_{lcs} and P_{lcs} are the recall rate and accuracy rate respectively, and $LCS(X,Y)$ is the length of the longest common subsequence, m and n represent the length of the reference sentence and the generated sentence respectively.

Table 3.1: Performance of different code representations.

Approaches	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**METEOR**	**ROUGE-L**
TXT	19.51	2.45	0.95	0.65	5.65	31.56
AST	18.97	3.95	1.87	0.89	5.97	31.23
CFG	19.20	2.45	1.12	0.67	5.12	31.46
TXT&AST	26.56	3.96	1.89	1.32	6.21	37.68
TXT&CFG	27.66	4.25	1.97	1.12	6.38	38.24
AST&CFG	26.35	2.65	0.96	0.97	5.87	38.13
All	**33.16**	**12.39**	6.21	5.10	**9.43**	**46.23**

Table 3.2: Performance of the hierarchical attention network.

Attention type	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**METEOR**	**ROUGE-L**
no atten	18.76	8.21	4.98	3.46	8.24	35.28
1-layer atten	25.79	8.45	5.73	4.67	8.79	38.49
2-layer atten	**33.16**	**12.39**	**6.21**	**5.10**	**9.43**	**46.23**

Table 3.3: Performance of deep reinforcement learning.

Approach	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**METEOR**	**ROUGE-L**
No DRL	26.89	7.21	3.76	2.31	8.21	35.78
With DRL	**33.16**	**12.39**	**6.21**	**5.10**	**9.43**	**46.23**

3.4.3 Experimental settings

In the experiment, I set vector size of the hidden layers of LSTM to be 512 for both encoder and decoder. The mini-batch size is initialized to 32, and the learning rate is initialized to 0.001. Firstly, the the actor and critic network are pretrained by 10 epochs respectively, and then the whole framework is trained by 10 epochs simultaneously. Python 3.6 is All the experiments are implemented based on Python 3.6.

3.4.4 RQ1: The performance analysis of different components

3.4.4.1 The performance considering different code representations

I estimate the proposed approach by different settings considering different components. The three different code representations are signed as **TXT**, **AST** and **CFG** respectively. The hierarchical attention mechanism is denoted as **HAN** and **DRL** refers to deep reinforcement learning.

- **TXT+HAN+DRL.** This baseline regards the source code as plain text and which is encoded by the LSTM-based hierarchical attention network and then train the model by DRL.
- **AST+HAN+DRL.** This approach takes the sequenced abstract syntax tree as the representation of the source code and then encodes it by the LSTM-based hierarchical attention network.
- **CFG+HAN+DRL.** This approach utilizes the sequenced control flow as the source code representation and then use the same framework.
- **TXT&AST+HAN+DRL.** In this approach, the plain text sequence and the sequenced abstract syntax tree as the representations of the source code and the framework concatenates their encoded vectors to obtain the hybrid code representation for the comment generation.
- **TXT&CFG+HAN+DRL.** As the same, the plain text sequence and the sequenced control flow are adopted as the representations of the source code and then follows the same framework.
- **AST&CFG+HAN+DRL.** In this approach, the sequenced abstract syntax tree and the sequenced control flow are utilized as the representations of the source code and then follows the same framework.
- **The proposed approach: TXT&AST&CFG+HAN+DRL.** This is the proposed approach in which the plain text sequence, the sequenced abstract syntax and the sequenced control flow are adopted as the representations of the source code. Then, the LSTM-based hierarchical network is utilized to encode them into vectors and then a hybrid layer is utilized to concatenated them into vector.

The experimental results compared between the proposed approach and the baselines described above is shown in Table 3.1. From the results we can see that the proposed approach can outperform almost all baselines considering most estimation metrics. In specific, the accuracy of generated comment by the proposed approach can outperform the baselines which use the plain text, the sequenced abstract syntax tree or the sequenced

control flow by 29.46% to 31.42% in terms of BLEU-1. Besides, the propose approach improves the generated comments accuracy by 16.58% to 20.53% in terms of BLEU-1 compared with approaches in which two source code representations are adopted. Furthermore, similar result trends are obtained considering the other estimation metrics. To sum up, better performance can be performed by the proposed approach according to the finer-grained code representations for code summarization.

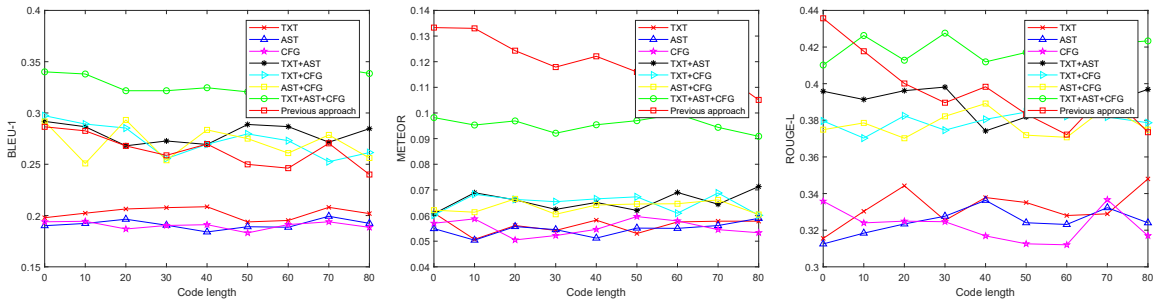
3.4.4.2 The performance of hierarchical attention mechanism

To estimate the performance of the hierarchical attention mechanism, I design the baselines by encoding the code representations with no attention, with 1-layer attention and with 2-layer attention respectively. In the no attention baseline, the code representations are encoded by normal LSTM without attention. While the code representations of the source code are encoded from tokens to function directly in the 1-layer attention baseline. In the proposed approach, the hierarchical structure of the code representations is adopted and the 2-layer attention mechanism is utilized.

The results is presented in Table 3.2, from which wen can observe that the baseline performed by 1-layer attention can achieve better performance by 2.92% to 37.47% consider the estimation metrics. While the proposed approach can improve the baseline with 1-layer attention by 4.36% to 49.59% considering different estimation metrics. These results demonstrate that the adopted hierarchical attention can improve the performance dramatically.

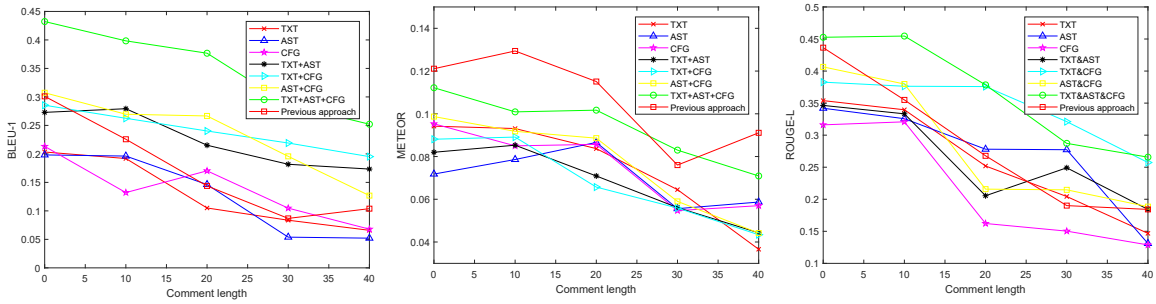
3.4.4.3 The performance of deep reinforcement learning

To estimate the performance of the proposed reinforcement learning component, I train the model by withing and without the component of reinforcement learning, and they are signed as “approach with DRL” and “approach without DRL”. The corresponding results are shown in Table 3.3, from which we can see that the proposed approach can achieve better values by 14.13% to 130.30% compared with the baseline without reinforcement learning. These results can prove that the proposed approach can improve the comment generation performance significantly.



(a) BLEU-1 (b) METEOR (c) ROUGE-L

Figure 3.9: The results trend vs. the code length.



(a) BLEU-1 (b) METEOR (c) ROUGE-L

Figure 3.10: The results trend vs. the comment length.

3.4.5 RQ2: Performance considering different code and comment length

To measure the influence on the code summarization performance considering different code and comment lengths, I vary the split of the dataset by the lengths of code and comment respectively. Figures 3.9 and 3.10 demonstrate the results, and from which we can see that the best performance can be achieved by the proposed approach compared with the baselines from the aspect of all the utilized metrics. From the aspect of BLEU, the proposed approach improves the baselines that adopted different representations of source code by 35.74%, 43.31%, 41.13%, 17.04%, 1l6.97%, and 12.66% respectively when the source code contains 40 tokens. In specific, the baselines that adopted two representations of the source code can all achieve better performance than the baselines which adopts only one representation of the source code. Furthermore, the proposed approach can always improve the performance compared with the baselines which adopted two representations of the source code. For other estimation metrics, the similar results can be achieved. These phenomenons can prove the effectiveness of the proposed

Table 3.4: The time cost to train different models (mins).

	TXT	**AST**	**CFG**	**TXT&AST**	**TXT&CFG**	**AST&CFG**	**All**
Actor	20	24	23	32	31	33	39
Critic	27	30	29	41	40	41	50
A2C	36	41	43	50	51	53	58

code representation component.

The code summarization result considering different comment lengths is presented in Figure 3.10, from which we can find that the proposed approach performs better compared with the baselines considering different lengths of comment. In specific, when the length of the comment is 20, the performance of the proposed approach improves 107.61%, 100.31%, 200.59%, 51.77%, 42.64%, and 47.77% respectively. We can observe that the performance tends to be worse when the length of the comment increased which is similar to the results in the neural machine translation task [55, 94].

3.4.6 RQ3: Time consumption analysis

To estimate the time complexity of the proposed approach, the average training time of each epoch considering different representations of the source code are recorded. The result is given in Table 3.4, from which we can find that for all the three stages in the proposed approach, each epoch costs less than 1 hour for training. This result shows that the time complexity of the proposed approach is low and the approach is reasonable for practical usage.

3.4.7 Case study

We demonstrate four real-world code examples for generating their comments using our approach in Table 3.5. In this table, we first show the code snippet in the second line and then give the ground truth comment which is the code comment that is collected together with the code snippet from GitHub. Next, the generated comments by different approaches are given. For our approach, shown as 2-layer+DRL, we have highlighted the words that are closer to the ground truth. It can be observed that the generated comments by our approach are the closest to the ground truth. Although the approaches with DRL (1-layer+DRL) can generate some tokens which are also in the ground truth, they cannot predict those tokens which do not frequently appear in the training data, i.e., `object` in

Table 3.5: Case study of code summary generated by each approach.

		Case example

Code snippet		`def Pool(processes=None, initializer=None, initargs=(), maxtasksperchild=None):`
		`from multiprocessing.pool import Pool`
		`return Pool(processes, initializer, initargs, maxtasksperchild)`

Ground truth		returns a process pool object.
Generated Comments	no attention	returns a list of all available vm sizes on the cloud provider.
	1-layer	returns the total number of cpus in the system.
	2-layer	return a list of all elements in te given order.
	1-layer+DRL	returns a process object with the given id.
	2-layer+DRL	returns a process object.

the case example. On the contrary, the deep-reinforcement-learning-based approach can generate some tokens which are closer to the ground truth, such as `process`. This can be illustrated by the fact that our approach has a more comprehensive exploration on the word space and optimizes the BLEU score directly.

3.5 Threats to Validity

The threat to validity is that I evaluate the proposed approach only based on the dataset which is consisted of Python code and comment pairs, therefore it may be unrepresentative of code summarization considering other programming languages. However, as the components in the proposed approach are all general models which can be adopted to realize the code annotation regarding other programming language. Besides, this approach is based on static analysis which could be a barrier to adopt the proposed approach, for example, to obtain the effective static analysis results, significant effort should be made.

3.6 Conclusion

In this chapter, I propose to utilize three representations of the source code considering both the unstructured and structural features, namely, the plain text, the sequenced abstract syntax tree and the sequenced control flow to represent the hierarchical structure of the source code. And the two-layer hierarchical attention mechanism is adopted to encode the source code representations. To estimate the effectiveness of the proposed approach, a few of experiments based on the dataset grabbed from real world projects are performed. The results prove the high quality of the generated comment.

CHAPTER 4

A Transformer-based Generative Adversarial Network Framework for Universal Code Summarization

While code summarization has been widely studied for enhancing software development and maintenance, many existing approaches exploit inadequate power of statement-wise semantic contributions for augmenting their performance. To solve such problem, in this chapter, I propose $Tr(AN)^2$, the first transformer-based generative adversarial network framework that integrates statement-wise semantic contributions for universal code summarization. Specifically, it first constructs a cross-language universal hierarchical semantic (UHS) model to classify statements according to their positions in the source code. Next, by integrating the UHS model with the transformer features, it proposes a tree-transformer-based generator for augmenting its generative effectiveness. At last, it is further enhanced by injecting the tree-transformer-based generator into the generative adversarial network. To evaluate the effectiveness of the proposed approach, a set of experimental studies and case studies are conducted upon the collected real-world dataset on multiple mainstream programming languages. The results suggest that the proposed approach can outperform most existing approaches significantly and the results of the case study further prove the information of $Tr(AN)^2$ from the developers' angle.

The reminder of this chapter is organized as follows. Section 4.1 gives a general introduction of this chapter. The background techniques are given in Section 4.2. Section

4.3 elaborates the detailed information of the designed approach. Section 4.4 demonstrates the information and results of the experimental study and case study. Section 4.6 concludes this chapter.

4.1 Introduction

As most of the effort on software development and maintenance is taken to the understanding of the tasks and the related source code [57], it is quite important for the documentation to provide their corresponding descriptions. Moreover, code documents have been widely adopted to benefit various software engineering techniques [27], e.g. software testing [124], fault localization [56], program repair [29]. To this end, by automatically generating comments for documenting code snippets [70], code summarization has been increasingly adopted and studied in both industry and academia [51, 71].

The recent research of code summarization progress towards deep-learning-based code summarization mainly utilize the machine translation models to generate comments, i.e., apply the deep neural network to encode the code snippet tokens into a hidden space and further decode them to natural language space [7, 20, 42, 113] for constructing the comment generation models. However, many of such approaches hardly exploit the power of statement-wise semantic contributions, which can potentially advance the code summarization process upon their proper usage. For instance, Figure 4.1 presents a code snippet with its ground-truth comment displayed in line 1. It can be observed that its comment can be generally constituted by the hierarchical semantics delivered from different statements. Since obviously line 10 contributes more semantics to summarizing the code snippet (i.e., forming partial comment “Return a copy of this month-day”) than any individual line from lines 6 to 9, we can further infer that the statement-wise contributions to constructing/summarizing program semantics can be distinguished based on the associated code block distributions, e.g., basic block statements (e.g., line 10) usually reflect more significant program semantics than other statements (e.g., line 6).

Although some existing approaches attempt to exploit program semantics through modeling them via abstract syntax tree (AST) [51**?**] or control flows [39], such models fail to establish straightforward statement-to-comment mappings for exploiting statement-wise semantic contributions to facilitate comment generation. Moreover, their adopted deep learning models tend to cause long-distance dependency issues which can seriously compromise the encoding effectiveness of their modeled program semantics [102]. To

```
1   /*Returns a copy of this month-day with the specified
      period added.*/
2   public MonthDay withPeriodAdded(ReadablePeriod period,
        int scalar) {
3     if (period == null || scalar == 0) { return this; }
4     int[] newValues = getValues();
5     for (int i = 0; i < period.size(); i++) {
6       DurationFieldType fieldType = period.
      getFieldType(i);
7       int index = indexOf(fieldType);
8       if (index >= 0) {
9         newValues = getField(index).add(this, index,
      newValues, FieldUtils.safeMultiply(period.getValue(
      i), scalar)); } }
10      return new MonthDay(this, newValues);
11    }
12
```

Figure 4.1: One motivating example of statements making different contributions to summarizing programs. The statement in line 10 can contribute more information for the generation of the comment as it contains more tokens in the given comment.

address such issues, in this work, I propose $Tr(AN)^2$, the first transformer-based generative adversarial network framework (GAN) that integrates statement-wise semantic contributions for universal code summarization. Specifically, $Tr(AN)^2$ first develops a universal hierarchical semantic (UHS) model that splits code blocks within functions/methods upon language-specific code block delimiters, e.g., "{ }" in Java, to classify statements for collecting the underlying method/function-oriented hierarchical semantics, i.e., statement-wise semantic contributions. Next, by combining the UHS model with a transformer-based encoder, $Tr(AN)^2$ develops a tree-transformer generator for exploiting the collected statement-wise semantic contributions to generate comments. At last, $Tr(AN)^2$ injects the tree-transformer generator in SeqGAN [118] for further augmenting the effectiveness of universal code summarization. As a result, after collecting the dataset which contains the $< code; comment >$ pairs, i.e., the source code and the their comments written by developers, $Tr(AN)^2$ can train a semantic-resourceful comment generator for summarizing code for various programming languages, i.e., universal code summarization.

