The Journal of Systems & Software 181 (2021) 111036

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

SeCNN: A semantic CNN parser for code comment generation☆

Zheng Li [a], Yonghao Wu [a], Bin Peng [a], Xiang Chen [d,e], Zeyu Sun [b,c,*], Yong Liu [a,**], Deli Yu [a]

[a] College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, China
[b] Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
[c] Software Institute, Peking University, 100871, China
[d] School of Information Science and Technology, Nantong University, Nantong, China
[e] Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, China

ARTICLE INFO

Article history:
Received 31 May 2020
Received in revised form 8 June 2021
Accepted 5 July 2021
Available online 13 July 2021

Keywords:
Program comprehension
Code comment generation
Convolutional Neural Network
Long short-term memory network

ABSTRACT

A code comment generation system can summarize the semantic information of source code and generate a natural language description, which can help developers comprehend programs and reduce time cost spent during software maintenance. Most of state-of-the-art approaches use RNN (Recurrent Neural Network)-based encoder–decoder neural networks. However, this kind of method may not generate high-quality description when summarizing the information among several code blocks that are far from each other (i.e., the long-dependency problem). In this paper, we propose a novel **Se**mantic **CNN** parser SeCNN for code comment generation. In particular, we use a CNN (Convolutional Neural Network) to alleviate the long-dependency problem and design several novel components, including source code-based CNN and AST-based CNN, to capture the semantic information of the source code. The evaluation is conducted on a widely-used large-scale dataset of 87,136 Java methods. Experimental results show that SeCNN achieves better performance (i.e., 44.69% in terms of *BLEU* and 26.88% in terms of *METEOR*) and has lower execution time cost when compared with five state-of-the-art baselines.

1. Introduction

During the process of software development and maintenance, developers spend nearly 60% time on source code comprehension (Xia et al., 2017). Code comments, which describe a piece of code by natural language, are the most intuitive and effective way for the developers to understand software code. High-quality code comments play an important role in software maintenance and reuse. Unfortunately, due to tight project schedule (Hu et al., 2018a), a large number of software projects do not provide complete comments, or some comments are outdated due to software updates. This can significantly reduce the readability and maintainability of the program. Moreover, writing code comments is also a tedious and time-consuming task in software development, and requires a lot of effort from the developers.

Automatic code comment generation techniques are effective ways to address those issues. Given a code snippet, a code comment generation system can generate the target code comment in natural language. Previous code comment generation approaches can be classified into two categories: template-based approaches and AI (Artificial Intelligence)-based approaches. The template-based approaches usually predefine a set of sentence templates and fill them by the content of the target code segment (Haiduc et al., 2010b,a; Rodeghero et al., 2015; Moreno et al., 2013). Although significant progress has been made based on keywords and sentence templates selection, these template-based approaches still have limitations (LeClair et al., 2019).

With the development of AI-based approaches, more studies tend to apply the encoder–decoder framework (Sutskever et al., 2014) to code comment generation. In this framework, the RNNs are usually served as the encoder and the decoder. When applied to code comment generation, it takes the source code as the input sequence and generates the code comment as the output sequence (Iyer et al., 2016). However, as a strict structural text, source code contains rich structural information, which is important for program modeling (Sun et al., 2019b). To address this issue, state-of-the-art approaches generate the comment based on the Abstract Syntax Tree (AST) of the code via RNNs (Hu et al., 2018a; LeClair et al., 2019).

Based on our analysis, these approaches still face two problems. The first problem is the long dependency problem. The code comment may summarize the information among several

☆ Editor: Aldeida Aleti.

* Corresponding author at: Software Institute, Peking University, 100871, China.
** Corresponding author.
E-mail addresses: szy_@pku.edu.cn (Z. Sun), lyong@mail.buct.edu.cn (Y. Liu).

https://doi.org/10.1016/j.jss.2021.111036
0164-1212/

code blocks that are far from each other. The second problem is the semantics encoding problem. Existing AST-based approaches serialize the AST into a sequence of tokens via a traversal method and extract the features by RNNs. However, RNNs are designed for encoding sequences, which still cannot capture the structural semantics well.

In this paper, we propose a novel **Se**mantic **CNN** parser SeCNN to generate code comments for Java methods, which are functional units of Java programming language, and we only choose the first sentence of Javadoc as the comment of the corresponding Java method, because it typically describes the functionalities of Java methods according to Javadoc guidance. Since CNNs are able to capture features of code effectively among different blocks by sliding windows (Sun et al., 2019b), **SeCNN** uses a CNN, which captures features effectively among different code blocks, to alleviate the long-dependency problem (Bengio et al., 1994). Furthermore, we design several novel components, including source code-based CNN and AST-based CNN, an Improved Structure-Based Traversal (ISBT) method to encode the semantics of the source code, and identifier split via camel casing[1] to alleviate the out-of-vocabulary problem. SeCNN uses two Convolutional Neural Networks (CNNs) to encode semantic information of source code. One CNN is used to extract lexical information from code tokens, and another CNN is used to extract syntactic information from ASTs. To generate the code comment, SeCNN uses Long Short-Term Memory (LSTM) with an attention mechanism as the decoder to generate code comments.

To evaluate the effectiveness of SeCNN, we conduct experiments on a large-scale dataset of 87,136 Java methods (Hu et al., 2018b). Experimental results show that SeCNN achieves better performance in terms of *BLEU* (Papineni et al., 2002) and *METEOR* (Banerjee and Lavie, 2005) metrics compared with state-of-the-art approaches.

To our best knowledge, the main contributions of this paper can be summarized as follows:

- We propose a novel method SeCNN to generate code comments. In particular, SeCNN uses two CNNs as encoder to capture semantic information of source code, and uses LSTM with an attention mechanism as decoder to generate code comments.
- We propose a novel AST traversal method, named as Improved Structure-Based Traversal (ISBT), which can better encode the structure information. Moreover, to alleviate the out-of-vocabulary problem of source code's token and AST node, we use camel casing conversion to split source code's identifiers into several words, which can effectively decrease the number unique words in both token and AST node vocabulary.
- We evaluate SeCNN on a dataset of 87,136 Java methods. The experimental results show that the SeCNN is more effective and more efficient compared with five state-of-the-art baselines.
- To facilitate the replication of our study and evaluation of future code comment generation techniques, our source code and dataset used in this paper are all available in the GitHub repository.[2]

The rest of the paper is organized as follows: Section 2 presents the background of this paper. Section 3 introduces the framework and details of SeCNN. Section 4 and Section 5 show the experiment setup and result analysis. Section 6 discusses threats to validity of this paper. Section 7 surveys the studies related to code comment generation and shows the novelty of our study. Finally, Section 8 concludes the paper and shows potential future work.

[1] For example, "currentDepth" can be splited into "current" and "depth"; "isDoubleEqual" can be splited into "is", "double", and "equal".

