G-DCS: GCN-Based Deep Code Summary Generation Model

Changsheng Du[1,2], Yong Li[1,2], Ming Wen[2]*

[1] *College of Computer Science and Technology, Xinjiang Normal University, China*
[2] *Xinjiang Electronics Research Institute, China*
xiaodukuko@outlook.com, liyong@live.com, wmconet@126.com

Abstract

In software engineering, software personnel faced many large-scale software and complex systems, these need programmers to quickly and accurately read and understand the code, and efficiently complete the tasks of software change or maintenance tasks. Code-NN is the first model to use deep learning to accomplish the task of code summary generation, but it is not used the structural information in the code itself. In the past five years, researchers have designed different code summarization systems based on neural networks. They generally use the end-to-end neural machine translation framework, but many current research methods do not make full use of the structural information of the code. This paper raises a new model called G-DCS to automatically generate a summary of java code; the generated summary is designed to help programmers quickly comprehend the effect of java methods. G-DCS uses natural language processing technology, and training the model uses a code corpus. This model could generate code summaries directly from the code files in the coded corpus. Compared with the traditional method, it uses the information of structural on the code. Through Graph Convolutional Neural Network (GCN) extracts the structural information on the code to generate the code sequence, which makes the generated code summary more accurate. The corpus used for training was obtained from GitHub. Evaluation criteria using BLEU-n. Experimental results show that our approach outperforms models that do not utilize code structure information.

Keywords: GCN, Summary generation, Deep learning

1 Introduction

In the design or maintenance of software, developers need to expend 59% of their time understanding the meaning of the program. A good summary is very important for program understanding [1]. Previous studies have shown that for most programmers, the quality of code summaries is positively correlated with the speed of understanding code. Developers can quickly understand the meaning of code through natural language description [2]. But code summaries often do not match or are missing due to various factors in the programming process. Automatic code summary generation can quickly generate summary information when only code is available, helping developers faster comprehend the idea of the code and shorten the program development cycle.

Deep learning-based code summary generation techniques are mostly based on the assumption of the naturalness of the code [3]. This category consists of two principal forms of approaches, one based on the classical encoder-decoder model, and the other is a combination of learning algorithms using other types of techniques, such as graph neural networks, reinforcement learning, etc. In 2016 Iyer [4] et al. introduced deep neural networks into the study of automatic code abstract generation. LSTM and attention mechanisms in a seq-to-seq neural machine translation framework to automatically generate summaries for code, thereafter, deep neural network techniques have gradually become the dominant technique in code summarization. Their model is called Code-NN.

A common problem with models based only on traditional encoder-decoder is that they do not make effective use of the structural information on the code itself. In this paper, we present the GCN-based model for deep code digest generation. In contrast to Code-NN, this technique uses a GCN [5] to deal with the structural information on the code and combines it with the semantic information on the code itself to generate a sequence of representations of the code, relying on neural machine translation technology (NMT) [6] to generate a summary of the code.

This paper is divided into six chapters. Chapter 1 introduces the background of code digest generation. Chapter 2 reviews the research results in the field of source code summary generation, and analyzes some problems existing in the existing work. In Chapter 3, aiming at the problems raised in Chapter 2, we propose the G-DCS model and introduce the structure and implementation details of the model. Chapter 4 introduces the data used in the experiment and some parameter settings of the model. Chapter 5 analyzes the experimental data and some factors affecting the experimental results. Chapter 6 summarizes our work and points out the next research direction.

2 Related Work

Chen, et al. proposed a deep learning approach for reliability assessment [7], Yu, et al. proposed a Tree-LSTM to solve the code semantic cloning problem [8], Wang, et al. used GNN to measure the similarity of code pairs [9],

*Corresponding Author: Yong Li; E-mail: liyong@live.com
DOI: 10.53106/160792642023072404014

Wei, et al. used a deep end-to-end model for code cloning detection [10], Qu, et al. used pre-trained models for software defect detection [11], Li, et al. proposed a neural machine translation-based approach to automatically fix errors in code programs [12], Tran, et al. evaluated the performance of imbalance on machine learning for network intrusion [13]. However, it is necessary to develop a direction for using the deep learning method to solve the problem of the automatic generation of code summaries.

The vector features of the source code need though carefully collect various types of data to construct a corpus of relevant domains and update the neural network model variables by supervised or unsupervised training to improve the model's accuracy. Compared with traditional source code encoders based on information extraction techniques or classical machine learning techniques, deep neural network encoders can extract structural and semantic features of the code more accurately.