Unlike most deep-learning-based approaches restrained from long-distance dependency issue [102], the adopted transformer in $Tr(AN)^2$ is advanced as it applies the self-attention mechanism which can compute in parallel and preserve the integral textual weights for encoding to enhance the text representation accuracy and effectiveness [98]. Moreover, such advantage can be even strengthened for encoding the classified statements to approach fine-grained program semantic representation. At last, the utilized

GAN framework can be used to solve the exposure bias issue for further optimizing the performance of the code summarization model.

To evaluate the effectiveness of $Tr(AN)^2$, a set of experiments based on the real-world data collections of three mainstream programming language (Python, Java, and C#) are conducted . The experimental results suggest that the proposed approach can outperform multiple recent approaches, e.g., $Tr(AN)^2$ can significantly improve the accuracy up to 40.64% from the aspect of BLEU1 compared with the mentioned baseline approaches of all the three languages. Additionally, case study is also conducted to further verify the effectiveness of the proposed proposed approach. Specifically, I issue 100 comments generated by the proposed approach to the related developers to ask for their feedback. As a result, I receive 24 responses where none of them are negative and 16 developers acknowledge the correctness of the generated source code annotations.

In short, the innovations in this research are as following:

- As far as I know, I design the first transformer-based generative adversarial network framework that integrates statement-wise semantic contributions.
- To obtain the representation of the source code precisely with its hierarchical semantics, I propose the tree-transformer-based encoder upon a universal hierarchical semantics (UHS) model for injecting the impact from the statement-wise semantic contributions to the encoding process.
- To evaluate $Tr(AN)^2$, based on the real-world benchmarks, a few of experiments are performed. The results show that the proposed approach can outperform most state-of-the-art existing approaches consider the accuracy. Besides, the results of the case study with the developer demonstrate also the quality of the generated comments.

4.2 Background

The background techniques utilized in the proposed approach, such as language model, transformer, and GAN are described in this section.

4.2.1 Language model

Usually, the language model of decoder combined with machine translation is constructed as the odds distribution based on a natural language words sequence. Assumed that there

is a sequence of length T, language model usually computes its occurrence probability according to the conditional probability of the n words, which is signed as n-gram [103]. The definition of n-gram language model is calculated in according to Equation 4.1.

$$p(y_{1:T}) = \prod_{t=1}^{i=T} p(y_t|y_{1:t-1}) \approx \prod_{t=1}^{t=T} p(y_t|y_{t-(n-1):t-1}) \tag{4.1}$$

As definition of n-gram model, only a certain number of natural language words can be referred to forecast the target word, while the language model according to neural network can utilize the information of more predecessor words to predict the aim word. There are three layers included in the deep neural network: the input layer maps the words x_t to vectors, the hidden layer recurrently computes and updates the hidden state h_t by reading the current input word vectorx_t, and the output layer computes the probability distributions of the subsequent words. For certain word y_t, its next word y_{t+1} can be predicted according to the probability $p(y_{t+1}|y_{1:t})$ which is estimated according to:

$$p(y_{t+1}|y_{1:t}) = g(\mathbf{h}_t) \tag{4.2}$$

here g means the stochastic function in the output layer which predict the target words based on the hidden state $\mathbf{h}_t$ which is calculated according to Equation 4.3:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, w(x_t)) \tag{4.3}$$

where the weight of token x_t is signed as $w(x_t)$.

4.2.2 Transformer

Transformer [98] is proposed to improve the representation of the sequence effectively by utilizing the self-attention mechanism. In these two years, many research work focus on transformer, such as BERT [25], ERNIE [93], and XLNET [111], are proposed to improve the performance of all kinds of the NLP tasks, e.g., text classification [110], natural language inference [18], retrieval question answering [99] and so on.

There are N identical layers in transformer, and each layer includes two sub-layers. as shown in Figure 5.5, the mechanism of self-attention is realized in the first sub-layer, and the second sub-layer is just a straightforward feed-forward neural network.

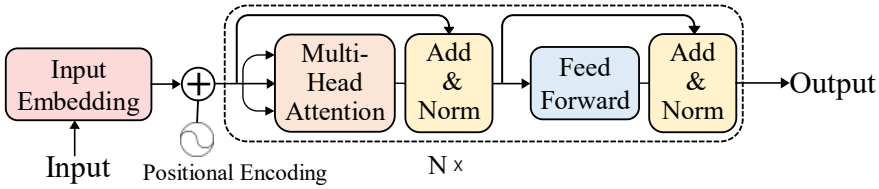


Figure 4.2: The transformer model architecture.

4.2.2.1 The self-attention mechanism

The self-attention mechanism of transformer can also be described as the intermediate attention among the tokens in the input sequence. It is calculated as follows: given the input, based on which the queries, keys and values that are with the same dimension d_k are calculated by multiplying a matrix. Then the dot products of the query with all keys is calculated and then the results is divided by $\sqrt{d_k}$. At last, the softmax function is applied on the results to get the weights, namely attention, on the given values. Actually, the attention function on all the queries which are packed into a matrix Q is computed simultaneously, thus the calculation cost can be reduced. Additionally, matrices K and V represent the packed keys and values, thus the output of the attention matrix is calculated as:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{4.4}$$

The self-attention mechanism of transformer can obtain the relationships among all the tokens of the input sequence. By taking the advantage of weights among the integral tokens, such mechanism can substantially mitigate the long distance dependency problem which is resulted by CNN, i.e., the contributions of the input tokens that from the long range are compromised. Furthermore, the multi-head attention enables the model to estimate in parallel which can dramatically reduce the calculating expense which is resulted by RNN that teh tokens are encoded sequentially.

4.2.2.2 The feed-forward neural layer

Except from the multi-head attention layer, a fully connected feed-forward network is also contained in each layer, and which is applied to each position respectively. As there is no recurrence or convolution contained in transformer, to present the sequence order, it shoots “positional encodings” to the input embedding.

4.2.3 Generative adversarial network

Generation Adversarial Net (GAN) [30] advances in learning over large (unlabeled) data by combining a generative model G and a discriminative model D. In particular, the generative model G captures the data distribution and trains them to maximize the probability of D such that D can determine whether the output of G is the ground-truth training data.

The GAN framework aims to learn the generator's distribution p over the data set x. In particular, the generator presents a mapping to data space as $G(y;\theta)$, where G is a differentiable function with a parameter set θ. The discriminator $D(x;\theta)$ computes the probability that x being part of the ground-truth training data other than the generator. We train D to maximize the probability of assigning the correct labels to both training examples and generated samples from G. I simultaneously train G to minimize $log(1-D(G(y)))$. Therefore, D and G play the following two-player minimax game with the value function $V(D,G)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p(x)}[logD(x)] + \mathbb{E}_{y\sim p(y)}[log(1-D(Gy))] \tag{4.5}$$

In order to apply GAN to text generation, we can denote a text generation task as a Markov Decision Process (MDP) under GAN with policy gradient, as presented in Figure 4.3. Specifically, given the dataset of text sequence, a generative model G_θ is trained to produce a sequence $Y_{1:T} = (y_1, ...y_T)$ where θ refers to the policy and $\mathscr{Y}$ refers to a dictionary where the words are drawn, i.e., $y_t \epsilon \mathscr{Y}$. Next, a θ-parameterized discriminative model D_θ is further trained to provide a guidance for improving generator G_θ with the odds which indicates the likelihood that sequence $Y_{1:T}$ is a piece of ground-truth training data.

4.3 A Transformer-based Generative Adversarial Network Framework for Universal Code Summarization

In this section, I present $Tr(AN)^2$, the first transformer-based generative adversarial network framework for universal code summarization. Specifically, first, $Tr(AN)^2$ develops a universal hierarchical semantic (UHS) model. For any method/function-level code snippet, its corresponding UHS model is applied to extract its statement-wise

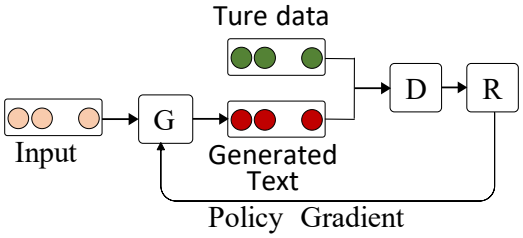


Figure 4.3: The illustration of generative adversarial network for text generation by policy gradient. (Firstly, I train the discriminative model D based on the real data and the data produced by the generator G. Then, wen train the generator G based on policy gradient.)

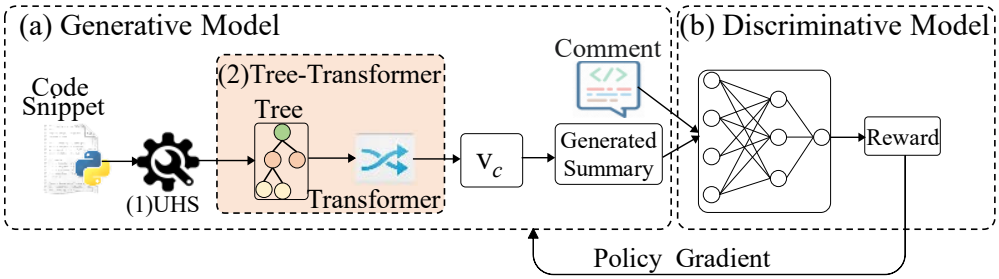


Figure 4.4: The overview of $Tr(AN)^2$.

hierarchy via the code block delimiters. Next, $Tr(AN)^2$ develops a tree-transformer to encode the UHS-derived statement hierarchy into vectors. At last, $Tr(AN)^2$ injects the tree-transformer to the GAN framework for augmenting the effectiveness of code summarization. The framework of $Tr(AN)^2$ is demonstrated in Figure 5.3.

4.3.1 Constructing the universal hierarchical semantic (UHS) model

In this work, I propose a Universal Hierarchical Semantic (UHS) model to exploit the statement-wise semantic contributions to comment generation directly. In particular, UHS constructs a tree by modeling each source code statement as a node and distributing all the nodes to different levels based on their associated code block delimiters, e.g., "{ }" in Java and indents in Python. Intuitively, such delimiters can reflect the source code semantic hierarchy. For instance, in a Java method, a statement in the basic block under no "{ }" is executed whenever its associated method is called. Such statement thus can reflect an essential piece of semantics of the method, i.e., the methods cannot be correctly executed without it. On the other hand, a statement under "{ }" possibly

indicates that (1) its executions are optional, e.g., under nested "if" block. Therefore, it can be inferred that such statement can reflect an optional piece of semantics, i.e., the method can be correctly executed under certain circumstances where such statement is not executed; or (2) it delivers limited semantic on its own. Specifically under a loop, usually all its contained statements together can deliver complete semantics while an individual statement only delivers a limited semantic scope. Therefore, we can infer that code blocks split by their delimiters can reflect the underlying program semantic hierarchy, i.e., statement-wise semantic contributions. To this end, I determine to adopt code block delimiters for constructing the cross-language source code semantic hierarchy, i.e., the UHS model, which can be represented as a tree. In this chapter, I want to explore the contribution of statement-level semantic for source code summarization, thus the AST structure cannot satisfy the requirement. Instead, each statement is considered as a whole, and the block delimiters are utilized to construct the semantic hierarchy.

After modeling individual source code statement as a node, for any given programming language, its delimiter set B with all its code block delimiters is extracted. Note that normally statements are "wrapped" under pairwise delimiters. In particular, a level index for the tree level corresponding to each statement is defined. Accordingly, for each statement, its level index is determined by counting the number of its "wrapping" code block delimiters. More specifically, first, given a method/function, its header is extracted as the root node. Next, all its statements are sequentially read till the end of the method. When the left delimiter is encountered, the level index increments by 1; otherwise when the right delimiter is encountered, the level index decrements by 1. At last, the UHS tree are constructed by laying and connecting the method header and all the statements according to the resulting level indices.

The details of how to construct a UHS tree is demonstrated in Algorithm 1. Specifically, first, all the statements along with their level indices are initialized and stored (lines 1 to 5). Next, by identifying the method/function header as the root node (line 7), the remainder of the tree is constructed based on iterating the statements and updating their respective level indices (lines 9 to 25). In particular, when reading the statements in turn, there are a total of three cases of determining level indices. Firstly, if the level index of the present statement equals the level index of its preceding statement plus 1 (line 11, where e[0] denotes the level index and e[1:] stores the associated statement tokens), the present statement is constructed as the child node of its preceding statement. Secondly, if the level index of the present statement equals the level index of its preceding statement (line 16), the present statement is constructed as the sibling node

Algorithm 1 UHS Tree Construction

Input : source code

Output: tree representation of code

```
1: function CONSTRUCTTREE
2:     statement ← level_index + statement
3:     Initialize queue Q
4:     for each statement in code do
5:         s ← [level_index, statement]
6:         EnQueue(Q, s)
7:     end for
8:     level_index ← 0
9:     root ← GetHead(Q)
10:    while !Empty(Q) do
11:        e ← GetHead(Q)
12:        if e[0] == level_index +1 then
13:            root's child ← e[1]
14:            root ← e[1]
15:            level_index ++
16:        else
17:            if e[0] = level_index then
18:                root's sibling ← e[1]
19:            else
20:                if level_index - e[0] == i then
21:                    i-th generation ancestor's sibling of root ← e[1:]
22:                    level_index = level_index - i
23:                end if
24:            end if
25:        end if
26:    end while
27: end function
```

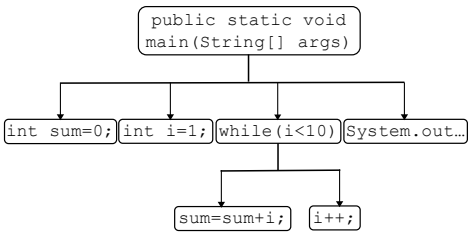
of its preceding statement. Lastly, if the level index of the present statement equals the level index of its preceding statement minus i (line 19), the present statement is constructed as the i-th generation ancestor's sibling of its preceding statement.

We apply a Java code snippet as demonstrated in Figure 4.5 to illustrate the UHS tree constructions process. In particular, Figure 4.5 (a) demonstrates an example of Java code snippet. Firstly, the level indices of statements 1–7 are labeled as: 0, 1, 1, 1, 2, 2, and 1 respectively. Accordingly, statement 1 is constructed as the root of the tree while statements 2, 3, and 4 are constructed as its child nodes because their level indices are

Java Code Snippet:

```
1. public static void main(String[] args) {
2.    int sum = 0;
3.    int i = 1;
4.    while (i<10){
5.       sum = sum + i;
6.       i++;}
7.    System.out.print(sum);}
```

(a)An example Java code snippet.



(b) The tree structure of the Java code snippet.

Figure 4.5: An example of code snippet and its tree structure.

one away from the root node. Next, statements 5 and 6 are constructed as the child nodes of statement 4, because statement 4 is their recent preceding node with level index 1. At last, statement 7 is constructed as the child node of statement 1 and the sibling node of statement 4. As a result, the UHS tree of the code snippet in Figure 4.5 (a) can be constructed in Figure 4.5 (b).

4.3.2 Constructing the tree-transformer encoder

In order to construct the tree-transformer encoder, a primitive transformer encoder is built to obtain the encoding of the comment and the program statements. Accordingly, the tree-transformer encoder is further constructed that integrates the UHS tree of the executing code snippet and the transformer encoder to enhance the encoding of the code accuracy.