[2] https://github.com/pengbin2018/SeCNN.

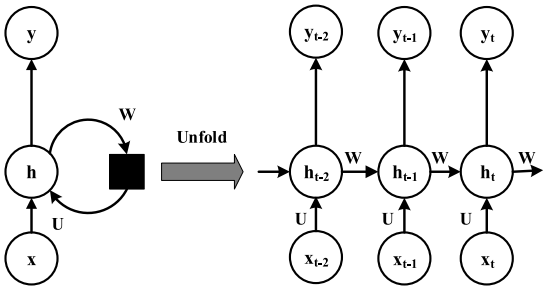

Fig. 1. Standard RNN and its unfolded.

2. Background

In this section, we first introduce the background of our study, and then show the motivation of our study.

2.1. Language model

Code comment generation techniques are inspired by the techniques used in text generation task of NLP field (Libovický and Helcl, 2018). The language model used to generate code comments is learned from code corpus. More specifically, for sequence X generated from the input sentence, where $X = (x_1, x_2, \ldots, x_n)$, the language model aims to estimate the probability of each element in the output sequence responding to sequence X.

In previous studies on code comment generation, Recurrent Neural Network (RNN) (Hu et al., 2018a) and Long Short-Term Memory Model (LSTM) (Shido et al., 2019) are two popularly used techniques, which will be introduced in the following two subsections.

2.1.1. Standard Recurrent Neural Network

Recurrent Neural Network (RNN) has been widely used in code comment generation approaches (Hu et al., 2018a; LeClair et al., 2019; Wei et al., 2019). Standard RNN takes sequence data as the input, and then it will perform recursion in the evolution direction (i.e., sequence generation direction) of the sequence. Finally, Standard RNN connects all nodes (i.e., cyclic units) in a chain. Fig. 1 shows the framework of standard RNN and its unfolded. More specifically, it reads the words in the sentence one by one, and predicts the possible subsequent words at each time step. Take the step t as an example, it calculates hidden state h_t according to the previous hidden state h_{t-1}.

Due to the ability of manipulating the sequence data, RNN can handle structural information to some extent. But it only uses the root features of structural information for supervised learning, which limits its effectiveness in code comment generation techniques.

2.1.2. Long short-term memory model

During the back-propagation process of standard RNN model, the gradient may explode or disappear, especially when there is a long dependency in the input sequence. To alleviate this problem, researchers improved RNN and proposed Long Short-Term Memory Model (LSTM) (Hochreiter and Schmidhuber, 1997; Chung et al., 2014). LSTM consists of three gates to control the state of memory cells. These three gates are called as the forget gate, the input gate, and the output gate, respectively. In particular, the forget gate can decide what information should be discarded or retained. The input gate is used to update the unit status. The output gate can determine the value of the next hidden state, which contains the relevant information previously entered. Fig. 2

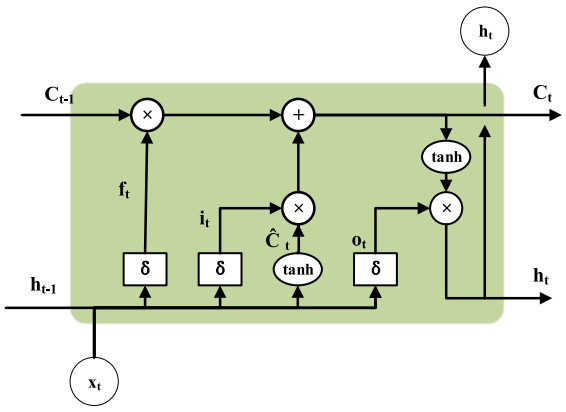

Fig. 2. The structure of an LSTM unit.

shows a typical memory cell of LSTM, where C_{t-1} is the content vector of the previous state, and h_{t-1} is the hidden state of the previous state.

In Fig. 2, σ is a sigmoid processing unit, which outputs a vector between 0 and 1 according the information provided by h_{t-1} and x_t.

f_t is generated by the forget gate, which decides what kind of information should be discarded or retained.

i_t is generated by the input gate, and it decides what kind of new information should be stored in the cell state.

$\widehat{C}$ is the new information added in the current state.

o_t is generated by the output gate, which decides what kind of value should be set to the output.

C_t is the content vector of the current state.

2.2. Convolutional Neural Network

Convolutional Neural Network (CNN) is one of the representative neural networks in the field of deep learning, and it has made many breakthroughs in the field of image analysis (Szegedy et al., 2015) and computer vision (Ren et al., 2015). Recent research shows that the CNN model is also effective in Natural Language Processing (NLP), and can perform very well in sentence classification (Kim, 2014) and web search (Shen et al., 2014) tasks. CNN-based techniques usually include a number of convolutional layers and pooling layers. In the convolutional layer, CNN will calculate the dot product between an area of the input data and the weight matrix (called a filter). The filter will slide across the entire input data and repeat the same dot product calculation operation. Average pooling and maximum pooling are two commonly-used pooling methods, of which the latter one is used the most. In CNN, the pooling layer is used to reduce the spatial dimension, however it does not reduce the depth of the network. When using the largest pooling layer, the largest feature points (i.e., the most sensitive area in the image) in the input area are used, and when using the average pooling layer, the average feature points of the input area are used.

In recent years, a number of CNN-based techniques (Kim, 2014; Szegedy et al., 2016; Kalchbrenner et al., 2014) have been proposed, and different techniques have advantage of handling different problems. Due to space limitation, we use TextCNN as an example to introduce the basic idea of CNN. TextCNN (Kim, 2014) is a type of CNN that is designed to handle text related problems. Fig. 3 shows the convolutional layer of one convolution kernel of TextCNN. The convolutional layer is used for vector feature extraction. In TextCNN, a vector of sentences with length n can be represented as $\boldsymbol{x_1}, \cdots, \boldsymbol{x_n}$, where $\boldsymbol{x_i}$ is the vector corresponding to

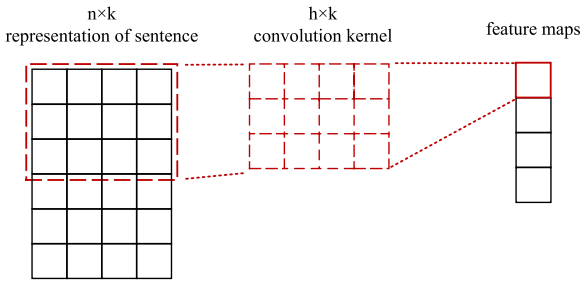

Fig. 3. Convolutional layer of TextCNN.

the ith word in the sentence. This convolution can be computed by the following formula:

$$c_i = f(W \cdot \boldsymbol{x_{i:i+h-1}} + b) \tag{1}$$

where W is the convolution kernel, which is applied to extract the features of h adjacent vectors. The size of W is $h \times k$, where k has the same dimension as the input vector, and h is the set of sliding window size. $\boldsymbol{x_{i:i+h-1}}$ represents the adjacent k vectors. b is a bias and f is a non-linear function. c_i is the feature extracted from adjacent k vectors.