2.1 Code-based Keyword Extraction Method

Many methods have been invented to generate code summaries over the past ten years. Automatic code summarization research started in 2010, and the initial research mainly used information retrieval techniques [14], for example, VSM [15], LDA [16] and LSI [17]. While there is still room for this technology to grow, there are two limitations that cannot be ignored. First, it is difficult to extract valid keyword information if the code segment is poorly named, and second, such methods are too dependent on the ability to retrieve similar code segments. Since 2016, deep neural networks have developed rapidly, based on deep learning techniques that have gradually replaced based information retrieval techniques as the mainstream techniques for automatic code abstract generation research.

2.2 Deep Learning-based Approach

Iyer [4] et al. first introduced NMT techniques into the research area of code summarization. They combined LSTM neural network and attention mechanism to design a summary generation model based on the encoder-decoder structure. In the data pre-processing stage, they view the source code as plain text and transform it into a collection of lexical vectors using common NLP techniques. The lexical vectors are sequentially fed into the model's encoder, and the final vector representation of the source code is obtained through the LSTM neural network. In the summary generation stage, the decoder of the model uses the vector representation of the source code to output each word of the corresponding summary in turn.

In contrast to traditional natural languages, programming languages contained artificially designed complex structures. To get the structural and semantic features hidden inside the code only using classical information extraction algorithms is laborious. In recent years, researchers have tried to analyze source code fragments with coding algorithms based on lexical features, syntactic structures, and semantic structures to generate relevant abstractions and annotations. Hu [18] et al. introduced DeepCom, a well-known abstract generation model based on an encoder-decoder architecture, which uses a special traversal algorithm of an abstract syntax tree to obtain a structure-rich code vector, and then decodes it into natural language using a decoder based on LSTM neural networks.

2.3 Graph Convolutional Neural Network

Convolutional neural networks [19-20] have grown rapidly and attracted a lot of attention due to their powerful modelling capabilities in the past few years. The introduction of convolutional neural networks has brought greater improvements than traditional methods in areas such as processing images and processing natural languages, such as machine translation, image recognition and speech recognition. Convolutional neural networks are good, but they are still limited to data in Euclidean domains. However, there is a lot of data in our real life that does not have a regular spatial structure, called non-euclidean data, such as recommendation systems, computational geometry, brain signals, molecular structures, etc. Graph Convolutional Network is a method that can perform deep learning on graph data, and this method has been shown to largely outperform other related methods of citation networks and knowledge graph datasets [5]. GCN is a natural generalization of convolutional neural networks in the graph domain. It can perform end-to-end learning of both node feature information and structure information and is currently one of the best choices for graph data learning tasks.

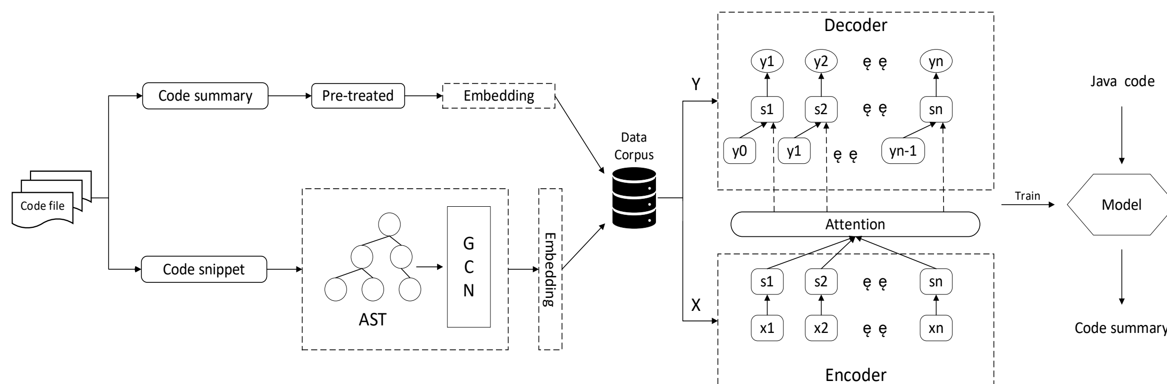

Figure 1. Overall model architecture

3 G-DCS Model

The model processing process of data is divided into three main phases: the data processing phase, the model training phase, and the model testing phase. The abstract syntax tree (AST) is used for the processing of the code in the data processing phase, and the GCN is used to process the AST and abstract the structural information in the code. The overall architecture of the model uses an end-to-end encoder-decoder architecture, in which the encoder and decoder use GRU, which simplifies the parameters of forwarding propagation compared to LSTM, and can achieve comparable results compared to LSTM and is easier to train compared to LSTM, which can largely improve the training efficiency. The general framework of the model is shown in “Figure 1”.