4.3.2.1 Transformer Encoder.

As in Section 4.2.2, transformer consists of N identical layers where at each layer, the multi-head attention employs h attention heads and performs the self-attention mechanism. In each attention head, the sequence of input vectors $X = (x_1, x_2, ..., x_m)$, which is the embedding of a piece of source code based on word2vec [65], is transformed into the sequence of output vectors, $S = (s_1, s_2, ..., s_n)$ as Equation 4.6:

$$
\begin{aligned}
e_{ij} &= \frac{W^q x_i (W^k x_j)^T}{\sqrt{d_k}} \\
\alpha_{ij} &= \frac{exp\ e_{ij}}{\sum_{k=1}^{m} exp\ e_{ik}} \\
s_i &= \sum_{j=1}^{m} \alpha_{ij} (W^v x_j)
\end{aligned}
\tag{4.6}
$$

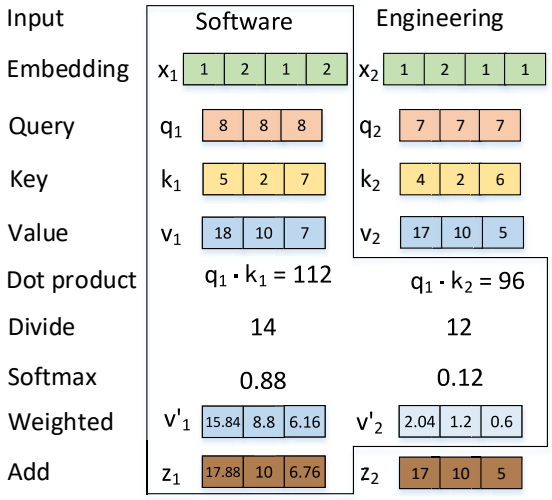


Figure 4.6: An example of transformer encoder.

where W^q, W^k and W^v are the parameters that are unique per layer and attention head. More specifically, for vector x_i which is the representation of each token, I then calculate its representation based on the self-attention mechanism: (1) to derive q_i, k_i and v_i, namely the vectors of query, key and value which are marked as W^q, W^k and W^v, I first multiply the input token vector x_i with a initialized matrix, (2) then by using the dot product of $q_i \cdot k_j$, the contributions of x_i from the vectors of all the input tokens are calculated, where $j \epsilon [1, n]$ and n means the input token number, (3) next, I compute the scores e_{ij} of x_i by $\sqrt{d_k}$ and d_k is k_i's dimension number and normalize the results by softmax to derive the weights of all the input tokens α_{ij}, (4) then an interim vector space $v^{'}$ is computed by calculating the product of these weights and their related value vectors, and (5) at last, z_i which is the final vector representation of x_i is obtained, through calculating the summation of the vectors contained in $v^{'}$. Finally, to get the final vector representation of the input tokens, namely s_i, the feed-forward neural network are adopted to further process the obtained token vectors.

Figure 4.6 presents how does the transformer encode the input sequences. Firstly, the natural language words "*Software*" and "*Engineering*" are translated to vectors x_1 and x_2 respectively. For the input vector x_1, its realted query vector q_1, key vector k_1, and value vector v_1 are calculated at begin. Then, its "attention" scores paid from other input tokens, i.e., x_1 and x_2, is calculated based on $q_1 \cdot k_1$ (112) and $q_1 \cdot k_2$ (96). Assume that the value of d_k is 64, by dividing and normalizing the resulted dot products by $\sqrt{d_k}$, the "attentions" from x_1 and x_2 is 0.88 and 0.12. Finally, the value of z_1 can be calculated according to 0.88*v_1 + 0.12*v_2.

Algorithm 2 Tree-Transformer Encoding

Input : ordered tree

Output: vector representation of the tree

```
1: function POSTORDERTRAVERSE
2:     i ← root node
3:     node_list ← { }
4:     if isLeaf(i) then
5:         return i;
6:     else
7:         for i's child_j do
8:             if isLeaf(child_j) then
9:                 node_list.append(child_j)
10:            else
11:                PostorderTraverse(i's child nodes)
12:            end if
13:        end for
14:        i ← transformer(node_list)
15:    end if
16: end function
```

4.3.2.2 Tree-Transformer Encoder.

In order to construct the tree-transformer encoder to incorporate the UHS tree of code snippets for encoding, first, an ordered UHS tree is obtained by applying the UHS model on the given code snippets. Next, the UHS tree is traversed and each of its nodes, namely each statement, is encoded into a vector by the transformer.

Consider the special structure, I traverse the tree obtained based on UHS in a post-order manner. Additionally, I also develop a transformer-merger strategy during the tree traversal. In particular, assume node n_i and its parent node n_j in the tree structure. If node n_i is the leaf node, I update it by the transformer-produced vector of n_j, namely V_{n_j} and concatenating their respective vectors as $\{V_{n_i}, V_{n_j}\}$ and then process n_j's other child nodes and update its vector with them; on the other hand, I traverse n_i's child nodes. In this way, I capture the underlying low-level code blocks and concatenate them with their high-level code blocks as a whole to update their overall weights in the network. As a result, the representation of the input sequence can be obtained by encoding the tree derived from UHS using the tree-transformer, and Algorithm 2 describe the detailed information of the the tree-transformer encoding process.

The example in Figures 4.5 (a) and 4.5 (b) is utilized to show how does the tree-transformer encoder work. Concretely as shown in Figure 4.5 (a), nodes “int sum = 0”, “int i = 1”, “while(i<10)” and “System.out.print(sum);” are constructed as siblings because they are assigned with the same level index 1. Their preceding statement “public static void main(String[] args)” are constructed as the parent (root) node. Next, by utilizing transformer, all the statements of the program code are encoded into vectors respectively. Then, as the child nodes of the root “int sum = 0;” and “int i = 1;” are all leaf nodes, the root’s vector is updated by injecting the concatenated vectors of its two child nodes. At the same time, as the child node of the root “while(i<10)” is not leaf node, its child nodes “sum = sum + i;” is encoded with “i++;” by transformer, followed by encoding the resulting vectors with “while(i<10)”. Finally, the root with all the derived vectors of its child nodes are encoded to obtain the final vector representation of the tree structure for this code snippet. Figure 4.7 further illustrates the transformer-merger process for encoding a code snippet block composed of statements s_4, s_5 and s_6 of the code snippet example shown in Figure 4.5. Specifically, assuming that v_j represents the vector of token j and statement s_4 can be vectorized as <v_{while}, v_i, $v_<$, v_{10}>, firstly, the vectors of statements “while (i<10)”, “sum = sum + i;” and “i++;” are encoded by transformer with their respective resulting vectors v_{s4}, v_{s5}, and v_{s6}, where v_{s4} is derived by applying transformer on <v_{while}, v_i, $v_<$, v_{10}> (namely the tokens of statement 4), v_{s5} is derived by applying transformer on <v_{sum}, $v_=$, v_{sum}, v_+, v_i> (namely the tokens of statement 5), and v_{s6} is derived by applying transformer on vector <v_i, v_+, v_+> (namely the tokens of statement 6). Next, as v_{s5} and v_{s6} are the child nodes of v_{s4}, then they are represented as a sequence, namely, <v_{s4}, v_{s5}, v_{s6}> and transformer is utilized to encode this sequence, i.e., the resulting vector v_b is derived by further applying transformer on <v_{s4}, v_{s5}, v_{s6}>.

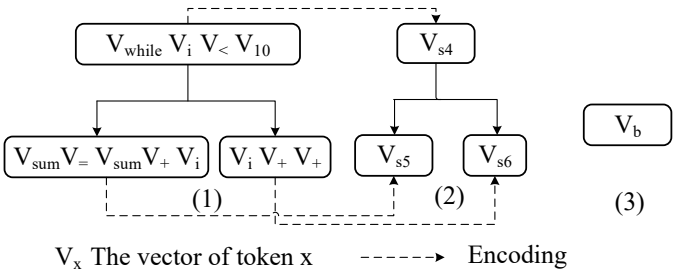


Figure 4.7: An example of the encoding process of a code block by tree-transformer. (1) Embedding the tokens in each statement into vectors. (2) Encoding each statement by transformer. (3) Encoding the vector of the statements in the same block by transformer.

4.3.3 SeqGAN for code summarization

By adopting the tree-transformer as the generative model of the proposed framework, the SeqGAN [118] is adopted for the comment generation task because SeqGAN extends GAN to generate sequences in turn under a reward signal provided by the RL-based discriminative model. In particular, I characterize a comment generation task as a Markov Decision Process (MDP), i.e., given a dataset of real-world code and comments, the generator G_θ is trained to produce a sequence $Y_{1:T} = (y_1, ...y_T)$. More specifically, θ refers to the policy parameters to be learned and the $action$ space is signed as a dictionary $\mathscr{Y}$ from which the words are drawn, i.e., $y_t \epsilon \mathscr{Y}$. Accordingly, at certain time t, state s is calculated by encoding the produced tokens $(y_1, ..., y_{t-1})$ and the next selected token y_t refers to the action a. As shown in Equation 4.7, the generator aims to obtain the $policy$ that can get the maximal reward of the produced natural language sequence,

$$\max_\theta \mathscr{L}(\theta) = \max_\theta \mathbb{E}_{\substack{\mathbf{x}\sim\mathscr{D} \\ \hat{\mathbf{y}}\sim P_\theta(\cdot|\mathbf{x})}} [R(\hat{\mathbf{y}}, \mathbf{x})] \tag{4.7}$$

here $\mathscr{D}$ refers to the dataset, $\hat{\mathbf{y}}$ means the $action$s/words produced, and the reward function is signed as R.

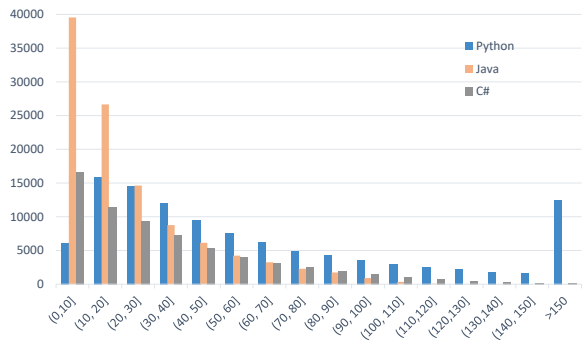
Next, a θ-parameterized discriminative model D_θ can be further trained to provide a guidance for improving generator G_θ under a probability indicating the likelihood a sequence $Y_{1:T}$ being a piece of ground-truth training data. As Figure 4.3 illustrated, the positive examples from the dataset and the negative examples produced by generator G_θ are adopted to train the discriminative model D_θ. Simultaneously, the generator G_θ is renewed by adopted policy gradient according to the reward calculated by the discriminative model. The input sequence $x_1, ..., x_T$ is signed as:

$$\theta_{1:T} = x_1 \oplus x_2 \oplus ... \oplus x_T, \tag{4.8}$$

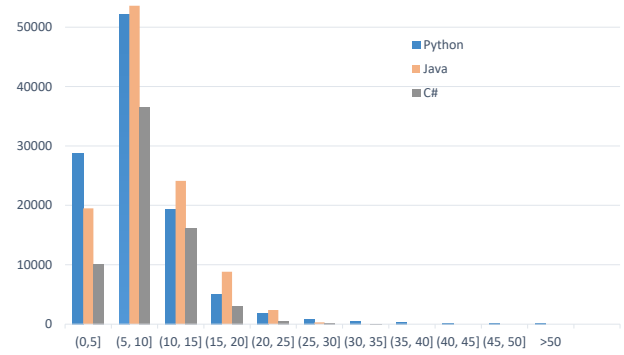
here $x_t \epsilon R^k$ is the embedding of of the input sequence and $\oplus$ denotes the concatenation operator. Then to generate the new features, a window size of l words is operated by the kernel $w \epsilon R^{l \times k}$, namely:

$$c_i = \rho(w \otimes \theta_{i:i+l-1} + b), \tag{4.9}$$

here $\otimes$ means the summation operator of element-wise production, and ρ denotes the non-linear function. Then, the pooling operator is applied over the feature maps obtained by applying various kernels. At last, the probability of output is computed by applying a layer with sigmoid activation function to act as the reward to feed back to the generator.

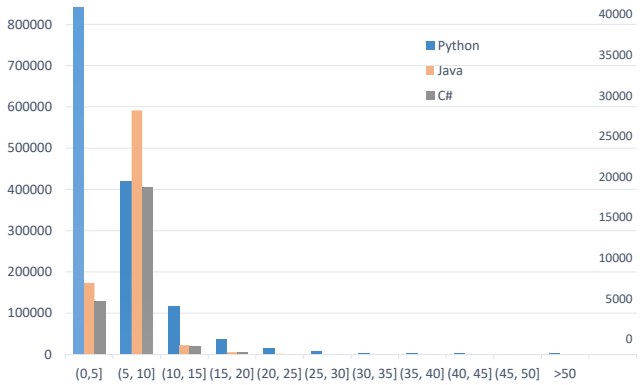


(a) Code length distribution.

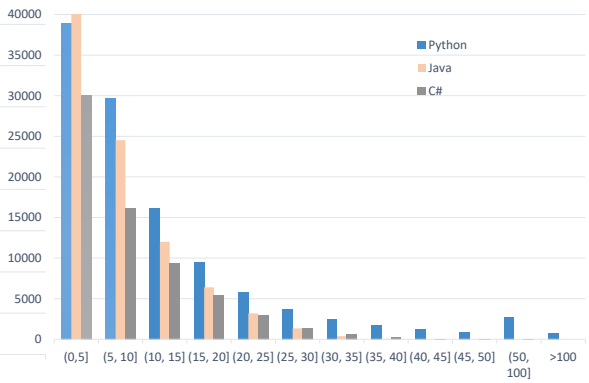


(b) Comment length distribution.

Figure 4.8: Length distribution of data.



(a) Token number distribution.



(b) Statement number distribution.

Figure 4.9: Statistic analyses of the source code.

4.4 Evaluation

To estimate the accuracy of the proposed approach, a few of studies including experimental study and case study are conducted.

4.4.1 Experimental studies

4.4.1.1 Benchmark construction

Dataset. To prove the effectiveness and the universality of the proposed $Tr(AN)^2$, three mainstream high-level programming languages (Python, Java, and C#) are utilized for the proposed experiments. For Python, the dataset given in [15] is utilized, in which about 120,000 pairs of code and comment are included that are grabbed from real projects in Github [2]. There are about 50,400 program tokens and 31,300 natural language words included in this dataset. For Java, I adopt 120,000 <code; comment> pairs of the

dataset presented in [5] which includes about 50,000 program tokens and 32,000 natural language tokens. For C#, I adopt the dataset collected in [43] which is composed with about 65,000 <code; comment> pairs and these dataset includes 40,000 code tokens and 25,000 comment tokens. These three datasets are commonly utilized in which the rare word are regarded as unknown words and the repetitive source code are removed. Each dataset is split into 60%, 20% and 20% for training, validation, and testing respectively.

To analysis the features of the datset, I have also done statistics analysis and the results are shown in Figures 4.8 and 4.9. Figure 4.8 shows the distributions of code and comment length. As shown in Figure 4.8 (a), there are more shorter code snippets than longer ones among all the programming languages; from Figure 4.8 (b), we can find that most comments of the Python code contain from 0 to 25 words, the length of most comments of the Java code are between 5 and 20 words, and the length of most comments of the C# code are between 0 and 20 words. Besides, Figure 4.9 presents the number distribution of the token and statement in the proposed adopted datasets. The distribution of the token number in each statement is shown in Figure 4.9 (a) and Figure 4.9 (b) presents the number distribution of the statement. According to these results, we can know that each statement includes 0 to 20 tokens, fand each function includes 0 to 40 statements fro these three datasets.