2.3. Motivation

Code comment generation approaches can be used to help developers understand the purpose and content of code, but the existing RNN-based encode-decoder approaches cannot generate high-quality descriptions when summarizing information among several code blocks that are far from each other (i.e., the long-dependency problem).

Besides, there is another problem of code comments generation. Developers usually define various new identifiers, and these identifiers are composed of multiple words. Such situation would lead to the problem of the vocabulary explosion, which has a negative effect on lexical information extraction while converting it into a vector. To deal with the problem, previous studies (LeClair et al., 2019; Hu et al., 2019) split the identifier of the code into multiple words. These words usually contain lexical information, but lack of syntactic information, which could limit the effectiveness of code comment generation approaches.

To improve the quality of generated comments, in this paper, we propose a novel code comment generation approach SeCNN. Different from the previous approaches, SeCNN split the identifiers in both code and AST into multiple words, then it uses two CNNs to extract source code semantic information, and finally SeCNN will use LSTM with an attention mechanism to generate comments. The motivation of using CNN is that previous studies (Sun et al., 2019b; Allamanis et al., 2016) have proved CNN's capability of extracting lexical and syntactic information of source code. The reason of using LSTM is that previous studies (Gehring et al., 2017) in the field of natural language show that LSTM with an attention mechanism is very suitable for text generation. Moreover, different from the traditional attention mechanism (Bahdanau et al., 2014; Luong et al., 2015), our attention mechanism focuses on the features extracted by CNN, not the input sentences.

3. Approach

In this section, we first introduce the framework of SeCNN, and then show the details of this proposed approach.

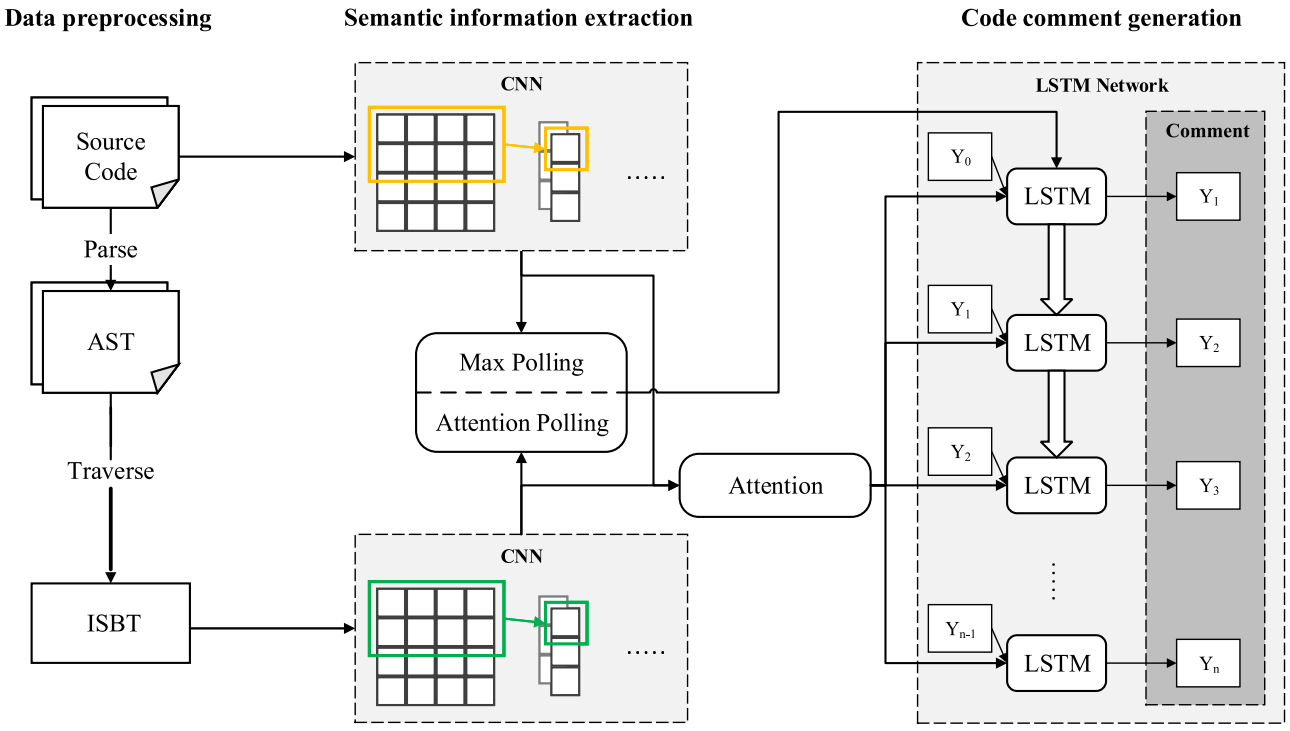

Fig. 4. The framework of our proposed approach SeCNN.

3.1. Framework of SeCNN

A significant issue with existing code comment generation techniques is that source code introduces new vocabulary faster than natural languages (Hu et al., 2018a; Karampatsis et al., 2020). SeCNN employs CNN to extract semantic information of the source code, and splits the identifier from both code and AST into multiple words to handle the vocabulary explosion challenge.

Fig. 4 shows the framework of SeCNN. In Fig. 4, it can be found that SeCNN consists of three steps: data preprocessing, semantic information extraction, and code comment generation. In the data preprocessing step, we convert the source code into the code token vector and the AST vector. To alleviate out-of-vocabulary problem, we use camel casing conversion to split the identifiers from both code tokens and AST nodes into several words, which was described in detail in Section 5.4.1. Our method's innovation is mainly reflected in the second step of Fig. 4. In this step, we use two CNNs to extract semantic information of source code. Finally, in the code comment generation step, we use LSTM with attention mechanism to decode semantic information and generation comments. The details of each step in SeCNN are introduced in the following subsections.

3.2. Data preprocessing

Since the inputs of CNN models are vectors, we need to convert all the inputs into vectors in the data preprocessing step.

3.2.1. Source code preprocessing

Source code consists of keywords, operators, identifiers and symbols, which can be used to learn lexical information by using neural network algorithms. To construct the input sequence for neural network algorithms, we employ a widely-used tool javalang[3] (Hu et al., 2019) to convert source code into tokens. Furthermore, to address the vocabulary explosion problem, we split each identifier in code according to the camel casing conversion, and all code tokens are converted to lowercase. Then, for a sequence data of code tokens, let $X = \langle x_1^{(code)}, \ldots, x_n^{(code)} \rangle$, and we use word embedding to convert it into the vectors $\langle \boldsymbol{x_1}^{(code)}, \ldots, \boldsymbol{x_n}^{(code)} \rangle$, where $\boldsymbol{x_i}^{(code)}$ is a k-dimensional vector indicating the ith code token x_i.