3.1 Code Representation

GCN:

Since the code itself is very structured and includes a large number of structural information on itself, to extract such structural information, we use a graph convolutional neural network to process it. The forward propagation of the graph neural network is as follows:

$$H(l+1) = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}). \quad \textbf{(1)}$$

Suppose a sequence of code has n tokens, the number of features of each token is d, these the nodes features form a feature matrix is $X \in R^{n\times d}$. The relationships between nodes can also create the adjacency matrix $A \in R^{n\times n}$.

$$\tilde{A} = A + I. \quad \textbf{(2)}$$

I is the unit matrix, $\tilde{D}$ is the degree matrix of $\tilde{A}$, σ is a nonlinear activation function, and W is the parameter for which the model is to be trained. For the input layer, H equals X.

AST:

In this model, different methods are used to handle the source code and abstraction in the dataset separately. Firstly, the code fragments are processed accordingly, i.e., a runnable program code is constructed with its abstract syntax tree (AST), then the source code vocabulary is constructed by traversing the syntax tree, and then the embedding (code embedding) is represented as a vector, while the edge and node information of the syntax tree is The adjacency matrix is constructed, and the sequence information and structure information of the source code fragments are combined using GCN, the process is shown in “Figure 2”.

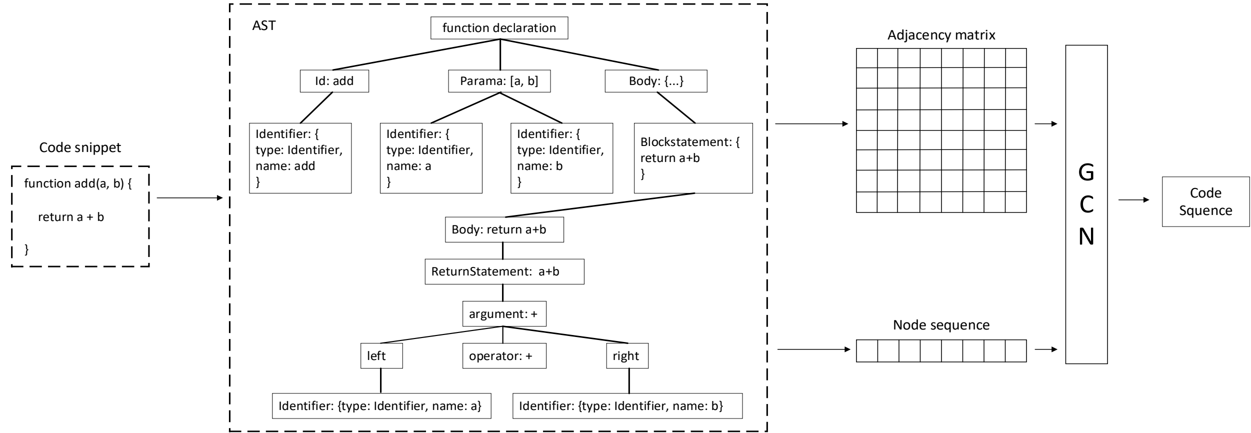

Figure 2. Code processing

Next is the processing of the summary, extracting the text summary of the source code file, and performing text pre-processing (word separation, adding <‘bos’>, <‘eos’>, <‘pad’>, <‘unk’> et. special characters) to construct the summary vocabulary, and representing the summary sequence as a summary vector through the embedding layer.

3.2 End-to-End Architecture

End-to-end models have been widely used in machine translation, text summarization, dialogue systems [21], etc. We also use end-to-end models to learn the source code to generate code summaries of this article, and the end-to-end model of G-DCS consists of three parts: an encoder, a decoder and an attention part, where both the encoder and decoder use GRU. The encoder and decoder framework in this model is shown in “Figure 3”.