Baselines. I select several existing approaches, i.e., CodeNN [43], CoaCor [113], and Rencos [120] for performance comparison with $Tr(AN)^2$. In particular, CodeNN [43] is the first comment generation approach of source code via neural network. It first encodes the program code to sequence vectors and then decodes them to produce annotations based on the seq2seq model realized by LSTM. *CoaCor* [113] adopts the plain sequence of the program code and realizes code summarization by using the seq2seq framework based on LSTM. Rencos [120] is recently proposed to integrate the powers of retrieval-based and NMT-based approaches and can be considered as state of the art. Given the program code, Rencos first retrieves the dataset to find the most similar code, then the input and the two source code retrieved are encoded, and eventually fuses them during decoding to generate the summary.

In addition, I also evaluate the variant techniques of $Tr(AN)^2$ to explore the performance of its components. Specifically, ${Tr(AN)^2}_{base}$ adopts the transformer encoder with the OpenNMT framework [48]; ${Tr(AN)^2}_{GAN}$ integrates the transformer encoder with GAN; ${Tr(AN)^2}_{tree}$ adopts the tree-transformer encoder with the OpenNMT framework; and ${Tr(AN)^2}_{tree+GAN}$ refers to the complete $Tr(AN)^2$ which integrates the tree-transformer encoder with code snippet and GAN-based reinforcement learning.

4.4.1.2 Experimental setups

In the experiments, I set the word embedding size, the batch size, the layer size and the head number to 1280, 2048, 6, and 8 respectively. I pretrain both the generative model and the discriminative model with 20000 steps each, and then I train the generative adversarial framework with 100000 steps concurrently. Python 3.5 is utilized to conduct all the experiments.

4.4.1.3 Implementation

I implement the proposed approach based on SeqGAN [118]. After obtaining the dataset, they are split to 60%, 20% and 20% for training, validation and testing respectively. Then, the tree structure of the code snippet is assembled respectively. Next, the code and corresponding comment pairs are inputted to the SeqGAN framework for training according to the settings in Section 4.4.1.2. Then, the trained model is obtained for the code summarization task and conduct testing based on the model. To evaluate the performance, similar as previous work, I evaluate the quality of the generated comments in terms of their similarity with the corresponding ground-truth comments. There are three commonly utilized estimation metrics adopted in the research area of natural language processing particularly in the the neural machine translation task are utilized: BLEU [77], METEOR [12], and ROUGE [58]. These metrics mainly calculate the similarity degree between the generated natural language text and the ground-truth by measuring the frequency of the tokens occurrence in both of them from different aspects. As comment generation is also a text generation task in which the natural language words composed the output, thus these metrics are utilized to assess the accuracy of the produced annotation. Particularly, the most common metric–BLEU adopted in the natural language generation task which estimates the n-gram accurateness based on a set of reference sentences. METEOR considers the stemming and synonymy matching and estimates to what extent the generated outputs grab the input information based on the references according to recall. ROUGE-L calculates the similarity in sentence level by matching the longest common sequence between two text sequence.

4.4.1.4 Result analysis

For the Python and C# Datasets Table 4.1 and 4.3 demonstrate the results of code summarization obtained from all the approaches consider of the metrics described above for the dataset of Python and C#. For the results of dataset Python, while the

Table 4.1: The result of comment generation with different metrics for the Python dataset.

Approaches	**BLEU1**	**BLEU2**	**BLEU3**	**BLEU4**	**ROUGE-L**	**METEOR**
CodeNN	23.47	8.77	5.16	3.64	31.00	9.42
CoaCor	30.56	14.98	11.06	8.57	34.27	12.06
Rencos	24.02	7.33	3.71	2.47	28.28	10.82
$Tr(AN)^2{}_{base}$	24.84	12.69	10.35	9.67	28.92	10.33
$Tr(AN)^2{}_{GAN}$	24.53	13.73	11.57	10.04	28.74	10.17
$Tr(AN)^2{}_{tree}$	31.05	17.62	17.03	14.13	43.92	10.74
$Tr(AN)^2{}_{tree+GAN}$	**38.10**	20.20	**17.70**	**16.30**	**50.74**	**12.98**

compared approaches achieve the performances from 23.47% to 30.56% in terms of BLEU1, $Tr(AN)^2$ can achieve 38.10%. Particularly, we can observe the detailed phenomenon described blow. Firstly, we can see that $Tr(AN)^2$ can outperform the compared approaches significantly consider the estimation metrics. For instance, $Tr(AN)^2$, i.e., $Tr(AN)^2{}_{tree+GAN}$ can achieve better performance from 19.79% to 38.40% from the aspect of BLEU compared with baseline. Such advantages of performance can prove the outstanding of the proposed proposed $Tr(AN)^2$ which can be illustrated as follows: (1) the adopted transformer-enabled self-attention mechanism outperforms the regular attention mechanism, it is because the adopted self-attention mechanism can grab the intermediate attention among the token contained in the input sequence effectively to better reflect their semantics and therefore can optimize the weights of the language model globally; (2) the proposed adopted discriminative model from the GAN framework can extend the performance advantage by providing a reward to the generative model to enhance the accuracy of the produced comments. In addition, we can observe that each component of $Tr(AN)^2$ can contribute to improving the results. For example, through adopting the tree-transformer encoder, $Tr(AN)^2{}_{tree}$ can dramatically outperform $Tr(AN)^2{}_{base}$ which only adopts the transformer encoder by 20.00% consider BLEU. Besides, we can also see that the tree transformer which leverages the UHS model can enhance the the language model by augmenting the hierarchical semantics for tokens effectively. Moreover, by applying GAN, $Tr(AN)^2{}_{tree+GAN}$ outperforms $Tr(AN)^2{}_{tree}$ by 18.50% from the aspect of BLEU, by which the strength of GAN can be further validated. The result of C# demonstrates about the similar performance.

For the Java Dataset The results of the Java dataset for all the approaches from

Table 4.2: Code summarization results with different metrics for the Java dataset. (Best scores are in boldface.)

Approaches	**BLEU1**	**BLEU2**	**BLEU3**	**BLEU4**	**ROUGE-L**	**METEOR**
CodeNN	33.20	21.87	17.71	15.03	38.24	16.13
CoaCor	33.25	19.71	14.83	12.19	36.03	14.85
Rencos	32.98	21.42	17.19	14.80	36.00	17.26
$Tr(AN)^2{}_{base}$	35.24	17.56	13.96	13.01	38.51	17.96
$Tr(AN)^2{}_{GAN}$	34.95	17.31	13.12	12.98	38.00	17.86
$Tr(AN)^2{}_{tree}$	36.23	18.70	15.50	14.30	39.25	17.41
$Tr(AN)^2{}_{tree+GAN}$	**37.80**	20.60	16.40	14.30	**43.42**	**18.37**

Table 4.3: Code summarization results with different metrics for the C# dataset. (Best scores are in boldface.)

Approaches	**BLEU1**	**BLEU2**	**BLEU3**	**BLEU4**	**ROUGE-L**	**METEOR**
CodeNN	22.51	9.76	5.95	4.78	25.58	8.94
CoaCor	20.13	8.25	5.34	4.21	25.16	10.46
Rencos	24.96	12.25	7.01	5.28	28.64	10.89
$Tr(AN)^2{}_{base}$	22.17	9.23	5.98	4.16	25.48	8.66
$Tr(AN)^2{}_{GAN}$	22.59	8.38	5.03	3.66	25.54	9.03
$Tr(AN)^2{}_{tree}$	23.13	9.79	5.34	3.14	26.74	10.65
$Tr(AN)^2{}_{tree+GAN}$	**26.65**	12.44	6.91	4.13	**30.46**	**11.57**

the aspect of the metrics described above is presented in Tables 4.2. We can observe that similar to the Python evaluations, $Tr(AN)^2$ can achieve better performance compared with the baseline approaches from the aspect of the most evaluated metrics. We can see from the results that although some approaches can outperform $Tr(AN)^2$ consider BLEUN ($N \geqslant 2$), the BLEU1 result of $Tr(AN)^2$ still outperform the compared approaches. This indicates that $Tr(AN)^2$ can generate more correct comment tokens while the order of the tokens may be wrong. In some cases, more comment tokens can still be helpful for the understanding of the source code in different order.

To summarize, while the compared approaches incur evident performance variance that result in different performance rankings under different programming languages, $Tr(AN)^2$ can significantly and consistently outperform them all under all the selected

Table 4.4: Sample issues for code summarization case study

Example	example 1	example 2

Code snippet	`def ValidateXsrfToken(token,action):`
	`  usr_str = _MakeUserStr()`
	`  token_obj = memcache.get(token,`
	`  namespace=MEMCACHE_NAMESPACE)`
	`  if not token_obj:`
	`  return False`
	`  token_str, token_action=token_obj`
	`  if usr_str != token_str or action`
	`  != token_action:`
	`  return False`
	`  return True`
		`def add_callers(target, source):`
		`  new_callers = {}`
		`  for func,caller in target.iteritems():`
		`  new_callers[func] = caller`
		`  for func,caller in source.iteritems():`
		`  if func in new_callers:`
		`  new_callers[func] = caller +`
		`  new_callers[func]`
		`  else:`
		`  new_callers[func] = caller`
		`  return new_callers`

Ground-truth	Validate a given XSRF token by retrieving it from memcache.	Combine two caller lists in a single list.
Generated comment	Validate a given xsrf token by retrieving it.	Combine two lists in a list.
Feedback	*"Yes, this is correct. Validate a retrieved XSRF from the memory cache and then with the token perform an associated action."*	*"The `pstats` package is used for creating reports from data generated by the Profiles class. The `add_callers` function is supposed to take a `source` list, and a `target` list, and return `new_callers` by combining the call results of both target and source by adding the call time."*

programming languages.

4.4.2 Case studies

To further estimate the effectiveness of $Tr(AN)^2$ from the practical point of view, wen then conduct some case studies. In particular, similar to [120], I first randomly collect 100 code snippets from the testing set and generate their comments by the trained $Tr(AN)^2$ model. Then, the produced natural language annotations are issued to the related developers and I ask them to estimate if the annotations can describe their source code correctly, where each issue is formalized as "Hi there, I was reading the code of the function <function name> in <the function path>. My understanding is <generated comment>. Do I understand that right?" As a result, I received a total of 24 responses, among which 16 developers issued positive feedback to the generated comments. More specifically, 11 developers thought that the generated comments can summarize the main intent of their source code directly and 5 developers described the detailed intent of their corresponding source code. The remaining have not judge the quality of the generated summary, e.g., some developers questioned if the issue was sent by robot.

Table 4.4 demonstrates the selected examples which present the source code, the comments given by the developer, the generated comments, and the feedback from the

developers. The first case demonstrates that the developer affirm that the generate annotations by the proposed approach is correct. From the examples we can observe that the comment given by the developer is that "Validate a given XSRF token by retrieving it from memcache", and $Tr(AN)^2$ generates "Validate a given xsrf token by retrieving it". The proposed approach can generate the important words given by the original comments that indicate the major features accomplished by the code snippet, e.g., "validate" and "retrieving". Developers from the second case have not confirm that the generated code summary can describe their aim directly, while they are still supportive to the generated summary. The original comment of the second code snippet is "Combine two caller lists in a single list", and the proposed approach generates "Combine two lists in a list". Although the developer has not directly confirmed the correctness of the generated comments, his feedback that explains the usage of this code snippet can be extended for implying that the generated comments conforms to his program logic. Moreover, it can be simply inferred by comparing the generated and original comments that they are semantically close to each other.

4.5 Threats to Validity

There are two main threats to validity. Firstly, deep-learning-based approaches usually are sensitive to datasets. To reduce this threat, multiple datasets widely-used in different programming languages (Java, Python, and C#) [41, 43, 100] are utilized to evaluate the effectiveness of the proposed approach respectively. The results that the proposed approach obtains a consistent performance around 40% can indicate that $Tr(AN)^2$ does not tend to be sensitive on datasets. Secondly, it can be inadequate to only rely on the limited number of token-counting-based metrics to fully evaluate the effectiveness of $Tr(AN)^2$. To reduce this threat, I first adopt multiple evaluation metrics, i.e., BLEU, METEOR, and ROUGE to estimate the quality of the generated code comments. Next, I also conduct several case studies by issuing the generated code comments to the related developers to ask for their feedback. The results that $Tr(AN)^2$ significantly outperforms other compared approaches under all the selected metrics and none of the feedback are negative indicate that $Tr(AN)^2$ is able to generate high-quality code comments.

4.6 Conclusion

In this work, I propose a transformer-based generative adversarial network framework for universal code summarization by injecting the syntactic structure of the source code. Specifically, it first builds a universal hierarchical semantic (UHS) model for reflecting the hierarchical semantics of the source for multiple programming languages. Accordingly, it then further constructs a tree-transformer for encoding the derived source code representation. Furthermore, the generative adversarial network framework is constructed by adopting the tree-transformer as its generative model. We conduct a set of experimental studies under multiple mainstream high-level programming language datasets (Python, Java and C#) to estimate the effectiveness of the proposed approach, where the experimental results suggest that the proposed approach significantly outperforms multiple baselines. Besides, a set of case studies are also conducted by issuing the annotations generated by the proposed approach to the related developers and ask for their feedback. As a result, none of the feedback are negative.

CHAPTER 5

On Semantic-rich Code Summarization by Vital Code and Transformer-based Model

While code summarization has been widely studied for enhancing software development and maintenance, the existing approaches nevertheless incur evident issues. Specifically, almost all approaches only consider to generate the general intent of the method without documenting their parameters. Besides, the information-retrieval-based approaches are argued to be data dependent, while the deep-neural-network-based approaches tend to be restrained from long-distance dependency, excessive computation cost, and/or inadequate semantic delivery. To address such issues, in this work, I propose the first neural network model for the dual tasks of method comment and parameter comment generation. Specifically, a programming-analysis-based component is designed to extract the important statements set and the usage statements set of the parameter in the code snippet to obtain the main semantic information and discard the useless noise information for the dual tasks. Next, the copy-attention-integrated transformer based the Neural Machine Translation framework is utilized for both the method comment and the parameter comment generation to provide complete java documentation for the code snippets. To evaluate the effectiveness of the proposed approach, a set of experimental studies are conducted based on the collected dataset grabbed from github. The results prove that the proposed approach can outperform multiple state-of-the-art approach significantly.

The organization of this chapter is shown as follows. Section 5.1 gives a general

introduction of this chapter. A motivating example is illustrated in Section 5.2. Section 5.3 elaborates the detailed information of the proposed approach. The results and analysis of the experiments are described in Section 5.4. Threats to validity are indicated in Section 5.5. Section 5.6 concludes this chapter.

5.1 Introduction

Since understanding the tasks and the related code snippet cost much of the effort during the software development and maintenance [57], thus it is quite necessary for effective documentation to provide their corresponding descriptions. To this end, code summarization has been increasingly adopted and studied in both industry and academia to generate high-level natural language comments to document source code automatically [51, 113].

Although existing code summarization techniques show promising results in generating summary for a code snippet [43, 51, 91, 113], the automatically generated summaries by these techniques have some limitations that may reduce their usefulness for developers in code understanding. Firstly, the generated summaries may be *redundant comments* [60] that do not add much value to code understanding. For example, Figure 5.1 shows an example of method summary for the method `setPort` that simply repeats the method name without any additional information. On the other hand, several widely-adopted commenting guidelines [64, 109] indicate that developers should avoid writing code comments that merely repeat what the code does. Secondly, most of the code comment generation techniques only produce one sentence as a description for a given method. Such method summaries may be incomplete as they only explain the general intent of the methods without documenting their parameters. To generate complete API documentation for Java programs, prior work suggested augmenting method summaries with parameter comments [92]. However, as the parameter comments are only loosely integrated with the corresponding method summaries, the entire generated code comment may not precisely capture the semantics of the method.

While many existing approaches integrate deep learning [8, 42, 53, 91] or information retrieval [36, 71, 108, 120] techniques to approach the correctness of the generated comments, i.e., enhance the word-level mappings between the code summary and the ground truth, their generated function/method-level comments can incur the following two challenges to result in inadequate semantics.