3.2.2. Abstract syntax tree with improved structure-based traversal

Code token can be used to learn lexical information, but it does not contain syntactical information. Abstract Syntax Tree (AST) is an abstract representation of the syntax structure of source code, and AST can be used to learn syntactical information of source code. In this paper, we also use javalang tool to parse source code to generate its AST. In particular, our approach is based on the Structure-Based Traversal (SBT) method proposed by Hu et al. (2018a), which aims to preserve the structural information of source code to the greatest extent during syntax information extraction.

Although SBT method is promising in the code comment generation task, the original sequence generated by SBT contains too much duplicate content. Moreover, SBT uses a pair of brackets to represent the tree structure, this is not conducive to coding structural information. SBT sequence also contains the identifier of source code, so it also contains the vocabulary explosion problem.

To solve these problems, we propose a novel AST traversal method, named as Improved Structure-Based Traversal (ISBT) method. ISBT method can better encode structure information of the code. Based on ISBT method, we propose a new component to encode our ISBT sequence. Fig. 5 uses a simple example to illustrate how ISBT method traverses a tree. First, we use SBT to traverse the AST to generate the SBT sequence. Then, after traversing the AST with SBT, we use the serial number of the AST via pre-order traversal to replace the brackets in the SBT sequence, and split the SBT sequence into two parts: serial numbers and AST nodes, as shown in Fig. 5. Each AST node has two attributes (i.e., type and value). Finally, to address the vocabulary explosion challenge, we use camel casing conversion to split the

[3] https://pypi.org/project/javalang/.

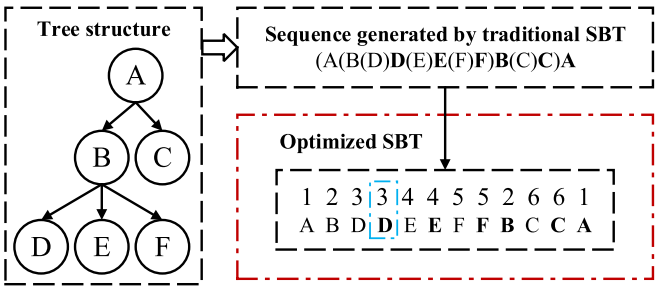

Fig. 5. An example of sequencing an AST to a sequence by ISBT (the main idea is to replace the brackets in SBT with the first order traversal number).

value of the node of type SimpleName. This type of node corresponds to the identifier of the source code. The serial number of the original node is copied to each split word. For example, a node is "isLayered" and its serial number is "3". The node "isLayered" is split to "is Layered", and the serial number "3" is copied to "3 3". Based on the above analysis, it can be seen that we can restore a SBT sequence unambiguously from a generated sequence by using ISBT. Therefore, our improvement on original SBT can still keep all information.

To construct the inputs for CNN models, we propose a ISBT-Based CNN to encode the ISBT sequence. More specifically, for a serial number sequence $X = \langle x_1^{(\text{serial})}, \ldots, x_m^{(\text{serial})} \rangle$, and a node sequence $X = \langle x_1^{(\text{node})}, \ldots, x_m^{(\text{node})} \rangle$, we use word embedding to convert them into vectors $\langle \boldsymbol{x}_1^{(\text{serial})}, \ldots, \boldsymbol{x}_m^{(\text{serial})} \rangle$ and $\langle \boldsymbol{x}_1^{(\text{node})}, \ldots, \boldsymbol{x}_m^{(\text{serial})} \rangle$, where $\boldsymbol{x}_i^{(\text{serial})}$ and $\boldsymbol{x}_i^{(\text{node})}$ are both k-dimensional vectors. To extract their features, we use ISBT-Based CNN to integrate two sequences (i.e., number and AST node) into one sequence. This convolution can be computed as follows:

$$\boldsymbol{x}_i^{(\text{isbt})} = \text{ReLU}(W^{(\text{isbt})}[\boldsymbol{x}_i^{(\text{node})}; \boldsymbol{x}_i^{(\text{serial})}]) \tag{2}$$

where $W^{(\text{isbt})}$ is the weight of the ISBT-Based CNN kernel, and $\boldsymbol{x}_i^{(\text{isbt})}$ is the vector extracted by $\boldsymbol{x}_i^{(\text{node})}$ and $\boldsymbol{x}_i^{(\text{serial})}$ through convolution. Besides, ReLU (Rectified Linear Unit) is a commonly used non-linear activation function in neural network, which is defined as follows:

$$f(x) = x^+ = max(0, x) \tag{3}$$

This activation function was first introduced to a dynamical network by Hahnloser et al. in 2000 (Hahnloser et al., 2000), and this has been widely used in subsequent neural network research (Hansel and Van Vreeswijk, 2002; Kadmon and Sompolinsky, 2015; Sun et al., 2019a; Engelken et al., 2020).

3.3. Semantic information extraction

SeCNN uses CNN model to extract semantic information, and we show the details of this step in the following subsections.

3.3.1. Convolutional neural network model

In our proposed approach, we use two CNN-based models to capture the semantic information of source code. In particular, one CNN is used to extract lexical information from code tokens, and another CNN is used to extract syntactic information from ASTs.

The vectors of all code tokens can be denoted as $\langle \boldsymbol{x}_1^{(\text{code})}, \ldots, \boldsymbol{x}_n^{(\text{code})} \rangle$, where $\boldsymbol{x}_i^{(\text{code})}$ is the k-dimensional input vector, and n means there are n vectors. Then, we apply a series of convolutional layers to extract their features $\langle \boldsymbol{y}_1^{(\text{code})}, \ldots, \boldsymbol{y}_n^{(\text{code})} \rangle$. Such convolution can be calculated as follows:

$$\boldsymbol{y}_i^{\text{code}} = f(W^{\text{code}} \cdot \boldsymbol{x}_{i:i+h-1}^{\text{code}}) \tag{4}$$

where W^{code} is the convolution kernel, which is the trainable parameter and is updated while training. The size of W^{code} is $h \times k$, where k has the same dimension as the input vector, and h is the sliding window size. $\boldsymbol{x}_{i:i+h-1}^{\text{code}}$ represents the adjacent k vectors. f is a non-linear function (e.g., ReLu function). $\boldsymbol{y}_i^{\text{code}}$ is the feature vector extracted from the $\boldsymbol{x}_{i:i+h-1}^{\text{code}}$. Here $\cdot$ represents convolution operator represents convolution, which means multiplying the two matrices' corresponding positions and then adding all products' values.