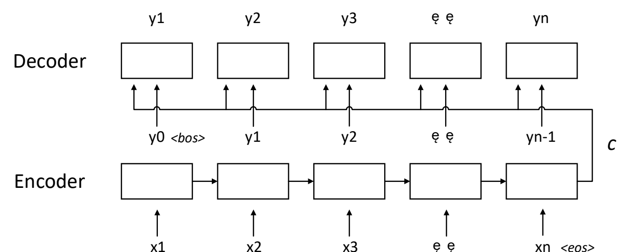

Figure 3. Encoder and decoder architecture

Encoder:

The encoder is responsible for transforming a variable-length input sequence into a constant-length background variable c, which contains the sequence information about the encoded input. The special symbol “<eos>” after each sequence indicates the termination of the sequence. In decoder models without attention-based mechanisms, the hidden state of the encoder at the final time step is generally used as the encoding information on the sentence.

Suppose the input sequence $(x_1, x_2, \ldots, x_t)$, x_i is the i-th word is in the input sequence at time step t.

$$h_t = f(x_t, h_{t-1}). \quad (3)$$

Where h_t is the hidden state of the current time step, h_{t-1} is the hidden state of the previous time step x_t is the input of the current time step. Assume that the hidden states of each time step are $(h_1, h_2, \ldots, h_T)$.

$$c = f(h_1, h_2, \ldots, h_T). \quad (4)$$

Decoder:

For the given sequence $(x_1, x_2, \ldots, x_T)$, the background variable c encodes the information about the entire sequence. In the decoder, the output of a certain time steps $y_{t'}$.

$$P(y_{t'}) = P(y_{t'} \mid y_1, \ldots, y_{t'-1}, c). \quad (5)$$

The hidden state $s_{t'}$ of the current time step of the decoder can be calculated using the input $y_{t'-1}$ of the previous time step of the decoder and the background vector c.

$$s_{t'} = g(y_{t'-1}, c, s_{t'-1}). \quad (6)$$

After the hidden state of the decoder is obtained, the softmax function can be used to calculate $P(y_{t'})$.

Attention mechanism:

The attention mechanism, in a broad sense, contains query terms and the one-to-one corresponding key terms K and value terms V, where the value terms are the set of terms to be weighted and averaged. In the weighted average, the weights of the value terms are derived from the query term Q.

$$Attention(Q, K, V) = Softmax(QK^T)V. \quad (7)$$

For encoders with attention mechanisms, the hidden state of time step t' is calculated by functioning according to the hidden state of the decoder at the previous time step $t'-1$, and the input of the Softmax operation is obtained, and the encoder hidden variable on each step is obtained by Softmax calculation, and then a weighted average is done to obtain the background vector c. As shown in “Figure 4”.

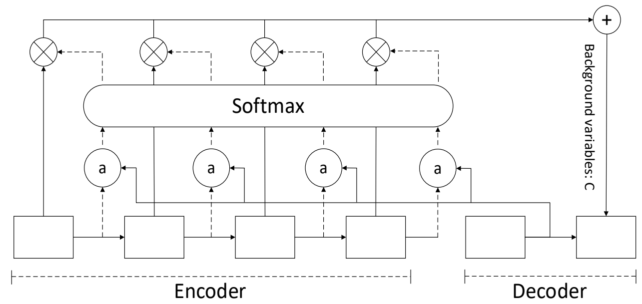

Figure 4. Attention mechanism

The background variable of the decoder at time step t' is the weighted average of all hidden states of the encoder.

$$c_{t'} = \sum_{t=1}^{T} \alpha_{t't} h_t. \quad (8)$$

t' denotes the time step of the decoder, and t denotes the time step of the encoder. At a given time step t', the probability distribution of the weights $\alpha_{t't}$ at $t = 1, \ldots, T$.

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^{T} \exp(e_{t'k})}, \ t = 1, \ldots, T. \quad (9)$$

$e_{t't}$ depends on both the decoder’s time step t' and the encoder’s time step t. We denote the hidden state of the decoder by s and the hidden state of the encoder by h.

Loss function:

In this article, the loss function of model training is defined using minimized cross-entropy.

$$H(y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j}^{L} \log(y_j^{(i)}). \quad (10)$$

The N denotes the number of training samples, L denotes the size of the target summary, and $y_j^{(i)}$ denotes the j-th word in the i-th summary generated. Optimization using the Adam algorithm.

4 Experimental Analysis

4.1 Dataset Description

Since the tag data available in the field of code summary generation is very sparse, this paper uses java function segments with summaries collected on Github and uses the first bureau of the function summary as the summary tag for the relevant code segment. The details of the dataset are shown in “Table 1” and “Table 2”.