First, generating only function/method-level comments often hardly deliver sufficient

```
1  /*set the port number.*/
2  public ClusterServerModifyRequest setPort(Integer port
       ) {
3  this.port = port;
4  return this;
5  }
6
```

Figure 5.1: A method summary that simply restates the method name

semantics for developers to fully leverage the corresponding source code. For instance, for complex (large-scale) functions/methods, it is quite essential for developers to understand the semantics behind parameters/variables rather than only the function/method-level comments prior to invoking them. Though some existing approaches [68, 91, 92] attempt to construct templates to generate comments for function/method parameters, such templates are argued to cause limited scalability [41, 52, 105].

Second, even the generated function/method-level comments themselves cannot fully capture the essential semantics of their associated functions/methods. In particular, many existing deep-learning/information-retrieval-based approaches adopt $\langle code, comment \rangle$ pairs as the ground truth which the overall comment generation process is subject to. In their training stages, the adopted function/method-level comments essentially guide the comment generation model training such that the resulting model can approach the quality of the ground truth, i.e., the quality of the generated comments is strongly bounded by the quality of the ground truth. Hence, the existing approaches are likely to deliver inferior semantics compared with the ground truth in the model training stage where the resulting model can generate function comments with inferior semantics. Moreover, particularly for simple (small-scale) functions/methods, since their semantics can be easily understood or simply reflected by their names, the contributions of delivering program semantics of such generated inferior-semantic comments can be further restrained.

To address the aforementioned challenges, in this work, I propose $TranS^3$, a semantic-rich code summarization framework that automatically generates function comments and parameter comments via constructing one neural network model. Specifically, $TranS^3$ is launched by identifying and collecting multiple statement groups, including $UseSet$ extracted by analyzing the use of the associated function parameters and $KeySet$ extracted certain statements with major contributions to structuring the overall function semantics. Next, $TranS^3$ adopts a transformer layer and a copy attention layer to encode the

collected corpus of the $\langle UseSet$, *parameter comment*$\rangle$ and $\langle KeySet$, *function comment*$\rangle$ pairs respectively and enables their corresponding decoding process to construct the neural network model. As a result, given any function/method, the resulting transformer-based neural machine translation model can deliver its function comment and parameter comment.

$TranS^3$ advances in generating semantic-rich comments due to the following reasons. For the first challenge, for training the comment generator, the selected statement groups can better represent the overall semantics of the associated functions by reducing the noise caused by retaining all the statements in the traditional approaches. In addition, the adopted copy attention mechanism can not only effectively capture the semantic connections between source code and natural language but also generate rare words from the source code that are not included in the original vocabulary. For the second challenge, $TranS^3$ expands the original $function\ comments$ to the $function\ parameter\ comments$ to inject more semantics to the training stage. Accordingly, the function comments generated by the resulting model out of $TranS^3$ are expected to deliver more semantics as well.

To evaluate the effectiveness of $TranS^3$, several experiments are performed according to the collected real-world data collections. From which the results demonstrate that the proposed approach can achieve better performance compared with the recent published approaches, e.g., compared with the selected baselines, the proposed approach can enhance the code summarization accuracy from 20.80% to 40.64% from the aspect of BLEU.

To sum up, the following give the major contributions of this work:

- As far as I know, this is the first framework for generating scalable semantic-rich comments, i.e., function comments and parameter comments.
- I propose $TranS^3$ to realize the dual tasks (generating function comments and parameter comments simultaneously) based on a transformer-based neural network model, which is constructed by integrating multiple statement groups extracted after program analysis with the transformer model and the copy attention mechanism.
- To evaluate $TranS^3$, I establish a dataset which includes around 110k $\langle UseSet$, *parameter comment*$\rangle$ and $\langle KeySet$, *function comment* $\rangle$ pairs extracted from real-world influential projects. Such dataset can also be used for future relevant research.

- To evaluate the performance of $TranS^3$, a few experiments are performed based on the collected dataset and real-world benchmarks. Form which the results demonstrate that the proposed approach can achieve better performance compared with the baselines.

5.2 Motivating Example

In this section, I use the code snippet and its corresponding comments in Figure 5.2, which are extracted from the Java project "`JodaOrg/joda-time`", as an illustrative example. From this example, we can observe that the code snippet aims to determine the month-day with a specified period added, and its function comment is "returns a copy of this month-day with the specified period added". Most existing comment generation approaches only consider the method comment, which is incomplete to comprehend the intent and the usage of the method under analysis. A complete comment should not only contain the method comment which shows its intent but also the parameter comment which indicates its usage. Besides, to grab the semantic information of this code snippet, not all statements in the code snippet contribute equally to the comment generation as some statements do not include the words in the comment at all. Thus, to grab the main semantic information of the code snippet and remove noise to the model, I propose to select the important statements ($KeySet$) according to several rules as described in Section 5.3.1 for the method comment generation. Furthermore, I also propose to generate the parameter comment to enrich the method documentation. Since the parameter comment is mainly provides the information of the parameter, i.e., its type and usage, only statements where the parameter appears can contribute to the parameter comment generation. Thus, for the parameter comment generation, I refer to the use of variable in data flow analysis and propose to select the $UseSet$ of the parameter. For example, in Figure 5.2, the method intents to return a new month day with a period added. Thus statement 13 "`return new MonthDay(this, newValues)`" tends to provide more semantic information as it contains tokens "return", "month" and "day". The parameter "period" appears in statements 6, 8, 9 and 12, while the parameter "scalar" appears in statements 6 and 12, thus each parameter together with their signature are selected to construct their $UseSet$, respectively. Namely, the $UseSet$ of "period" is $\{s_5, s_6, s_8, s_9, s_{12}\}$ and the $UseSet$ of "scalar" is $\{s_5, s_6, s_{12}\}$ where s_i is the statement at the i-th line in the code snippet.

By utilizing program analysis which selects the $KeySet$ and $UseSet$, the proposed

```
1  /*Returns a copy of this month-day with the specified
     period added.
2  @param period the period to add to this one, null
     means zero
3  @param scalar the amount of times to add, such as -1
     to subtract once
4  @return a copy of this instance with the period added,
      never null */
5  public MonthDay withPeriodAdded(ReadablePeriod period,
      int scalar) {
6     if (period == null || scalar == 0) { return this; }
7     int[] newValues = getValues();
8     for (int i = 0; i < period.size(); i++) {
9        DurationFieldType fieldType = period.
     getFieldType(i);
10       int index = indexOf(fieldType);
11       if (index >= 0) {
12          newValues = getField(index).add(this, index,
     newValues, FieldUtils.safeMultiply(period.getValue(
     i), scalar)); } }
13    return new MonthDay(this, newValues);
14 }
15
```

Figure 5.2: Motivating example

approach constructs the $\langle UseSet, \textit{parameter comment} \rangle$ and $\langle KeySet, \textit{function comment} \rangle$ pairs as an input to the the proposed transformer-based model for method comment and parameter comment generation respectively, which is used to obtain the documentation for the code snippet.

5.3 On Semantic-rich Code Summarization by Vital Code and Transformer-based Model

This section presents $TranS^3$, a semantic-rich code summarization framework that simultaneously generates function comments and parameter comments as a dual task as demonstrated in Figure 5.3. Initially, $TranS^3$ identifies and selects statement groups to construct training data for each task respectively. In particular, for each function, $TranS^3$ constructs a $UseSet$ by collecting statements out of the usage analysis of function parameters and a $KeySet$ out of the statements that can contribute to structuring the overall program semantics. Next, $TranS^3$ forms and inputs the $\langle UseSet, \textit{parameter comment} \rangle$ and $\langle KeySet, \textit{function comment} \rangle$ pairs out of the collected functions, which are further encoded via a transformer layer, respectively. Moreover, to enhance the scalability of $TranS^3$, an identical layer with the copy attention mechanism is injected on

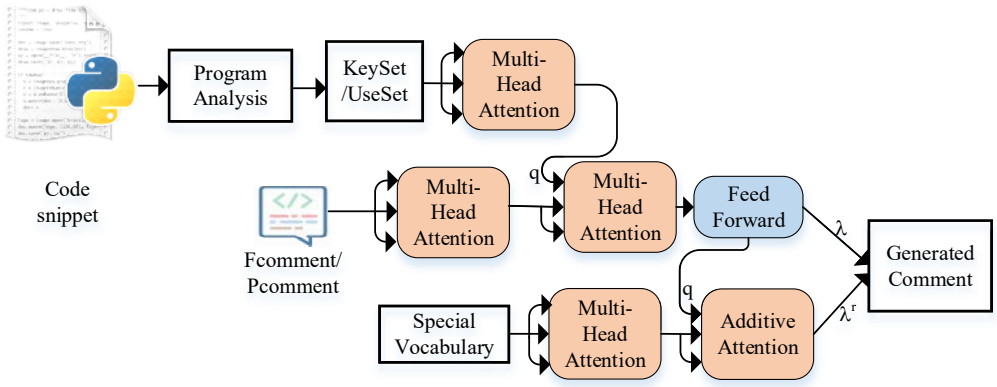


Figure 5.3: The framework overview of the proposed approach $TranS^3$.

top of the transformer layer for generating rare words from the source code. Meanwhile, $TranS^3$ also enables a decoding process for the encoded pairs, i.e., generating their corresponding comments. Such encoding and decoding processes are iterated until the model converges, i.e., the completion of the neural translation modeling. From Figure 5.3, we can know that apart from the the hidden state s from the input source code, the state of the rare word is also obtained by integrating the embedding vectors of the special vocabulary $\mathbf{v}_r$ and s. At last, for each task, $TranS^3$ computes the distribution λ form the encoded code representation and the distribution λ^r from the rare word vocabulary to obtain the final parameter comments or function comments.

5.3.1 Constructing $KeySet$ and $UseSet$

While many existing approaches adopt all statements of a function for training the comment generator, my insight, on the other hand, is that different statements enable different contributions to code summarization. For example, in Figure 5.4, statements 4, 5, 10, 12, 16, 14 and 18 can contribute more than other statements to constructing program semantics since they contain the main tokens in the method name (“factory”, “delete”) which indicate the intent of the code snippet, thus these statements would contribute more to code summarization. More specifically, I infer that a proper selection of statements to form the training data can facilitate a more precise representation of program semantics and the resulting model can further approach the ground truth compared with the traditional code representations. On the other hand, discarding statements that contribute less to program semantics can potentially remove noise for model training and thus enhance its quality. To this end, for generating parameter comments, I propose to select the statements that are semantically relevant to the

parameters, namely $UseSet$. For generating function comments, I propose to select the statements that can significantly contribute to structuring the overall function-level semantics, namely $KeySet$.

5.3.1.1 $UseSet$ selection

Intuitively, the statements that are semantically relevant to parameters are the ones where parameters are used (e.g., assigned, parameterized in other functions). For any parameter, such statements together can indicate its major semantic intent. Thus, in $TranS^3$, such statements are selected and put in the $UseSet$ of a function, which can be extracted by applying the parameter usage analysis. The example shown in Figure 5.4 are utilized to explain the rationale behind $UseSet$ selection. To be specific, as statements 4, 10, 16, and 18 contain the usage to parameter "name", namely, use(name) = {4, 10, 16, 18}, thus they are collected to constitute the $UseSet$ of parameter "name". In contrast, other statements do not contain "name", therefore they are excluded from $UseSet$. From statement 4, we can observe that the operations to parameter "`name`" is that find factory according to "`name`", while from statements 10, 16 and 18 which include "`factory with name =..., name, ...`" or "`Factory name =..., name)`", we can infer that parameter "`name`" is the name of a factory. Thus, the statements in $UseSet$ can provide rich semantic information for the parameter description generation. Different from [92] which only selects partial statements from $UseSet$, $TranS^3$ uses all the statements in $UseSet$ to avoid losing semantic information. To train a deep learning model for accurate parameter comment generation, all the semantic information about the use of parameters should be provided to the model.

5.3.1.2 $KeySet$ selection

The following types of statements are selected for the $KeySet$.

Ending statement. As a function often takes a sequence of statements to accomplish a task, its ending statement is likely to indicate the main intent of the function. Take the code snippet in Figure 5.2 as an example, each statement performs a certain operation, while the last statement `returns the new MonthDay`, which provides the main information of the comment: return month day. Therefore, the ending statement tends to be the core operation in most methods and is the key statement for deducing the intent.

Method call statement. Method call statements play a vital role in constructing function semantics. Usually they can be categorized by whether they are associated with return values. Typically, a method call statement without a return value represents

```
1   public void deleteFactory(String name) {
2       List<FactoryDto> factories;
3       try {
4       factories = findFactory(name);
5       if (factories.isEmpty()) {
6         return;}
7     } catch (NotFoundException e) {
8       return;
9     } catch (ApiException | IOException e) {
10      LOG.error(format("Error of getting info about
       factory with name='%s': %s", name, e.getMessage()),
       e);
11      return;}
12    FactoryDto factory = factories.get(0);
13    try {
14      requestFactory.fromUrl(factoryApiEndpoint +
       factory.getId()).useDeleteMethod().request();
15    } catch (IOException | ApiException e) {
16      LOG.error(format("Error of deletion of factory
       with name='%s': %s", name, e.getMessage()), e);
17      return;     }
18    LOG.info("Factory name='{}' removed", name);}
19
```

Figure 5.4: Example of $KeySet$ and $UseSet$ selection.

an essential set of actions for the associated function, which is likely to be literally represented by its name. On the other hand, a called method with a return value is likely to be assigned to a variable or parameterized in another method, where in either way, such method call statement can facilitate the overall task of the function and thus reflects the semantics for the function intent. For example, in Figure 5.4, lines 4, 5, 12 and 14 are method call statements. Line 4 is a method call returning a value, which assigns the returned value to variable `factories`, and this method call statement provides the source of the `factories`. Line 14 is a method call that does not return a value, where the obtained value is added to the `requestFactory`, and this method call statement is also main action of this function. Therefore, they are added to $KeySet$. Furthermore, if a statement has a method call that has the same action with the function, it is more likely to conduct main action, thus it is an important statement. As an example, Line 14 of `deleteFactory` has the same action with the function, and it is an very important statement in achieving the intended functionality of `delete`. Therefore, such statements are collected to construct $KeySet$.

Assignment statement. An assignment statement assigns data to variables. Such statement usually reflects the variable conversion within a function which constructs its semantic flow. For example, consider the statement on line 12 in `deleteFactory`.

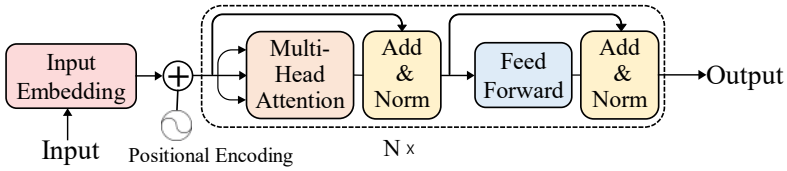


Figure 5.5: The transformer model architecture.

Although this statement is selected by the function call statement rule, little information is known about the variable `factory`. Thus, its data facilitator on line 12, `FactoryDto factory = factories.get(0)` is selected.

Control statement. A control statement (i.e., an statement with one of the if, while, for or switch keywords) controls the execution flow of a function and thus constructs its high-level semantics. For example, consider the method `deleteFactory` in Figure 5.4. Line 6 contains a return statement that is control dependent on line 5, making line 5 an important control statement. The summary content would be inadequate without the control statement.

After collecting the $UseSet$ and $KeySet$ of each function, I further form the ⟨$UseSet$, *function comment*⟩ and the ⟨$UseSet$+$KeySet$, *function comment+parameter comment*⟩ pairs for training the transformer-based neural network model in $TranS^3$.

5.3.2 Encoding $UseSet$ and $KeySet$

$TranS^3$ enables a transformer-based neural machine translation model to encode the collected ⟨$UseSet$, *parameter comment*⟩ and the ⟨$KeySet$, *function comment*⟩ pairs to a hidden space as the code representation. In particular, Is adopt two layers—a transformer layer as a basic encoder and a copy attention layer to alleviate the out-of-vocabulary issue.