Notice that our convolution slightly differs from TextCNN (Kim, 2014). To maintain the same feature vector dimension before and after convolution, we use k convolution kernels for each convolution layer. To keep the same number of feature vectors before and after convolution, we apply zero to pad the input vector. Therefore, the feature vector extracted by each convolutional layer has the same dimension as the original input vector. Therefore, shortcut connections are feasible in SeCNN. To solve the problem of network degradation, that is, as the depth of the neural network layer increases, the accuracy rate first rises and then reaches saturation, and then continue to increase the depth will cause the accuracy rate to decline, we use a shortcut to connect each layer in parallel before activating the function.

Similar as code tokens, for the vectors $\langle \boldsymbol{x}_1^{(\text{isbt})}, \ldots, \boldsymbol{x}_m^{(\text{isbt})} \rangle$ of ISBT information, we use the same convolution operation to extract feature vectors $\langle \boldsymbol{y}_1^{(\text{isbt})}, \ldots, \boldsymbol{y}_m^{(\text{isbt})} \rangle$.

Finally, to obtain the semantic vectors, we use the concat function provided by Tensorflow to connect $\langle \boldsymbol{y}_1^{(\text{code})}, \ldots, \boldsymbol{y}_n^{(\text{code})} \rangle$ and $\langle \boldsymbol{y}_1^{(\text{isbt})}, \ldots, \boldsymbol{y}_m^{(\text{isbt})} \rangle$. The semantic vectors are represented as $\langle \boldsymbol{y}_1^{(\text{sem})}, \ldots, \boldsymbol{y}_{n+m}^{(\text{sem})} \rangle$.

3.3.2. Pooling

SeCNN uses pooling to construct the LSTM initial state, which is the encoding of the input and guides the decoder process. LSTM initial state includes context vector (c_state) and hidden state (h_state). We use two pooling algorithms, MAX pooling, and attention pooling. MAX pooling can reduce the feature dimension of the input data. Attention pooling can combine lexical and grammatical information. To construct the content vector with semantic information, SeCNN will firstly apply MAX pooling (Krizhevsky et al., 2017) on the features $\langle \boldsymbol{y}_1^{(\text{code})}, \ldots, \boldsymbol{y}_n^{(\text{code})} \rangle$ extracted from the code. Then, SeCNN uses a fixed-size controlling vector code pooling to compute the attention weights for ISBT Encoder. After that, SeCNN uses features $\langle \boldsymbol{y}_1^{(\text{isbt})}, \ldots, \boldsymbol{y}_m^{(\text{isbt})} \rangle$ extracted from ISBT to do attention pooling, which aims to make c_state contain lexical and grammatical information. Similarly, the features $\langle \boldsymbol{y}_1^{(\text{isbt})}, \ldots, \boldsymbol{y}_m^{(\text{isbt})} \rangle$ are used to do MAX pooling to get a controlling vector isbt_pooling. Then we apply attention pooling on the isbt_pooling and features $\langle \boldsymbol{y}_1^{(\text{code})}, \ldots, \boldsymbol{y}_n^{(\text{code})} \rangle$ to get the vector c_state. Finally, we use c_state and h_state as the initial state of LSTM in Decoder.

3.4. Code comment generation

In the third step of SeCNN, we use LSTM with an attention mechanism to decode the semantic information of source code and generate code comments.

3.4.1. Attention mechanism

SeCNN uses attention mechanism to assign a weight to each semantic vector extracted by CNN models. The weight value determines how important this vector is to the output target words. Specifically, in our proposed approach, we use the classical attention method proposed by Bahdanau et al. (2014) as the attention mechanism in SeCNN.

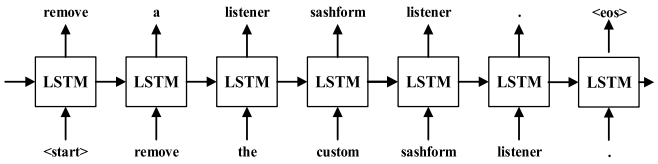

Fig. 6. Decoder in the training mode.

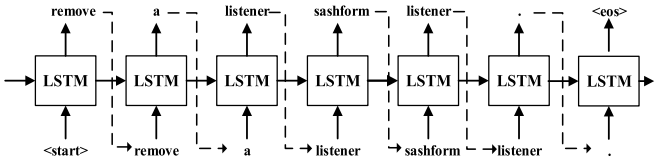

Fig. 7. Decoder in the testing mode.

The attention mechanism needs to calculate a context vector $\boldsymbol{c_i}$ for predicting each target word $\boldsymbol{y_i}$ by the following formula:

$$\boldsymbol{c_i} = \sum_{j=1}^{m+n} a_{ij} \boldsymbol{y_j^{(sem)}} \tag{5}$$

where $\boldsymbol{y_j^{(sem)}}$ is the semantic vector extracted by CNN models, and a_{ij} indicates the corresponding attention weight of $\boldsymbol{y_j^{(sem)}}$.

To calculate a_{ij}, SeCNN firstly calculates an alignment model score e_{ij} to measure how well of each input matches the current output of the decoder, and the calculation formula is defined as follows:

$$e_{ij} = \boldsymbol{a}(\boldsymbol{h_{i-1}}, \boldsymbol{y_j^{(sem)}}) \tag{6}$$

where $\boldsymbol{a}$ is the alignment model, which is a feed-forward neural network.

Finally, we use the softmax function to normalize the score to obtain the attention weight, and the formula is defined as follows:

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{m} exp(e_{ik})} \tag{7}$$

3.4.2. Sequence output

The purpose of decoder is to decode the semantic vectors generated by CNN models and generate code comments $y_1, \ldots, y_n$. Here, y_i is predicted by the following formula:

$$y_i = P(y_i|y_1, \ldots, y_{i-1}, x) = g(y_{i-1}, h_i, c_i) \tag{8}$$

where g is a stochastic output layer and is used to estimate the probability of word y_i. c_i is the context vector, and h_i is the current hidden state.

In the decoder process shown in Fig. 4, we use ⟨start⟩ as the first input-word(Y_0) to the decoder. Then, we use ⟨eos⟩ as the last word(Y_n). Taking Case 3 in Table 10 as an example, LSTM receives a series of prior words as the input (Williams and Zipser, 1989) during the model training process. Fig. 6 shows the predicted result word by word from the decoder component in the training mode, and Fig. 7 shows decoder in the testing mode, which takes the previous predicted words as a part of input rather than labeled text.

4. Experiment setup

4.1. Research questions

In this section, we evaluate the effectiveness of our proposed approach by comparing it with state-of-the-art baselines when considering the accuracy and efficiency of generating Java method comments. Specifically, We focus on the following three research questions:

RQ1: Can SeCNN outperform state-of-the-art code comment generation baselines?