Table 1. Statistics for code snippets

Methods	All tokens	All identifiers	Training set	Test set	Validation set
1,200	101,460	11,207	1,000	100	100

Table 2. Statistics for code lengths and comments lengths

Code lengths				
Avg	Median	<100	<150	<200
80	40	79.9%	88.5%	92.5%
Summary lengths				
Avg	Median	<20	<30	<50
17.95	12	74.2%	86.1%	94.0%

4.2 Contrast Model

To evaluate the performance of the G-DCS model, we select several representative summary generation models for comparison experiments, including the Code-NN model, and the seq2seq using the attention mechanism.

Code-NN:

We replicate the Code-NN model for the task of summary generation of java code, which uses recurrent neural networks to build an end-to-end summary generation system that generates relevant summaries based on the word vectors of the source code. Since the architecture of the Code-NN model is very similar to seq2seq, so we replicated the Code-NN model that does not incorporate the attention mechanism to verify its effectiveness of the attention mechanism.

Seq2Seq:

The model is a code summary generation model implemented based on sequence-to-sequence learning algorithms. The model's encoder and decoder are also designed using independent LSTM neural networks, this network can extract the lexical features of the source code to generate Abstracts. The model inputs focused on lexical sequences of source code functions and output English summaries associated with these functions.

4.3 Pre-processing and Parameter Setting

In the process of data pre-processing, the Javalang tool was used for AST tree extraction of java code, and the NLTK splitting tool was used for java digest processing. Considering that the length of the code segment varies greatly, we set up the size of the code token sequence to 300 and truncate the sequence for more than 300 tokens. The node relationships in the AST of the code are represented by the adjacency matrix, and the length of the edges of the adjacency matrix is also set to 300.

The G-DCS model was built using the machine learning framework Pytorch, and the word vector and GRU hidden layer states were set to 300 dimensions. The model variable was optimized into the supervised training Adam algorithm, using the Adam optimizer and setting the learning rate to 0.001. To prevent overfitting of the model parameters, the dropout value was set to 0.5

4.4 Evaluation Measure

The evaluation criteria for the experiments used the BLEU [22] metric, which is common in the field of natural language processing, to evaluate the similarity between the summaries generated by various models and the reference summaries.

The general idea of the BLEU evaluation metric is the accuracy rate. Adding the given standard translation reference, the neural network generates the sentence as a candidate. The n-gram expresses the number of consecutive words is n.

$$BLEU_n = \frac{\sum_{c\in candidates}\sum_{n-gram\in c} Count_{clip}(n-gram)}{\sum_{c'\in candidates}\sum_{n-gram'\in c'} Count(n-gram')}. \quad \textbf{(11)}$$

The main task of the BLEU programming implementation is to compare the n-tuples of candidate and reference translations and to calculate the number of matches. The number of matches is independent of the position of the words. The higher the number of matches, the better the quality of the candidate translation. We consider the code summary auto-generation task as a machine translation task, and the generated content and the reference content are also both natural language sequences, so the BLEU metric is also very suitable as an evaluation criterion for the code annotation auto-generation task.

BLEU-n judges the division of the sequence into phrases of length 1 word, length 2 is a two-word phrase, and so on, and in general, the maximum phrase length is set to 4. BLEU-1 judges the word-level accuracy, and BLEU-4 can measure the fluency of sentences. In this experiment, we use BLEU2, BLEU3, and BLEU4 as the evaluation metrics for the experiment, respectively.

In the code summary generation task, it is much easier to generate summaries for short sequences than for longer sequences. In order to make the evaluation results fairer, we add a penalty factor term to the evaluation metrics, giving a lower weight to the shorter sequences generated by the model and a higher weight to the longer sequences generated by the idea.

$$PF = \exp\left(\min\left(0, 1-\frac{len_{label}}{len_{pred}}\right)\right). \quad \textbf{(12)}$$

Where len_{label} is the length of the summary in the corpus and len_{pred} is the length of the summary formed by the model.

5 Results

In this section, we focus on evaluating the quality of different methods in generating java code summaries. The experiments have two main concerns.

RQ1: Is there any significant improvement in the quality of generated code summaries in our model versus the traditional automatic code summary generation model?

RQ2: The impact of adding structural information to the code summary generation model on the quality of code summary generation.

5.1 RQ1: G-DCS vs. Baseline

In the comparison experiments, we compared the G-DCS model with the Code-NN model and the Seq2Seq model on the BLEU-2, BLEU-3, and BLEU-4 metrics, respectively, and the obtained experimental results are shown in "Table 3".