In $TranS^3$, although the selected $UseSet$ and $KeySet$ both are associated with hierarchical semantics, they are represented as plain-text sequences of their associated pairs instead of any hierarchical representations for inputting them to the encoder. To illustrate, in general, the selected statements cannot reflect the complete associated function hierarchies where enabling hierarchical representation can easily lead to the misconception of the hierarchical semantics and further compromise the encoding effectiveness.

Based on the input which is composed of three matrices with the same dimension d_k: queries, keys and values, transformer first calculates the dot products of the query vector

with all the key vectors, then the results is divided by $\sqrt{d_k}$, and at last softmax is adopted to calculate the values of weight on the given values. Actually, the matrix Q which is the packing of the attention function on all the queries is computed simultaneously, thus the calculation cost can be reduced. Additionally, matrices K and V represent the packed keys and values, thus the output of the attention matrix is calculated as:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{5.1}$$

The self-attention mechanism of transformer can obtain the relationships among all the tokens of the input sequence. By taking the advantage of weights among the integral tokens, such mechanism can substantially mitigate the long distance dependency problem which is resulted by CNN, i.e., the contributions of the input tokens that from the long range are compromised. Furthermore, the multi-head attention enables the model to estimate in parallel which can dramatically reduce the calculating expense which is resulted by RNN that teh tokens are encoded sequentially.

5.3.2.1 The transformer layer

To improve the representation of the input, Transformer [98] is proposed by adopting the mechanism of self-attention which measures the relevance between one input token and the other tokens in the the input sequence to derive the its vector representation for the ultimate representation of the input sequence. There are N identical layers in transformer, and each layer includes two sub-layers. As shown in Figure 5.5, the mechanism of self-attention is realized in the first sub-layer, and the second sub-layer is just a straightforward feed-forward neural network. At each layer, the multi-head attention mechanism utilizes h attention heads and performs self-attention. Given the embedding of a piece of source code based on word2vec [65], which is signed as $X = (x_1, x_2, ..., x_m)$, each attention head transforms them into the sequence of output vectors $S = (s_1, s_2, ..., s_n)$ according to:

$$\begin{aligned} e_{ij} &= \frac{W^q x_i (W^k x_j)^T}{\sqrt{d_k}} \\ \alpha_{ij} &= \frac{exp\ e_{ij}}{\sum_{k=1}^{m} exp\ e_{ik}} \\ s_i &= \sum_{j=1}^{m} \alpha_{ij} (W^v x_j) \end{aligned} \tag{5.2}$$

where W^q, W^k and W^v are the the parameters that are unique per layer and attention head. More specifically, for vector x_i which is the representation of each token, then its representation is calculated based on the self-attention mechanism: (1) to derive q_i, k_i and v_i, namely the vectors of query, key and value which are marked as W^q, W^k and W^v, it first multiplies the input token vector x_i with a initialized matrix, (2) then by using the dot product of $q_i \cdot k_j$, it calculates the contributions of x_i from the vectors of all the input tokens, where $j \epsilon [1,n]$ and n means the input token number, (3) next, it computes the scores e_{ij} of x_i by $\sqrt{d_k}$ and d_k is k_i's dimension number and normalize the results by softmax to derive the weights of all the input tokens α_{ij}, (4) then it computes an interim vector space $v^{'}$ by calculating the product of these weights and their related value vectors, and (5) at last, it obtains z_i which is the final vector representation of x_i, through calculating the summation of the vectors contained in $v^{'}$. Finally, to get the final vector representation of the input tokens, namely s_i, the feed-forward neural network are adopted to further process the obtained token vectors.

Transformer is adopted to represent (encode) $UseSet$ and $KeySet$ because unlike other approaches [20, 37, 41, 43, 70, 113] where their commonly adopted CNN/RNN can cause long-distance dependency problem and the excessive computing problem, the adopted self-attention in transformer can result in efficient sequence representation. In particular, the multi-head attention mechanism is introduced by transformer to enable the framework to collectively pay attention to different information. Thus the final input representation can be determined by deriving the relevance among the present input token and other tokens in the input sequence by such mechanism. Thus this mechanism can significantly mitigate the long-distance dependency issue, i.e., the contributions of the tokens with long-distance are compromised. Furthermore, the multi-head self-attention mechanism enables the computation in parallel which can dramatically reduce the excessive computational expense resulted in by RNN in which the tokens are encoded sequentially.

5.3.2.2 The copy attention layer

In the encoding process, it is likely that the vocabulary mainly consists of the tokens which appear frequently in the adopted code snippets, while the other rare tokens are left out of the vocabulary and thus compromise the encoding effectiveness. To cope with the out-of-vocabulary issue, it incorporates the copy attention mechanism [89] with transformer to enable the framework to produce words both from the vocabulary and copy from the semantic-relevant rare word in the code snippet. In particular, it uses an

additional attention layer to learn the copy distribution on top of the decoder stack [75].

Let the special vocabulary, V_r, be the union of the rare words which all appear in the source code while not in the common vocabulary. The copy attention layer takes s_i as the query, word embedding v^r from V_r as the input and outputs a_i^r as the attention weights and s_i^r as the context vectors:

$$
\begin{aligned}
e_{ij}^r &= \frac{W^q v_i^r (W^k v_j^r)^T}{\sqrt{d_k}} \\
\alpha_i^r &= \frac{exp\ e_{ij}^r}{\sum_{k=1}^m exp\ e_{ik}^r} \\
s_i^r &= \alpha_i^r e^r
\end{aligned}
\tag{5.3}
$$

where W^s, W^r and b, are learnable parameters. As a result, p^r is the copy distribution over the rare words vocabulary:

$$
p^r(y_t) = \sum_{j:x_j=y_t} a_i^r \tag{5.4}
$$

Now, given a code snippet, it can obtain its code representation and its associated attention, and its out-of-vocabulary word representation and its associated attention respectively.

5.3.3 The decoding process

After the transformer and copy attention layers map the selected statement groups' embedding $(x_1, x_2, ...x_m)$ into the sequence of hidden states s_i and s_i^r respectively, it further adopts the softmax function z to produce the output token based on the hidden states s_i, i.e., at time step t, the conditional probability of y_t, namely the t-th word in the generated comment, according to the language model [17] is:

$$
p(y_t) = p(y_t|y_1, ...y_{t-1}) = softmax(Ws_i + c) \tag{5.5}
$$

where the bias vector c and weight matrix W are the learnable parameters.

As a result, the final distribution of the generated comment word y_t is defined by combining the two distributions, i.e., Equations 5.5 and 5.4, as:

$$
\begin{aligned}
P(y_t) &= \lambda^r p^r(y_t) + \lambda p(y_t) \\
\lambda, \lambda^r &= softmax(W^n[s_i; s_i^r] + b^n)
\end{aligned}
\tag{5.6}
$$

where W^n and b^n are learnable parameters.

Based on the adopted encoding-decoding framework, given the inputs $\langle UseSet$, *parameter comment*$\rangle$ and the $\langle KeySet$, *function comment*$\rangle$ pairs, the loss function of the model in $TranS^3$ to generate parameter and function comments is designed as:

$$
\begin{aligned}
L(\theta_f) &= -\sum_t log(P_f(y_{ft})) \\
L(\theta_p) &= -\sum_t log(P_p(y_{pt}))
\end{aligned}
\tag{5.7}
$$

where $P_f(y_{ft})$ and $P_p(y_{pt})$ are the probability distributions to generate the function comment y_f and parameter comment y_p at time step t respectively, θ_p and θ_f are the set of all learnable parameters.

5.4 Evaluation

In this section, to extensively demonstrate the effectiveness and efficiency of the proposed $TranS^3$, a set experimental study is conducted. In particular, a dataset based on real-world representative projects is constructed for conducting experiments and select state-of-the-art approaches for performance comparison.

5.4.1 Experimental studies

5.4.1.1 Benchmark construction

Dataset. It is essential to collect a set of $\langle UseSet$, *parameter comment*$\rangle$ and the $\langle KeySet$, *function comment*$\rangle$ pairs as the training data and ground truth prior to launching a convincing experimental study. To this end, I propose to construct the dataset from influential projects in GitHub. Specifically, to ensure the quality of the dataset, I first collect the Java repositories with more than 5000 stars and obtain a total of 379 projects. To avoid the duplication of the dataset items, only one item is utilized for the source code with the same function header. Then, among the selected projects, I select their methods with parameters, and extract the first sentences of their Javadocs as the function comment and the sentences beginning with "@param" as the parameter comments. As a result, around 110k function and parameter comments are collected. Table 5.1 presents the statistic analyses of the dataset, which includes the statement

number in each method, the token number in each statement, the token number in each function comment and the token number of parameter comment in each function considering the three part of the dataset respectively. From the first part of the table we can observe that each method includes 1 to 20 statement in all splits of the dataset. The second part of the table demonstrates the distribution of the token number in each statement which shows that most statements contain 1 to 40 tokens. The distribution of the token number of each code comment is shown in the third part, and from which we can know that most function comments include 1 to 20 words. Moreover, the fourth part shows that most parameter comments contain 1 to 30 tokens.

Baseline. To estimate the effectiveness of the proposed approach, some existing code summarization approaches are collected. They include *CodeNN* [43], *CoaCor* [113], *Rencos* [120] and *TL_CodeSum* [42], among which *CodeNN* In particular, *CodeNN* [43] is the first comment generation approach of source code via neural network. It is a LSTM-based encoder-decoder framework that encodes source code to context vectors with the attention mechanism and decode them to produce summaries. *CoaCor* [113] adopts the plain sequence of the program code and realizes code summarization by using the seq2seq framework based on LSTM. Consider both NMT-based and retrieval-based approaches, *Rencos* [120] is proposed. Given the program code, *Rencos* first retrieves the dataset to find the most similar code, then the input and the two source code retrieved are encoded, and eventually fuses them during decoding to generate the summary. *TL_CodeSum* [42] realizes comment generation based on a multi-encoder which encodes the API sequences together with the sequences of program and then the annotation is produced based on the transferred API knowledge.

In addition, I also design the evaluation to explore the performance of its components. For the baseline, they are combined with program analysis to explore the efficiency of program analysis adopted in the proposed approach, and they are marked as $CodeNN_{PA}$, $CoaCor_{PA}$ and $TL\_CodeSum_{PA}$ respectively.

For the proposed approach $TranS^3$, I utilize different components. Specifically, ${TranS^3}_{base}$ adopts the transformer encoder with the OpenNMT framework [47]; ${TranS^3}_{PA}$ add the program analysis component to ${TranS^3}_{base}$; ${TranS^3}_{copy}$ adopts the copy attention mechanism with ${TranS^3}_{base}$; and the complete $TranS^3$ which integrates the program analysis component and the copy attention mechanism.

Table 5.1: The statistics of dataset.

		training data	validation data	testing data
Statement number distribution in each method	(1, 10)	88762	9760	9736
	(10, 20)	1323	130	146
	(20, 30)	201	18	24
	(30, 40)	56	6	7
	(40, 50)	21	1	4
	>50	28	4	2
Token number distribution in each statement	(2, 20)	53722	5943	6056
	(20, 40)	16240	1786	1707
	(40, 60)	7286	751	781
	(60, 80)	3754	434	402
	(80, 150)	5459	578	562
	>150	3930	427	369
Token number distribution in each function comment	(1, 5)	25895	2811	2843
	(5, 10)	37176	4052	4173
	(10, 20)	26805	3006	2849
	(20, 30)	508	50	52
	>30	7	0	2
Token number distribution in each parameter comment	(1, 10)	49516	5445	5554
	(10, 20)	24808	2717	2701
	(20, 30)	10107	1070	1053
	(30, 40)	3543	413	365
	(40, 50)	1519	175	161
	>50	898	99	85

5.4.1.2 Experimental Setups

In the experiments, I set the word embedding size to 1280, set the the batch size to 2048, set the layer size to 6, and set the head number to 8. I choose the Python 3.5 to conduct all the experiments.

5.4.1.3 Implementation

I implement the proposed approach based on OpenNMT [47]. After obtaining the dataset, it is split to 80%, 10% and 10% for training, validation and testing respectively. Then, the $KeySet$ and $UseSet$ are selected respectively. Next, the code and corresponding comment pairs are inputted to the transformer-based model for training according to the settings in Section 5.4.1.2. Subsequently, the trained model is obtained for the comment generation task and conduct testing based on the model. Three widely-adopted estimation metrics in the research area of natural language processing are utilized: BLEU [77], METEOR [12], and ROUGE [58]. These metrics mainly calculate the similarity degree between the generated natural language text and the ground-truth by measuring the frequency of the tokens occurrence in both of them from different aspects. As comment generation is also a text generation task in which the natural language words composed the output, thus these metrics are adopted to assess the accuracy of the generated code annotation. Particularly, the most famous famous evaluate metric adopted in the natural language generation task, BLEU, estimates the n-gram accuracy by comparing with a few reference sentences. METEOR considers the stemming and synonymy matching and estimates the extent to which the results grab the content based on the references. ROUGE-L calculates the similarity in sentence level by matching the longest common sequence between two text sequence.

5.4.2 Analysis of the Result

In this section, the experimental results is presented in Table 5.2 and then I analysis the results through the following research questions. Table 5.2 demonstrates the results of method and parameter comment generation of all the pre-described approaches in terms of the selected metrics. Note that to evaluate the parameter comment generating performance, I also use the pairs of ⟨ code snippet, parameter comment ⟩ as the input of the sate-of-the-art approaches to train their model to generate parameter comment for comparison. Besides, as the parameter comment begins with "@param", to evaluate the performance of the actual content about the generated parameter comment, I calculate the value of the selected metrics by discarding "@param".

RQ1: What's the effectiveness of the proposed approach compared to the state-of-the-art approaches? From the result shown in Table 5.2 we can observe that while the compared approaches achieve the performances from 9.0% to 39.2% for the method comment generation and 7.0% to 19.5% for the parameter comment generation

Table 5.2: Code summarization results comparison with Baselines.

Approaches	BLEU	ROUGE-L	METEOR	Precision
	Method comment			
CodeNN	13.5	40.17	15.12	55.23
CoaCor	13.1	41.5	16.09	50.92
TL_CodeSum	17.7	40.31	16.89	45.73
Rencos	32.1	50.26	24.15	54.97
CodeNN$_{PA}$	16.3	36.05	15.65	49.31
CoaCor$_{PA}$	19.9	40.9	18.9	48.37
TL_CodeSum$_{PA}$	28.7	42.432	21.64	46.27
Rencos$_{PA}$	41.4	54.78	29.76	58.81
$TranS^3{}_{base}$	38.83	44.42	23.89	47.81
$TranS^3{}_{copy}$	38.86	44.14	23.58	48.05
$TranS^3{}_{PA}$	47.31	54.77	30.64	59.18
$TranS^3$	**47.12**	**54.69**	**30.49**	**58.80**
	Parameter comment			
CodeNN	8.01	28.00	11.62	40.43
CoaCor	11.3	34.28	14.39	40.77
TL_CodeSum	16.8	33.69	15.71	36.74
Rencos	19.5	34.48	16.83	37.25
CodeNN$_{PA}$	14.6	35.55	14.74	52.37
CoaCor$_{PA}$	19.8	46.05	19.79	52.65
TL_CodeSum$_{PA}$	32.3	49.37	25.12	51.83
Rencos$_{PA}$	36.8	49.83	27.17	51.00
$TranS^3{}_{base}$	43.35	56.65	34.58	72.33
$TranS^3{}_{copy}$	46.73	58.23	36.58	74.05
$TranS^3{}_{PA}$	55.38	68.63	41.47	78.12
$TranS^3$	**56.59**	**69.74**	**42.62**	**77.23**

from the aspect of BLEU, $TranS^3$ can approximate 47.12% and 56.59% respectively. In particular, consider the method comment generation, $TranS^3$ can surpass the existing state-of-the-art approaches from 20.20% to 80.90% from the aspect of BLEU. This result can determine the outstanding of $TranS^3$. First, the programming-analysis-based $UseSet$ and $KeySet$ selection can obtain the main semantic information of the parameter usage and the code snippet intent and discard the unimportant information which may add noise to the comment generation model. Second, the self-attention mechanism that adopted in transformer can also contribute to the performance improvement because the adopted self-attention mechanism can grab the influences of overall sequence on all the tokens of the selected $UseSet$ and $KeySet$ for better showing the semantics of the inputs and thus can optimize the language model weights. Besides, for the parameter comment generation, $TranS^3$ can achieve better performance compared with the baselines from 65.54% to 85.86% from the aspect of BLEU, which further prove the effectiveness of the proposed approach.