This RQ is designed to verify the code comment generation effectiveness of SeCNN. To answer this RQ, we conduct a series of empirical studies and compare SeCNN with other state-of-the-art code comment generation baselines, including DeepCom (Hu et al., 2018a), TL-CodeSum (Hu et al., 2018b), Hybrid-DeepCom (Hu et al., 2019), Dual Model (Wei et al., 2019) and AST-attendgru (LeClair et al., 2019).

RQ2: How does code and comment length affect the performance of our proposed approach SeCNN?

This RQ is designed to investigate the impact of source code and comment with different length on the code comment generation effectiveness of our proposed SeCNN. To answer this RQ, we collect and analyze the experimental results of using SeCNN in generating comments for source code with different lengths.

RQ3: What is the difference between comments generated by SeCNN and human-written comments?

This RQ is designed to investigate the quality of code comments generated by SeCNN in the manual manner. To answer this RQ, we compare the difference of code comments generated by SeCNN and written by human.

Table 1
Statistics for code snippets in the dataset.

# Projects	# Files	# Lines	# Items
9732	1,051,647	158,571,730	87,136

Table 2
Statistics for code and comments length.

Statistics for comment length					
Avg	Mode	Median	<20	<30	<50
8.86	8	13	75.50%	86.79%	95.45%
Statistics for code length					
Avg	Mode	Median	<100	<150	<200
99.94	16	65	68.63%	82.06%	89.00%

4.2. Dataset

In the Previous study, Hu et al. (2018b) collected a large corpus by matching Java methods and comments in 9732 projects from GitHub.[4] In our study, we also use the same dataset and use the same way to split this dataset into training set, validation set, and test set. Tables 1 and 2 show the statistics information of the dataset used in our study.

In Table 1, the column # Items indicates that this dataset has 87,136 pairs of ⟨code, comment⟩, where the average length of comments tokens is 8.86 and the average length of code tokens is 99.94. Moreover, 95% of the code comments are less than 50 words, and about 90% of the Java methods are less than 200 tokens. Note that the comment length is the number of words and punctuation in the comment, and the code length is the number of all words and symbols before the code segmentation. In the previous study, the data was divided into the training set, the test set, and the validation set according to the ratio of 8:1:1 (Hu et al., 2018b), so we adopted the same dataset split method, and the details can be found in Table 3.

[4] https://github.com/.

Table 3
Splits of dataset.

Dataset	# Items
Training set	69,708
Validation set	8,714
Test set	8,714

4.3. Performance metrics

To evaluate the effectiveness of SeCNN, we choose two MT (Machine Translation)-based metrics (i.e., *BLEU* and *METEOR*). These chosen evaluation metrics have been widely used in previous code comment generation studies (Hu et al., 2018b, 2019; Wei et al., 2019).

BLEU (Papineni et al., 2002) score is a widely used performance metric for text generation tasks (Klein et al., 2017) in Natural Language Processing (NLP) and has been used in the evaluation the quality of generated code comments (Hu et al., 2018a,b). It calculates the similarity between the generated sequence and the reference sequence. The value of *BLEU* scores ranges from 0 to 100%. The higher the *BLEU*, the closer the candidate is to the reference.

BLEU (Papineni et al., 2002) is defined as the geometric mean of n-gram matching accuracy scores multiplied by the simplicity penalty to prevent generating very short sentences. The value is calculated by the following formula:

$$BLEU = \mathrm{BP} \cdot (\sum_{i=0}^{n} w_n log p_n) \tag{9}$$

where p_n is the geometric average of the modified n-gram *Precision*, and w_n is the positive weights. In the above formula, BP is the brevity penalty, which can be computed as follows:

$$\mathrm{BP} = \begin{cases} 1 & c > r \\ e^{1-r/c} & c \leq r \end{cases} \tag{10}$$

where c represents the length of the candidate (generated comment) and r indicates the length of the reference (extracted from Javadoc).

More specifically, in our study, we use the sentence-level *BLEU* as our performance metric, which is a widely adopted choice in previous code comment generation studies (Hu et al., 2018a, 2019; Wei et al., 2019).

METEOR (Banerjee and Lavie, 2005) is a widely used machine translation-based metric. It evaluates translation hypotheses by aligning them to reference translations and calculating sentence-level similarity scores. The feature of *METEOR* is the introduction of synonym matching.

METEOR (Banerjee and Lavie, 2005) is explicitly designed to improve correlation with human judgments of machine translation quality at the segment level. The value can be computed as follows:

$$METEOR = (1 - P_{en}) \cdot F_{mean} \tag{11}$$

where P_{en} is a penalty coefficient, and F_{mean} is a parameterized harmonic mean. P_{en} can be computed as follows:

$$P_{en} = \gamma(\frac{ch}{m})^{\beta} \tag{12}$$

where ch is the number of chunks (In this study, chunks represent several adjacent words in the reference sentence). m is the number of matches. γ determines the maximum penalty ($0 \leq \gamma \leq 1$). β determines the functional relation between the fragmentation and the penalty. γ and β are set to 0.20 and 0.60 respectively (Denkowski and Lavie, 2014).

F_{mean} can be computed as:

$$F_{mean} = \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R} \tag{13}$$

where P is the unigram *Precision* and R is the unigram *Recall*. α is set to 0.85 (Denkowski and Lavie, 2014).

The correlation of two MT-based metrics and the performance of code comment generation techniques is also positive. A higher *BLEU* or *METEOR* value indicates a better code comment generation technique.

4.4. Statistical analysis

4.4.1. Hypothesis testing method

Since experiments between our proposed approach and existing baselines are conducted on the data with the same code and comments, we adopt the Wilcoxon signed-rank test to further analyze the experimental results.

Wilcoxon signed-rank test is an alternative hypothesis test approach when the test data cannot be assumed to be normally distributed (Ott and Longnecker, 2015). Therefore, it can provide a reliable statistical basis for comparing the effectiveness of different approaches.

4.4.2. Effect size

In statistical analysis, an effect size is a number measuring the strength of the relationship between two variables in a population or a sample-based estimation of that quantity. Specifically, we calculate the Cliff's Delta (Cliff, 1993), which is a non-parametric effect size measure, to quantify the amount of difference between the two groups. We leverage the Cliff's Delta to measure the difference between SeCNN and other baselines in terms of *BLEU* or *METEOR* metrics.

Besides, the effect size defines the Cliff's Delta value of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large respectively.

4.5. Baselines

In our study, we employ five state-of-the-art code comment generation approaches as the baselines. The detailed description of these baselines is introduced as follows:

Baseline 1: DeepCom (Hu et al., 2018a) is an attention-based Seq2Seq model for generating comments of Java methods. DeepCom takes ASTs as the input, and uses SBT method to convert these ASTs into sequences in a special format. To compare the performance of SeCNN and DeepCom, in our study, we re-implement the DeepCom model according to the same parameter settings used in the corresponding study (Hu et al., 2018a).