Table 3. Evaluation results on Java methods

Approaches	BLEU-2	BLEU-3	BLEU-4
Code-NN	9.32%	3.71%	0.63%
Seq2Seq	11.48%	6.83%	3.17%
G-DCS	**17.21%**	**14.18%**	**9.38%**

We did not take the IR method as the baseline model, because in previous studies, the Code-NN method was significantly better than IR. The Code-NN model does not use a language model but generates relevant summaries by extracting lexical information directly from the source code across an end-to-end recurrent neural network system.

In the original Code-NN model, the processing language is c#. In order to make Code-NN applicable to Java code, we have made some changes to the original Code-NN to make it applicable to the Java code summary generation task. This is based on the fact that the architecture of the Code-NN model is very similar to that of the Seq2Seq model. The rewritten Code-NN model does not use an attention mechanism, which is to compare with the attention-based Seq2Seq model to verify that attention improves the task of code summary generation.

In the seq2seq model, both encoder and decoder adopt RNN series models, specifically GRU. The calculation method of the attention mechanism in seq2seq is to obtain a probability distribution, that is, the weight of attention, through dot product operation according to the hidden state of each sequence time step in the decoder, and then through softmax. Then the vector corresponding to each word in the encoder sequence is weighted and the final result is obtained.

Among all the models participating in the experiments, the CODE-NN model produced the worst quality summaries. The Seq2Seq model ignored the structural properties of the code and generated summaries of unsatisfactory quality. Unlike the Seq2Seq model, the Code-NN model directly embeds the token of the source code to generate summaries but does not learn the semantic information of the source code. The seq2seq language model built by GRU makes effective use of the semantic information in Java code, test results show that the scores on Blue2, 3 and 4 are better than the Code-NN model. The G-DCS model can not only use the semantic information of the code, but also use the code structure information in the code representation stage, and fuse the structure information through GCN. Compared with the Code-NN model, the scores of the G-DCS model in blue-2, 3 and 4 increased by 7.89%, 10.47% and 8.75% respectively. Compared with the Seq2Seq model, the G-DCS model improved the scores of blue-2, 3 and 4 by 5.73%, 7.35% and 6.21% respectively.

It can be seen from the experimental results that our model has a significant improvement compared with other baseline models. The experiment verifies that the structure information plays an important role in the quality of code digest generation. By enriching the code structure information, this method can be applied in the subsequent research, no matter what training model is used.

5.2 RQ2: The Impact of Structural Information

When we further analyze Table 3, we will find more. Comparing the experimental results of G-DCS and Seq2Seq, we can find that when the code summary model introduces structural information, the BLEU scores of all levels of the model have been improved to a certain extent. This is because the seq2seq model only uses code semantic information and lacks structural information. The G-DCS model uses GCN to encode the token and AST of the code and combines the encoding information of the two to finally generate the embedded representation of the code sequence.

Such experimental results illustrate two points of information. First, during the training of the G-DCS model, the structure information of the code is fully learned. Secondly, the use of structured information plays an important role in the code summary generation task.

Some experimental results of the code summary generated by the G-DCS model are shown in “Table 4”. Due to space limitations, the examples are limited to short methods. The AST structure is not shown in the table because the AST is much longer than the source code, similar to the conversion process from code to AST shown in “Figure 2”, one very simple code may generate a very complex AST.

5.3 Influencing Factors of Experiment

5.3.1 Decoder Based on Attention Mechanism

Compared with Code-NN, the Seq2Seq model incorporates an attention mechanism in the encoder stage, and the corresponding weights are assigned for each step of the input sequence. From the experimental results, the addition of the attention mechanism leads to a more significant improvement in the generated summary in the bleu-2, bleu-3, and bleu-4 metrics, respectively.

5.3.2 Utilization of Code Structure Information

Compared with Code-NN and Seq2Seq, the G-DCS model not only adds the attention mechanism in the decoder but also uses the code structural information at the code representation stage and mixes the structural and semantic information of the code through GCN. Experimentally, the model outperforms the model that does not utilize structural information in the BLEU-2, BLEU-3 and BLEU-4 metrics.