RQ2: What is the effectiveness of the main components of the proposed approach? Two main components of the proposed approach are (1) programming-analysis-based $UseSet$ and $KeySet$ selection and (2) copy-attention-integrated transformer. To explore the effectiveness of them, different experiments are designed. First, the proposed approach with different components are based copy-attention-integrated transformer, while in the state-of-the-art RNN/LSTM-based approaches, RNN or LSTM is utilized. From the results we can observe that except *Rencos* which is not a simple RNN/LSTM-based approach for it retrieves the similar code snippets that can provide more information for comment generation, the proposed approach with different components can always outperform the RNN/LSTM-based approaches consider both method comment and parameter comment generation. This result comparison proves that the adopted copy-attention-integrated transformer is a effective component for comment generation. To verify the effectiveness of the proposed programming-analysis-based $UseSet$ and $KeySet$ selection component, it is first integrated with the sate-of-the-art deep-learning-based approaches, i.e. $CodeNN_{PA}$, $CoaCor_{PA}$, $TL\_CodeSum_{PA}$ and $Rencos_{PA}$ in Table 5.2. From the results we can observe that the performance of $TL\_CodeSum_{PA}$ and $Rencos_{PA}$ can be improved consider method comment generation and the performance of all the approaches can be improved observably consider the parameter comment generation. On the other hand, the proposed complete approach, i.e., $TranS^3$ and ${TranS^3}_{PA}$ who also utilizes the programming-analysis-based selection component also outperform the approach without the programming-analysis-based selection component dramatically

for both method comment and parameter comment generation. These results verify that the proposed programming-analysis-based $UseSet$ and $KeySet$ selection component can really retain the main semantic information of the code snippets for comment generation and filter the useless information to avoid add noise to the comment generation model that will influence the performance. It is mainly because that the statement selection component can choose the most important statements which include more semantic information about the intent of the source code, and it discards the useless statements which may be noise information for comment generation. On the other hand, the copy-attention-integrated transformer can learn the semantic information of the source code as the self-attention mechanism can learn the input information (the source code) better and the copy mechanism can help generate the words that is rare.

5.5 Threat to Validity

For the proposed approach and the results, there are several threats to validity, which are described as below. The first threat to validity is that I only consider the dataset of java source code to estimate the proposed framework. Although Java is not representative of all other programming languages, the adopted java dataset is large enough to prove the effectiveness of the proposed model. Besides, $TranS^3$ can be used to the generation of comment for other programming languages easily. Secondly, it can be inadequate to only rely on the limited number of token-counting-based metrics to fully evaluate the effectiveness of $TranS^3$. To reduce this threat, I first adopt multiple evaluation metrics, i.e., BLEU, METEOR, ROUGE, and Precision to estimate the quality of the generated code comments. The result that $TranS^3$ significantly outperforms other compared approaches under all the selected metrics indicate that $TranS^3$ is able to generate high-quality code comments.

5.6 Conclusion

In this work, I propose a transformer-based neural network model which combines programming analysis and transformer-based neural machine translation model for both method comment and parameter comment generation. Specifically, to abstract the main semantic information and discard the useless noise information for the dual tasks, it first designs a programming-analysis-based component to extract use statement set of the parameter ($UseSet$) and the important statement set in the code snippet ($KeySet$)

for code representation. Next, the copy-attention-integrated transformer based NMT (Neural Machine Translation) framework is utilized for both the method comment and parameter comment generation to provide complete java documentation for the code snippets. To estimate the performance of the the proposed approach, a set of experimental studies are performed, where the results prove that the proposed approach can achieve better performance significantly compared with the baseline approaches.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Code summarization generate the annotation of source code automatically by utilizing different techniques, such as, information retrieval, deep learning. Recently, with the development of deep learning and its widely utilization in different areas, neural machine translation structure has been introduced to the research of code summarization. However, due to the special grammar and syntax structure of the programming language and various shortcomings of different deep neural networks, the accuracy of the existing code summarization approaches is not high enough for practical application. Thus, the research on the representation of the source code and the improvement of the neural network have attracted more and more attention. Considering this status, this paper mainly propose three code summarization approaches: 1) The DRL-guided code summarization via hierarchical attention; 2) A Transformer-based Generative Adversarial Network framework for universal code summarization; 3) On semantic-rich code summarization by vital code and Transformer-based model. The main research work of this paper are as follows.

The comment generation based on DRL and hierarchical attention.

Through introducing different representations of the source code which includes the plain text, the sequenced abstract syntax tree and the sequenced control flow graph, this work proposes the first hierarchical-attention-based approach to show the hierarchical structure of source code by utilizing the attention mechanism at the statement level

and token level respectively. When constructing the code representation, this approach pays "attention" to tokens and statements to obtain the hierarchical structure of the source code. To estimate the effectiveness of the proposed approach, comprehensive experiments are conducted based on the collected real-world java dataset, and the results prove that the proposed approach can outperform competitive baselines according to multiple commonly utilized metrics.

A Transformer-based Generative Adversarial Network framework for universal code summarization. It proposes to utilize the generative adversarial network framework based on transformer for universal code summarization by injecting the syntactic structure of the source code in this work. Specifically,it first builds a universal hierarchical semantic (UHS) model for reflecting the hierarchical semantics of the source code for multiple programming languages. Accordingly, it further constructs a tree-transformer for encoding the derived source code representation. Furthermore, it utilizes the GAN framework by adopting the tree-transformer based neural machine translation model as its generative model. The discriminator model is utilized to feed back a reward to the generative model iteratively to tune the generator to improve the accuracy of the generated comment. We conduct a set of experimental studies under multiple mainstream high-level programming language datasets (Python, Java and C#) to estimate the performance of the proposed approach, and the results suggest that the proposed approach can achieve better result compared with several existing approaches. Besides, a few of case studies are performed by issuing the comment generated by the proposed approach to the developers to ask for their feedback. As a result, none of the feedback are negative.

On semantic-rich code summarization by vital code and Transformer-based model. This work proposes a transformer-based neural network model which combines programming analysis and transformer-based neural machine translation model for both method comment and parameter comment generation. Specifically, to abstract the main semantic information and discard the useless noise information for the dual tasks, the proposed approach first designs a programming-analysis-based component to extract use set of parameter ($UseSet$) and the important statements in the code snippet ($KeySet$) for code representation. Next, the copy-attention-integrated transformer based NMT (Neural Machine Translation) framework is utilized for both the method comment and parameter comment generation to provide integrity java documentation for the code snippets. A few of experimental studies are conducted to estimate the performance of the proposed approach, where the obtained results prove that the proposed approach can

achieve better performance compared with the existing approaches dramatically.

6.2 Future Work

With the development of Deep Learning technique, the research focus on deep-learning-based code summarization will still be a research hotpot in the future. This paper only proposes three code summarization approaches considering different features for different special scenes. Although some achievements have been obtained, there is still room for future improvement, which need more comprehensive research in the future. The main research focus in the future are concluded as follows.

The accuracy. Till now, the accuracy of the existing code summarization approaches is still very low for practical application. Thus, more attention should be paid to this issue.

Explainability. As the deep-learning-based approaches have shown high accuracy compared to the template-based approaches and the information-retrieval-based approaches, thus more attention will be paid to this direction. However, as I know, the deep neural network is just like a black box, the working mechanism inside it is still unexplainable. Therefore, more attention should be paid to the research about the explainability of code summarization, which is also a research hotpot in the deep learning area. Besides, the solving of this problem can help us explore more effective approach for code summarization.

Domain specific language. Considering the particularity of programming language and the syntax diversities of different programming languages, thus in the future, a universal code summariztion approach can be researched by introducing a domain specific language (DSL), which can translate source code in different programming languages to the defined DSL, and then a code summarization model can be used to obtain the source code comment. This DSL can be explored as the bridge between natural language and different programming languages, based on which the universal code summarization approach can be obtained for different programming languages.

The evaluation metric. As the generated comment is natural language sentence, all research utilize the natural language process evaluation metrics (like BLEU, METER, ROUGE) to evaluate the accuracy of the results. However, as the particularity of the code summariztion task, the generated comment obtained low BLEU score may also can describe the intent of the source code completely. For example, some projects or programmer may have their own comment styles by which some special words are

added which may be useless for the understanding of the source code, while the general comment generation approach cannot generate these words for special style.

Analyse the incorrectness for better solution. Another research direction can pay attention to the analysis of the incorrectness of the generated comment. I have conducted some analysis and categorised the incorrectness into four types: 1) Inconsistent, 2) Missing phrases, 3) Redundant (duplicated words), 4) Grammar mistakes (broken sentences). Then, I try to identify the root causes of different incorrectness by examining the source code, which will be helpful for looking for solutions to different problems.

Applying to other topics in software engineering. As code summarization has achieve certain performance, we could apply the results and technique for more further research in the software engineering area. For example, code search, code generation. For code search, we can utilize the generated code comment as auxiliary information of the code snippet for retrieval. Besides, the research about different code representations can be utilized directly. For the code generation task, as it is the dual task of the comment generation task, we can utilize the explored comment generation framework to solve the code generation problem.

These are the future research direction according to my thought according to my research experience on code summarization. Furthermore, with the development of different techniques and more in-depth research on code summarization, the future research direction and focus will become clearer and clearer.

BIBLIOGRAPHY

[1] *Abstract syntax trees*.
Accessed 16 August 2018. https://docs.python.org/2/library/ast.html.

[2] *Github*, https://github.com/.

[3] N. J. ABID, N. DRAGAN, M. L. COLLARD, AND J. I. MALETIC, *Using stereotypes in the automatic generation of natural language summaries for c++ methods*, in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 561–565.

[4] P. ABRAHAMSSON, O. SALO, J. RONKAINEN, AND J. WARSTA, *Agile software development methods review and analysis*, arXiv preprint arXiv:1709.08439, (2002).

[5] W. U. 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 (ACL), 2020, pp. 4998–5007.

[6] A. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers principles, techniques, and tools*, 1985.

[7] S. A. AKBAR AND A. C. KAK, *Scor: source code retrieval with semantics and order*, in Proceedings of the 16th International Conference on Mining Software Repositories, IEEE Press, 2019, pp. 1–12.

[8] M. ALLAMANIS, H. PENG, AND C. SUTTON, *A convolutional attention network for extreme summarization of source code*, in International Conference on Machine Learning, 2016, pp. 2091–2100.

[9] U. ALON, S. BRODY, O. LEVY, AND E. YAHAV, *code2seq: Generating sequences from structured representations of code*, arXiv preprint arXiv:1808.01400, (2018).

[10] S. BADIHI AND A. HEYDARNOORI, *Crowdsummarizer: Automated generation of code summaries for java programs through crowdsourcing*, IEEE Software, 34 (2017), pp. 71–80.

[11] D. BAHDANAU, K. CHO, AND Y. BENGIO, *Neural machine translation by jointly learning to align and translate*, (2015).

[12] 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, vol. 29, 2005, pp. 65–72.

[13] A. BANSAL, S. HAQUE, AND C. MCMILLAN, *Project-level encoding for neural source code summarization of subroutines*, in 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, 2021, pp. 253–264.

[14] A. V. M. BARONE AND R. SENNRICH, *A parallel corpus of python functions and documentation strings for automated code documentation and code generation*, arXiv preprint arXiv:1707.02275, (2017).

[15] A. V. M. BARONE AND R. SENNRICH, *A parallel corpus of python functions and documentation strings for automated code documentation and code generation*, arXiv preprint arXiv:1707.02275, (2017).

[16] E. T. BARR, M. HARMAN, P. MCMINN, M. SHAHBAZ, AND S. YOO, *The oracle problem in software testing: A survey*, IEEE Transactions on Software Engineering, 41 (2015), pp. 507–525.

[17] Y. BENGIO, R. DUCHARME, P. VINCENT, AND C. JAUVIN, *A neural probabilistic language model*, Journal of machine learning research, 3 (2003), pp. 1137–1155.

[18] S. BOWMAN, G. ANGELI, C. POTTS, AND C. MANNING, *A large annotated corpus for learning natural language inference*, in EMNLP, 2015, pp. 632–642.

[19] R. P. BUSE AND W. R. WEIMER, *Learning a metric for code readability*, IEEE Transactions on Software Engineering, 36 (2009), pp. 546–558.

[20] Q. CHEN AND M. ZHOU, *A neural framework for retrieval and summarization of source code*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 826–831.

[21] R. COLLOBERT AND J. WESTON, *A unified architecture for natural language processing: Deep neural networks with multitask learning*, in Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), 2008, pp. 160–167.

[22] L. F. CORTÉS-COY, M. LINARES-VÁSQUEZ, J. APONTE, AND D. POSHYVANYK, *On automatically generating commit messages via summarization of source code changes*, in 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2014, pp. 275–284.

[23] K. A. DAWOOD, K. Y. SHARIF, AND K. WEI, *Source code analysis extractive approach to generate textual summary*, Journal of Theoretical and Applied Information Technology, 95 (2017), pp. 5765–5777.

[24] S. C. B. DE SOUZA, N. ANQUETIL, AND K. M. DE OLIVEIRA, *A study of the documentation essential to software maintenance*, in Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information, ACM, 2005, pp. 68–75.

[25] J. DEVLIN, M.-W. CHANG, K. LEE, AND K. TOUTANOVA, *Bert: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805, (2018).

[26] B. P. EDDY, J. A. ROBINSON, N. A. KRAFT, AND J. C. CARVER, *Evaluating source code summarization techniques: Replication and expansion*, in IEEE International Conference on Program Comprehension, 2013, pp. 13–22.

[27] L. FAN, T. SU, S. CHEN, G. MENG, Y. LIU, L. XU, G. PU, AND Z. SU, *Large-scale analysis of framework-specific exceptions in android apps*, in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 408–419.

[28] J. FOWKES, P. CHANTHIRASEGARAN, R. RANCA, M. ALLAMANIS, M. LAPATA, AND C. SUTTON, *Autofolding for source code summarization*, IEEE Transactions on Software Engineering, 43 (2017), pp. 1095–1109.

[29] A. GHANBARI, S. BENTON, AND L. ZHANG, *Practical program repair via bytecode mutation*, in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, 2019, pp. 19–30.

[30] I. GOODFELLOW, J. POUGET-ABADIE, M. MIRZA, B. XU, D. WARDE-FARLEY, S. OZAIR, A. COURVILLE, AND Y. BENGIO, *Generative adversarial nets*, in Advances in neural information processing systems, 2014, pp. 2672–2680.

[31] E. GRAVE, A. JOULIN, AND N. USUNIER, *Improving neural language models with a continuous cache*, arXiv preprint arXiv:1612.04426, (2016).

[32] A. GRAVES, *Generating sequences with recurrent neural networks*, arXiv preprint arXiv:1308.0850, (2013).

[33] X. GU, H. ZHANG, D. ZHANG, AND S. KIM, *Deep api learning*, in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 631–642.

[34] L. GUERROUJ, D. BOURQUE, AND P. C. RIGBY, *Leveraging informal documentation to summarize classes and methods in context*, in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, IEEE, 2015, pp. 639–642.

[35] HAIDUC, SONIA, A. JAIRO, AND M. ANDRIAN, *Supporting program comprehension with source code summarization*, in Proceedings of the 32th IEEE/ACM International Conference on Software Engineering, ACM, 2010, pp. 223–226.

[36] 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, IEEE, 2010, pp. 35–44.

[37] T. HAIJE, B. O. K. INTELLIGENTIE, E. GAVVES, AND H. HEUER, *Automatic comment generation using a neural translation model*, Inf. Softw. Technol, 55 (2016), pp. 258–268.