Baseline 2: TL-CodeSum (Hu et al., 2018b) is a deep model that generates code comments by capturing semantics information from source code with the help of API knowledge. In our study, we directly use the experimental results of the corresponding study (Hu et al., 2018b) to perform comparison.

Baseline 3: Hybrid-DeepCom (Hu et al., 2019) is a variant of attention-based Seq2Seq model for generating comments for Java methods. Hybrid-DeepCom uses both the source code tokens and its AST structure to generate code comments. In our study, we re-implement Hybrid-DeepCom model according to the same parameter settings used in the corresponding study (Hu et al., 2019) to perform comparison.

Baseline 4: AST-attendgru (LeClair et al., 2019) is a neural model that combines words in source code with code structure. AST-attendgru involves two unidirectional Gated Recurrent Unit (GRU)

layers: one is used to process the words from source code, and the other is designed to process the AST. AST-attendgru modifies the AST flattening procedure proposed by Hu et al. (2018a). In our study, we re-implement AST-attendgru model according to the same parameter settings used in the corresponding study (LeClair et al., 2019) to perform comparison.

Baseline 5: Dual Model (Wei et al., 2019) is a dual learning framework to jointly train Code Generation (CG) and Code Summarization (CS) models. To enhance the relationship between the two tasks in the joint training process, in addition to imposing constraints on the probability, the Dual Model creatively proposes a constraint that uses the nature of the attention mechanism. In our study, we directly use the experimental results of the corresponding study (Wei et al., 2019) to perform comparison.

4.6. Experimental settings

For our proposed approach, the value of some parameters are set according to the previous related studies (Hu et al., 2018a,b, 2019). The value of the remaining parameters are optimized via the validation set (e.g., ISBT length, batch size). Notice the value of these parameters are not optimized on the test set.

We replace the constant numbers and strings in the source code with special marks ⟨unm⟩ and ⟨str⟩ respectively according to the setting of Hu et al. (2019). Table 2 shows 89.00% of Java methods are less than 200 tokens and we find that 90.17% of ISBT sequences do not exceed 300 tokens. Therefore, the maximum length of code sequence and ISBT sequence are set to 200 and 300. We use the special symbol ⟨pad⟩ to fill short sequences, and longer sequences will be clipped. Table 2 shows 86.79% of Java methods are less than 30 tokens. Therefore, the maximum comment length is set to 30. We add special tokens ⟨start⟩ and ⟨eos⟩ to the comments, where ⟨start⟩ is the beginning of the decoded sequence, and ⟨eos⟩ indicates the end of the decoded sequence. The vocabulary sizes of the code, ISBT, and comment are set to 30,000, 30,000, and 23,428 respectively according to the setting of Wei et al. (2019). Out-of-vocabulary tags will be replaced by ⟨unk⟩.

The model parameters and their settings are described as follows:

- We use the SGD algorithm to train the parameters, and the minimum batch size, which is used to randomly select the given number of samples from the training examples, is set to 128 (Hu et al., 2018a).
- The encoder model of SeCNN uses a nine-layer CNN. The size of all the convolution kernels is 4×500, and the convolution step is set to 4. These parameters are optimized by the validation set.
- The decoder model of SeCNN uses one-layer LSTM, where the hidden state is 500 dimension, and the embedded word is 500 dimension. The value of this parameter is optimized by the validation set, and more details can be found in Section 5.4.3.
- The initial learning rate is set to 1.0. The learning rate is decayed using the rate of 0.99. We use the exponential decay method to reduce the learning rate and set the learning rate decay coefficient to 0.99. The value of these parameters are optimized by the validation set.
- SeCNN clips the gradients norm by 5. We use dropout strategy during the training process and set dropout to 0.8 (Hu et al., 2018a).
- SeCNN uses cross-entropy minimization as the cost function (Hu et al., 2018a).

We use python 3.6 and Tensorflow[5] framework to implement SeCNN. All experiments conducted in our study run on a Linux Server (Intel(R) Xeon(R) Gold 6240 CPU @ 2.60 GHz CPU and a Tesla V100 GPU with 32 GB memory).

[5] https://www.tensorflow.org/.

Table 4
The average performance of different approaches.

Approach	*BLEU* (%)	*METEOR* (%)
DeepCom	40.51	22.24
TL-CodeSum	41.98	18.81
Hybrid-DeepCom	42.26	24.86
AST-attendgru	40.82	24.54
Dual Model	42.39	25.77
SeCNN	**44.69**	**26.88**

Table 5
P-value of Hypothesis in Section 5.1.

Approach	*BLEU*	*METEOR*
DeepCom	4.43e−88	3.29e−266
Hybrid-DeepCom	1.08e−33	4.98e−9
AST-attendgru	5.92e−121	4.35e−9

5. Result analysis

5.1. RQ 1: Can SeCNN outperform state-of-the-art code comment generation baselines?

As introduced in Section 4.5, we use five state-of-the-art and widely compared code comment generation techniques as baselines. In these baselines, we re-use the experiment results of two techniques (i.e., TL-CodeSum and Dual Model) and re-implement other three techniques (i.e., DeepCom, Hybrid-DeepCom and AST-attendgru).

To evaluate the performance of SeCNN and other baselines, we use two MT-based metrics, *BLEU* and *METEOR*, to measure the gap between automatically generated comments and manually written comments, and Table 4 shows the corresponding experimental results. From this table, we can find that SeCNN outperforms all the other five baselines in terms of both two metrics. More specifically, SeCNN achieves 2.30% to 4.18% and 1.11% to 4.64% improvements over other baselines in terms of *BLEU* and *METEOR* respectively. Such results indicate that SeCNN can learn the semantic information of the source code, and generate higher-quality code comments than other baselines.

In the field of automatic code comment generation, an improvement of up to 4.64% is an acceptable level, although the value is not impressive. For example, the method proposed by Devlin et al. (2018) obtains a 4.7% absolute accuracy improvement over the state-of-the-art; the Hybrid-DeepCom proposed by Hu et al. (2019) compared to the AST-based model DeepCom increases by 1.3%, and Zhou et al. (2019) proposed approach improves the results of *BLEU* from 38.08% to 40.52% and that of *METEOR* scores from 26.83% to 28.51%.

Furthermore, we conduct Wilcoxon signed-rank test to verify the competitiveness of our proposed approach SeCNN. Table 5 shows the results of the above hypothesis testing. Note that we only implemented three baseline approaches, so we can only perform hypothesis testing on these three approaches. The used hypothesis in our study is set as follows, H_0: There is no significant difference between SeCNN and each other approaches in terms of *BLEU* and *METEOR*. The significance level of this test is set as 0.05. It is shown in Table 5 that all the p-values are lower than 0.05, then the statistical results led to the rejection of the null hypothesis. These results imply that there is a significant difference between our proposed method and other approaches in terms of *BLEU* and *METEOR* metrics. Note that the results listed in Table 4 indicate that our method performs better than other baselines, so it is safe to conclude that SeCNN can significantly achieve better performance than other baseline approaches.