5.3.3 Other Factors

The lack of unified and standard datasets is a major obstacle to the rapid development of code summarization research [23]. The unification of test datasets plays a positive role in promoting neural code summarization research. Test datasets will directly affect the evaluation of summarization algorithms, and some neural code summarization systems get better evaluation results on their own selected and processed datasets, but when switching to other datasets for testing, the evaluation results will be more different. The Code-NN model was tested on the java dataset of Zhang [24] et al., and the BLEU-4 was 6.4%, while the BLEU-4 on another java dataset C2CGit, was 13.48%, with a result difference of 7.18%, which shows that the same code abstraction model was tested on different test datasets and different evaluation results were obtained, causing this phenomenon to The main reason for this phenomenon is that different studies use their own different methods to parse and process the dataset, so there is a great difference between the items, thus making the evaluation results difficult to compare, and the lack of a unified and standard test dataset makes the progress of code summary research slow.

Table 4. Examples of generated summary by G-DCS

Case ID	Java method	Summary

1	public JsonArray(){
	value = new ArrayList<JsonValue>();
	}
		Pred:
		Creates a new dataset with the given database connection.
		Value:
		Creates a new empty JsonArray.

2	StackFrame(AsmMethodSource){
	This.src=src;
	}
		Pred:
		Constructs a new word from the current server version running.
		Value:
		Constructs a new stack frame.

3	public Object clone(){
	try{
	return super.clone();
	}
	catch (CloneNotSupportedException e){
	throw new InternalError();
	}
	}
		Pred:
		Returns a copy of this deque.
		Value:
		Returns a shallow copy of this list

4	Static <T>T checkNotNull(T reference){
	if (reference == null){
	throw new NullPointerException();
	}
	return reference;
	}
		Pred:
		Ensures that an object reference passed as a parameter to the calling method is not a method.
		Value:
		Ensures that an object reference passed as a parameter to the calling method is not a null.

5	public Boolean is PrimaryKey(){
	return isPrimaryKey(false);
	}
		Pred:
		Returns true if this contour path is closed.
		Value:
		Returns true if the entity contains all of the primary key fields, but NO others.

6 Conclusion

In this article, a GCN-based deep code summary generation model G-DCS is proposed for the source code summary generation task. The model is built based on the neural machine translation (NMT) framework to transform the input source code into a code summary described in natural language. Compared with other baseline models, the structural features of the programming language are exploited, and the source code features are extracted more comprehensively using GCN to fuse code structure and semantic information.

This method enriches the information of code representation, improves the quality of code digest generation, provides a clear idea for subsequent experiments, and also provides a new method for natural language processing of token representation in other fields

The model uses "graph convolution" to represent the code, and makes full use of the structure information of the code. The existing experimental results are more in line with expectations. In future work, we will start from the following three directions.

6.1 Getting More Structural Information from AST

Our early processing of the code is to generate an AST, and then we will get structure information from the AST. AST is a tree structure, which can better represent the structural information of the code, but it can not well represent the pre and post-dependencies of variables in the code. In the next work, we start with expanding the variable dependency of AST. By analyzing the before and after dependency of variables in the code, we increase the number of edges of AST nodes and expand AST into a graph structure, to extract more structure information.

6.2 Improve the Possible Gradient Disappearance Problem in the Training Model

Our model adopts the popular end-to-end architecture, including an encoder and decoder using GRU. Because the neural network model such as RNN is still used, the model gradient may disappear when training a long sequence, resulting in the generated code summary not reaching the desired effect. In the next work, we refer to the encoder and decoder model based on the transformer. By using the multi-headed self-attention mechanism and adding the position-coding information of the sequence, we can improve the

gradient disappearance problem in the process of long sequence training.

6.3 Use the Pre-training Model to Further Improve the Effectiveness of the Model

The pre-training model has performed well in many natural language processing tasks. In previous studies, the use of a pre-training model can improve the training effect of the original model by multiple orders of magnitude. In future research, we will also use the pre-training model to enhance the learning ability of the model and further improve the effectiveness of the code summary generation model.

Acknowledgment

This work is supported by the Xinjiang Key Research and Development Program (2022B01007-1), the National Natural Science Foundation of China (62241209) and the Xinjiang Tianshan Youth Project of China (2020Q019).

References

[1] X. Xia, L.-F. Bao, D. Lo, Z.-C. Xing, A. E. Hassan, S.-P. Li, Measuring program comprehension: A large-scale field study with professionals, *IEEE Transactions on Software Engineering*, Vol. 44, No. 10, pp. 951-976, October, 2018.

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

[3] A. Hindle, E. T. Barr, M. Gabel, Z.-D. Su, P. Devanbu, On the naturalness of software, *Communications of the ACM*, Vol. 59, No. 5, pp. 122-131, May, 2016.