[38] M. HAMMAD, A. ABULJADAYEL, AND M. KHALAF, *Summarizing services of java packages*, Lecture Notes on Software Engineering, 4 (2016), pp. 129–132.

[39] J. HENKEL, S. K. LAHIRI, B. LIBLIT, AND T. REPS, *Code vectors: Understanding programs through embedded abstracted symbolic traces*, (2018), pp. 163–174.

[40] S. HOCHREITER AND J. SCHMIDHUBER, *Long short-term memory*, Neural computation, 9 (1997), pp. 1735–1780.

[41] X. HU, G. LI, X. XIA, D. LO, AND Z. JIN, *Deep code comment generation*, in Proceedings of the 26th Conference on Program Comprehension, ACM, 2018, pp. 200–210.

[42] X. HU, G. LI, X. XIA, D. LO, S. LU, AND Z. JIN, *Summarizing source code with transferred api knowledge*, in International Joint Conference on Artificial Intelligence 2018, International Joint Conferences on Artificial Intelligence, 2018, pp. 2269–2275.

[43] 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 (1), 2016, pp. 2073–2083.

[44] Z. JI, Z. LU, AND H. LI, *An information retrieval approach to short text conversation*, arXiv preprint arXiv:1408.6988, (2014).

[45] M. KAJKO-MATTSSON, *A survey of documentation practice within corrective maintenance*, Empirical Software Engineering, 10 (2005), pp. 31–55.

[46] N. KALCHBRENNER, L. ESPEHOLT, K. SIMONYAN, A. V. D. OORD, AND K. KAVUKCUOGLU, *Neural machine translation in linear time*, arXiv preprint arXiv:1610.10099, (2016).

[47] G. KLEIN, Y. KIM, Y. DENG, J. SENELLART, AND A. RUSH, *Opennmt: Open-source toolkit for neural machine translation*, in Proceedings of ACL 2017, System Demonstrations, Association for Computational Linguistics, 2017, pp. 67–72.

[48] G. KLEIN, Y. KIM, Y. DENG, J. SENELLART, AND A. M. RUSH, *Opennmt: Open-source toolkit for neural machine translation*, in Proceedings of ACL 2017, System Demonstrations, 2017, pp. 67–72.

[49] V. R. KONDA AND J. N. TSITSIKLIS, *Actor-critic algorithms*, in Advances in neural information processing systems, 2000, pp. 1008–1014.

[50] 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.

[51] A. LECLAIR, S. JIANG, AND C. MCMILLAN, *A neural model for generating natural language summaries of program subroutines*, in Proceedings of the 41st International Conference on Software Engineering, IEEE Press, 2019, pp. 795–806.

[52] A. LECLAIR, S. JIANG, AND C. MCMILLAN, *A neural model for generating natural language summaries of program subroutines*, (2019), pp. 795–806.

[53] Y. LECUN, Y. BENGIO, AND G. E. HINTON, *Deep learning*, Nature, 521 (2015), pp. 436–444.

[54] J. LI, Y. LI, G. LI, X. HU, X. XIA, AND Z. JIN, *Editsum: A retrieve-and-edit framework for source code summarization*, in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 155–166.

[55] J. LI, D. XIONG, Z. TU, M. ZHU, Z. MIN, AND G. ZHOU, *Modeling source syntax for neural machine translation*, arXiv preprint arXiv:1705.01020, (2017).

[56] X. LI, W. LI, Y. ZHANG, AND L. ZHANG, *Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization*, in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA., 2019, pp. 169–180.

[57] B. P. LIENTZ AND E. B. SWANSON, *Software maintenance management*, IEE Proceedings E Computers and Digital Techniques Transactions on Software Engineering, 127 (1980).

[58] C. Y. LIN, *Rouge: A package for automatic evaluation of summaries*, Text Summarization Branches Out, pp. 74–81.

[59] Z. LIU, X. XIA, A. E. HASSAN, D. LO, Z. XING, AND X. WANG, *Neural-machine-translation-based commit message generation: how far are we?*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 373–384.

[60] A. LOUIS, S. K. DASH, E. T. BARR, AND C. SUTTON, *Deep learning to detect redundant method comments*, arXiv preprint arXiv:1806.04616, (2018).

[61] M. MALHOTRA AND J. K. CHHABRA, *Class level code summarization based on dependencies and micro patterns*, in 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), IEEE, 2018, pp. 1011–1016.

[62] P. W. MCBURNEY, C. LIU, C. MCMILLAN, AND T. WENINGER, *Improving topic model source code summarization*, in Proceedings of the 22nd international conference on program comprehension, 2014, pp. 291–294.

[63] P. W. MCBURNEY AND C. MCMILLAN, *Automatic documentation generation via source code summarization of method context*, in Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 279–290.

[64] S. MCCONNELL, *Code complete*, Pearson Education, 2004.

[65] T. MIKOLOV, K. CHEN, G. CORRADO, AND J. DEAN, *Efficient estimation of word representations in vector space*, arXiv preprint arXiv:1301.3781, (2013).

[66] A. MNIH AND Y. W. TEH, *A fast and simple algorithm for training neural probabilistic language models*, arXiv preprint arXiv:1206.6426, (2012).

[67] J. MOORE, B. GELMAN, AND D. SLATER, *A convolutional neural network for language-agnostic source code summarization*, in Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, 2019, pp. 15–26.

[68] L. MORENO, J. APONTE, G. SRIDHARA, A. MARCUS, L. POLLOCK, AND K. VIJAY-SHANKER, *Automatic generation of natural language summaries for java classes*, in 2013 21st International Conference on Program Comprehension (ICPC), IEEE, 2013, pp. 23–32.

[69] L. MORENO, G. BAVOTA, M. DI PENTA, R. OLIVETO, A. MARCUS, AND G. CANFORA, *Arena: an approach for the automated generation of release notes*, IEEE Transactions on Software Engineering, 43 (2016), pp. 106–127.

[70] L. MORENO AND A. MARCUS, *Automatic software summarization: the state of the art*, in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 511–512.

[71] D. MOVSHOVITZ-ATTIAS AND W. W. COHEN, *Natural language models for predicting programming comments*, in Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), vol. 2, 2013, pp. 35–40.

[72] A. NARAYANAN, M. CHANDRAMOHAN, R. VENKATESAN, L. CHEN, L. YANG, AND S. JAISWAL, *graph2vec: Learning distributed representations of graphs*, arXiv preprint arXiv:1707.05005, (2017).

[73] N. NAZAR, H. JIANG, G. GAO, T. ZHANG, X. LI, AND Z. REN, *Source code fragment summarization with small-scale crowdsourcing based features*, Frontiers of Computer Science, 10 (2016), pp. 504–517.

[74] K. NGUYEN, H. D. III, AND J. BOYD-GRABER, *Reinforcement learning for bandit neural machine translation with simulated human feedback*, in Conference on Empirical Methods in Natural Language Processing, 2017, pp. 1464–1474.

[75] K. NISHIDA, I. SAITO, K. NISHIDA, K. SHINODA, A. OTSUKA, H. ASANO, AND J. TOMITA, *Multi-style generative reading comprehension*, arXiv preprint arXiv:1901.02262, (2019).

[76] Y. ODA, H. FUDABA, G. NEUBIG, H. HATA, S. SAKTI, T. TODA, AND S. NAKAMURA, *Learning to generate pseudo-code from source code using statistical machine translation (t)*, in Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, 2015, pp. 574–584.

[77] 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 on association for computational linguistics, Association for Computational Linguistics, 2002, pp. 311–318.

[78] T. A. PIRINEN, *Neural and rule-based finish nlp models-expectation, experiments and experiences*, in The fifth Workshop on Computational Linguistics for Uralic Languages, 2019, pp. 104–1114.

[79] E. M. PONTI, R. REICHART, A. KORHONEN, AND I. VULIC, *Isomorphic transfer of syntactic structures in cross-lingual nlp*, in The 56th Annual Meeting of the Association for Computational Linguistics, 2018, pp. 1531–1542.

[80] J. POUGET-ABADIE, D. BAHDANAU, B. V. MERRIENBOER, K. CHO, AND Y. BENGIO, *Overcoming the curse of sentence length for neural machine translation using automatic segmentation*, arXiv preprint arXiv:1409.1257, (2014).

[81] RABINOVICH, MAXIM, S. MITCHELL, AND D. KLEIN, *Abstract syntax networks for code generation and semantic parsing*, arXiv preprint arXiv:1704.07535, (2017).

[82] M. M. RAHMAN, C. K. ROY, AND I. KEIVANLOO, *Recommending insightful comments for source code using crowdsourced knowledge*, in 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2015, pp. 81–90.

[83] S. RAI, T. GAIKWAD, S. JAIN, AND A. GUPTA, *Method level text summarization for java code using nano-patterns*, in 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2017, pp. 199–208.

[84] M. RANZATO, S. CHOPRA, M. AULI, AND W. ZAREMBA, *Sequence level training with recurrent neural networks*, arXiv preprint arXiv:1511.06732, (2015).

[85] P. C. RIGBY AND M. P. ROBILLARD, *Discovering essential code elements in informal documentation*, in 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 832–841.

[86] P. RODEGHERO, C. MCMILLAN, P. W. MCBURNEY, N. BOSCH, AND S. D'MELLO, *Improving automated source code summarization via an eye-tracking study of programmers*, (2014), pp. 390–401.

[87] R. ROSENFELD, *Two decades of statistical language modeling: Where do we go from here?*, Proceedings of the IEEE, 88 (2000), pp. 1270–1278.

[88] J. SCHMIDHUBER, *Deep learning in neural networks: An overview*, Neural Networks, 61 (2015), pp. 85–117.

[89] 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 (Volume 1: Long Papers), Vancouver, Canada, July 2017, Association for Computational Linguistics, pp. 1073–1083.

[90] H. SHI, H. ZHOU, J. CHEN, AND L. LI, *On tree-based neural sentence modeling*, arXiv preprint arXiv:1808.09644, (2018).

[91] 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.

[92] G. SRIDHARA, L. POLLOCK, AND K. VIJAY-SHANKER, *Generating parameter comments and integrating with method summaries*, in IEEE International Conference on Program Comprehension, 2011.

[93] Y. SUN, S. WANG, Y. LI, S. FENG, X. CHEN, H. ZHANG, X. TIAN, D. ZHU, H. TIAN, AND H. WU, *Ernie: Enhanced representation through knowledge integration*, arXiv preprint arXiv:1904.09223, (2019).

[94] I. SUTSKEVER, O. VINYALS, AND Q. V. LE, *Sequence to sequence learning with neural networks*, in Advances in neural information processing systems, 2014, pp. 3104–3112.

[95] S. H. TAN, J. YI, YULIS, S. MECHTAEV, AND A. ROYCHOUDHURY, *Codeflaws: A programming competition benchmark for evaluating automated program repair tools*, in IEEE/ACM International Conference on Software Engineering Companion, IEEE, 2017, pp. 180–182.

[96] S. H. TAN, H. YOSHIDA, M. R. PRASAD, AND A. ROYCHOUDHURY, *Anti-patterns in search-based program repair*, in Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, 2016, pp. 727–738.

[97] S. THRUN AND M. L. LITTMAN, *Reinforcement learning: An introduction*, IEEE Transactions on Neural Networks, 16 (2005), pp. 285–286.

[98] 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.

[99] E. M. VOORHEES, *The trec question answering track*, Natural Language Engineering, 7 (2001), pp. 361–378.

[100] 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, ACM, 2018, pp. 397–407.

[101] J. WANG, X. XUE, AND W. WENG, *Source code summarization technology based on syntactic analysis*, Journal of Computer Applications, 35 (2015), pp. 1999–2003.

[102] J. WANG, L.-C. YU, K. R. LAI, AND X. ZHANG, *Dimensional sentiment analysis using a regional cnn-lstm model*, in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2016, pp. 225–230.

[103] S. WANG, D. CHOLLAK, D. MOVSHOVITZ-ATTIAS, AND L. TAN, *Bugram: bug detection with n-gram language models*, in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 708–719.

[104] X. WANG, L. POLLOCK, AND K. VIJAY-SHANKER, *Automatically generating natural language descriptions for object-related statement sequences*, in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 205–216.

[105] X. WANG AND B. ZHANG, *Comment generation for source code: State of the art, challenges and opportunities*, arXiv preprint arXiv:1802.02971, (2018).

[106] S. WISEMAN AND A. M. RUSH, *Sequence-to-sequence learning as beam-search optimization*, arXiv preprint arXiv:1606.02960, (2016).

[107] 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), IEEE, 2015, pp. 380–389.

[108] E. WONG, J. YANG, AND L. TAN, *Autocomment: Mining question and answer sites for automatic comment generation*, in Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2013, pp. 562–567.

[109] T. Y. WU AND D. J. ATKIN, *To comment or not to comment: Examining the influences of anonymity and social support on one's willingness to express in online news discussions*, New Media Society, 20 (2018), pp. 4512–4532.

[110] Z. XIANG, J. ZHAO, AND Y. LECUN, *Character-level convolutional networks for text classification*, in International Conference on Neural Information Processing Systems, 2015, pp. 649–657.

[111] Z. YANG, Z. DAI, Y. YANG, J. CARBONELL, R. SALAKHUTDINOV, AND Q. V. LE, *Xlnet: Generalized autoregressive pretraining for language understanding*, arXiv preprint arXiv:1906.08237, (2019).

[112] Z. YANG, D. YANG, C. DYER, X. HE, A. SMOLA, AND E. HOVY, *Hierarchical attention networks for document classification*, in Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL, 2016, pp. 1480–1489.

[113] Z. YAO, J. R. PEDDAMAIL, AND H. SUN, *Coacor: Code annotation for code retrieval with reinforcement learning*, in The World Wide Web Conference, ACM, 2019, pp. 2203–2214.

[114] I. YEN, S. ZHANG, F. BASTANI, AND Y. ZHANG, *A framework for iot-based monitoring and diagnosis of manufacturing systems*, in 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2017, pp. 1–8.

[115] P. YIN AND N. GRAHA, *A syntactic neural model for general-purpose code generation*, in Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 2017, pp. 440–450.

[116] A. T. YING AND M. P. ROBILLARD, *Code fragment summarization*, in Proceedings of the 2013 9th joint meeting on foundations of software engineering, 2013, pp. 655–658.

[117] L. YU, S. XIAOBING, AND L. BIN, *Research on automatic summarization for java packages*, Journal of Frontiers of Computer Science & Technology, 11 (2017), pp. 212–220.

[118] L. YU, W. ZHANG, J. WANG, AND Y. YU, *Seqgan: Sequence generative adversarial nets with policy gradient*, in Proceedings of the AAAI conference on artificial intelligence, vol. 31, 2017, pp. 2852–2858.

[119] J. ZHAI, X. XU, Y. SHI, G. TAO, M. PAN, S. MA, L. XU, W. ZHANG, AND L. TAN, *Cpc: Automatically classifying and propagating natural language comments*

via program analysis, in Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020), IEEE/ACM, 2020, pp. 1359–1371.

[120] J. ZHANG, X. WANG, H. ZHANG, H. SUN, AND X. LIU, *Retrieval-based neural source code summarization*, in Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020), IEEE/ACM, 2020, pp. 1–12.

[121] J. M. ZHANG, L. ZHANG, D. HAO, M. WANG, AND L. ZHANG, *Do pseudo test suites lead to inflated correlation in measuring test effectiveness?*, in 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019, 2019, pp. 252–263.

[122] L. ZHANG, *Hybrid regression test selection*, in Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, pp. 199–209.

[123] S. ZHANG, C. ZHANG, AND M. D. ERNST, *Automated documentation inference to explain failed tests*, in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), IEEE, 2011, pp. 63–72.

[124] H. ZHOU, W. LI, Z. KONG, J. GUO, Y. ZHANG, B. YU, L. ZHANG, AND C. LIU, *Deepbillboard: Systematic physical-world testing of autonomous driving systems*, in Proceedings of the 42nd International Conference on Software Engineering, ICSE, 2020, pp. 347–358.