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

[5] T. N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, *International Conference on Learning Representations (ICLR 2017)*, Toulon, France, 2017, pp. 1-14.

[6] Z. Ghahramani M. Welling, C. Cortes, N. Lawrence, K. Q. Weinberger, *Advances in Neural Information Processing Systems 27 (NIPS 2014)*, Curran Red Hook, NY, 2015.

[7] Y.-F. Chen, Y.-K. Lin, C.-F. Huang, Using Deep Neural Networks to Evaluate the System Reliability of Manufacturing Networks, *International Journal of Performability Engineering*, Vol. 17, No. 7, pp. 600-608, July, 2021.

[8] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, Q. Wang, Neural detection of semantic code clones via tree-based convolution, *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, Montreal, QC, Canada, 2019, pp. 70-80.

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

[10] H. H. Wei, M. Li, Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code, *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, Melbourne, Australia, 2017, pp. 3034-3040.

[11] Y.-B. Qu, W. E. Wong, D.-C. Li, Empirical Research for Self-admitted Technical Debt Detection in Blockchain Software Projects, *International Journal of Performability Engineering*, Vol. 18, No. 3, pp. 149-157, March, 2022.

[12] D.-C. Li, W. E. Wong, M.-Y. Jian, Y. Geng, M. Chau, Improving Search-based Automatic Program Repair with Neural Machine Translation, *IEEE Access*, Vol. 10, pp. 51167-51175, April 2022.

[13] N. Tran, H.-H. Chen, J. Jiang, J. Bhuyan, J.-H. Ding, Effect of Class Imbalance on the Performance of Machine Learning-based Network Intrusion Detection, *International Journal of Performability Engineering*, Vol. 17, No. 9, pp. 741-755, September, 2021.

[14] S. Haiduc, J. Aponte, A. Marcus, Supporting program comprehension with source code summarization, *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010, pp. 223-226.

[15] G. Salton, A. Wong, C.-S. Yang, A vector space model for automatic indexing, *Communications of the ACM*, Vol. 18, No. 11, pp. 613-620, November, 1975.

[16] D. Blei, A. Ng, M. Jordan, Latent dirichlet allocation, *The Journal of Machine Learning Research*, Vol. 3, pp. 993-1022, January, 2003.

[17] T. K. Landauer, P. W. Foltz, D. Laham, An introduction to latent semantic analysis, *Discourse processes*, Vol. 25, No. 2-3, pp. 259-284, 1998.

[18] X. Hu, G. Li, X. Xia D. Lo, Z. Jin, Deep code comment generation, *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, Gothenburg, Sweden, 2018, pp. 200-210.

[19] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, Vol 86, No. 11, pp. 2278-2324, November, 1998.

[20] A. Ajit, K. Acharya, A. Samanta, A review of convolutional neural networks, *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, Vellore, India, 2020, pp. 1-5.

[21] W. Chan, N. Jaitly, Q. Le, O. Vinyals, Listen, attend and spell: A neural network for large vocabulary conversational speech recognition, *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Shanghai, China, 2016, pp. 4960-4964.

[22] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu:

a method for automatic evaluation of machine translation, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, Pennsylvania, United States, 2002, pp. 311-318.

[23] A. LeClair, C. McMillan, Recommendations for Datasets for Source Code Summarization, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Minneapolis, Minnesota, USA, 2019, pp. 3931-3937.

[24] J. Zhang, X. Wang, H.-Y. Zhang, H.-L. Sun, X.-D. Liu, Retrieval-based Neural Source Code Summarization, *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, Seoul, Korea, 2020, pp. 1385-1397.

Biographies

Changsheng Du received the B.S. degree in Software Engineering from Nanyang Institute of Technology. He is a graduate student at Xinjiang Normal University, China. His current research interests include Machine Learning and Software Reliability Engineering.

Yong Li received the Ph.D. degree in Computer Science from Nanjing University of Aeronautics and Astronautics in 2018. He is currently an Associate Professor of Xinjiang Normal University and a Postdoctoral Fellow of Xinjiang Electronic Research Institute. His research interests include Machine Learning and Intelligent Software Engineering.

Ming Wen graduated from Xi’an University of Technology in 1988 with a major in automatic control. Now he is the director and researcher of software development and testing center of Xinjiang Electronic Research Institute. His research interests include Software Engineering and Artificial Intelligence.