Moreover, we also use Cliff's Delta to evaluate the difference between SeCNN and other baselines in terms of *BLEU* and *METEOR*

Table 6
The performance of different methods on three independent repeated experiments.

Approach	The first experiment		The second experiment		The third experiment	
	BLEU (%)	*METEOR* (%)	*BLEU* (%)	*METEOR* (%)	*BLEU* (%)	*METEOR* (%)
DeepCom	40.51	22.24	39.67	21.59	40.40	22.16
Hybrid-DeepCom	42.26	24.68	41.27	24.66	42.24	25.25
AST-attendgru	40.82	24.54	39.42	23.79	39.95	24.28
SeCNN	**44.69**	**26.68**	**44.33**	**26.71**	**44.60**	**26.96**

Table 7
Cliff's delta between SeCNN and baselines.

Approach	*BLEU*	*METEOR*
DeepCom	0.08	0.17
Hybrid-DeepCom	0.02	0.03
AST-attendgru	0.01	0.03

metrics. As shown in Table 7, the Cliff's Delta values are less than or equal to 0.17, which corresponds to negligible or small effect size. The results show that our method can outperform existing baselines, but to a lesser extent.

There are more findings when analyzing the performance comparison among the five baselines from Table 4. For example, it can be seen that Hybrid-DeepCom performs better than DeepCom, and the reason is that DeepCom only learns syntactic information but lacks of lexical information. Hybrid-DeepCom can slightly outperform TL-CodeSum, which shows that syntactic information is better than API information in generating code comments. Hybrid-DeepCom outperforms AST-attendgru, and this proves that RNN is suitable for constructing decoder model to generate comments. Moreover, the Dual Model are better than other four baselines, which shows that the code generation and comment generation tasks are related. In future research, we want to further improve the performance of our proposed approach by considering this relationship.

To alleviate possible sampling bias in random shuffle and sampling on the dataset, we repeated the experiment three times independently to evaluate the performance of our proposed approach. In our study, we repeated the whole process three times independently. For each experiment, we divided the dataset into the training set, the test set, and the validation set randomly at 8:1:1, and then re-trained each model. The results of each experiment are shown in Table 6. Note that we have re-implemented methods DeepCom, Hybrid-DeepCom, and AST-attendgru, but we cannot re-implement TL-CodeSum and Dual model, so we only list the experimental results of the above three methods in Table 6. Compared with DeepCom, Hybrid-DeepCom, and AST-attendgru, SeCNN still has the best performance in the three experiments. Overall, based on this experimental setting, SeCNN achieves 2.62% to 4.55% improvements in terms of *BLEU* and achieves 1.92% to 4.79% improvements in terms of *METEOR* over the three baselines.

> **Summary for RQ1:** The *BLEU* score of SeCNN can reach 44.69%, and the *METEOR* score can get 26.88%. Compared to the five state-of-the-art baselines, SeCNN can achieve the best performance in terms of *BLEU* and *METEOR* metrics.

5.2. RQ 2: How does code and comment length affect the performance of our proposed approach SeCNN?

In this RQ, we want to analyze the prediction accuracy of SeCNN with different lengths of source code and comments. As introduced in RQ1, in our study, we re-implement three baselines, which are DeepCom, Hybrid-DeepCom and AST-attendgru. Hybrid-DeepCom is extended from DeepCom, and shows better performance than DeepCom. Moreover, the experimental results in RQ1 show that AST-attendgru can achieve better performance than DeepCom. Therefore, we use two baselines, which are Hybrid-DeepCom and AST-attendgru, to evaluate the performance of SeCNN. The average results of SeCNN and other two baselines can be seen in Fig. 8.

Table 8
P-value of Hypothesis in Section 5.2.

	Approach	*BLEU*	*METEOR*
Code length	Hybrid-DeepCom	4.50e−6	0.004
	AST-attendgru	6.04e−6	0.018
Comment length	Hybrid-DeepCom	4.11e−6	3.30e−5
	AST-attendgru	1.86e−5	9.09e−5

Fig. 8(a) and (c) demonstrate the performance changes in terms of *BLEU* and *METEOR* metrics of these three techniques when considering different code lengths. To make it more clear, we use approximate code length to draw this Figure, and the approximate function is defined as follows:

$$F(x) = \begin{cases} 300 & x \geq 300 \\ (\lfloor x/10 \rfloor + 1) * 10 & x < 300 \end{cases} \quad (14)$$

where x is the actual code length, and $F(x)$ returns the corresponding approximate length of x.

From Fig. 8(a) and (c), it can be seen that SeCNN performs better than other two baselines in most cases. As the code length increases, both the *BLEU* and *METEOR* of SeCNN will increase first and then maintain the accuracy at similar level. Among the different code lengths shown in Fig. 8(a) and (c), SeCNN reaches the highest in the 25 and 24 node positions of these two line charts, which means that SeCNN can achieve the best performance in most cases, and their proportions are 83.33% and 80% respectively. Since short code is often incomplete, so such performance is reasonable.

Fig. 8(b) and (d) demonstrate the changes in terms of *BLEU* and *METEOR* metrics of these three techniques when considering different comment lengths. In most cases, SeCNN has the highest *BLEU* and *METEOR* scores. Based on the impact of the different comment lengths shown in Fig. 8(b) and (d), SeCNN can achieve the best performance at the most cases in these two subfigures, and their proportions are 89.66% and 79.31% respectively. More specifically, as the comment length increases, the *METEOR* score of SeCNN increases first and then maintains similar accuracy. With the comment length increasing, the *BLEU* score of SeCNN increases first and then decreases. When the comment only contains 5 to 10 words, SeCNN will achieve the highest *BLEU* score.

Besides, we utilize the Wilcoxon signed-rank test to verify the competitiveness of our proposed method. In other words, we use statistical analysis to verify whether the polyline of our method in Fig. 8 is significantly higher than other methods. Specifically, the hypothesis used in this section is defined as follows, H_0: There is no significant difference between SeCNN and each other approach in different code lengths or comment lengths in terms of *BLEU* and *METEOR*. Table 8 reports the p-value of H_0 on two approaches using Wilcoxon signed-rank test. As Table 8 shows, the p-value in all situations is less than 0.05, so it is safe to accept the above conclusions that SeCNN can achieve significantly better performance than Hybrid-DeepCom and AST-attendgru.

Furthermore, Table 9 shows the Cliff's Delta between SeCNN and other approaches in different code lengths or comment lengths in terms of *BLEU* and *METEOR* metrics. The results listed

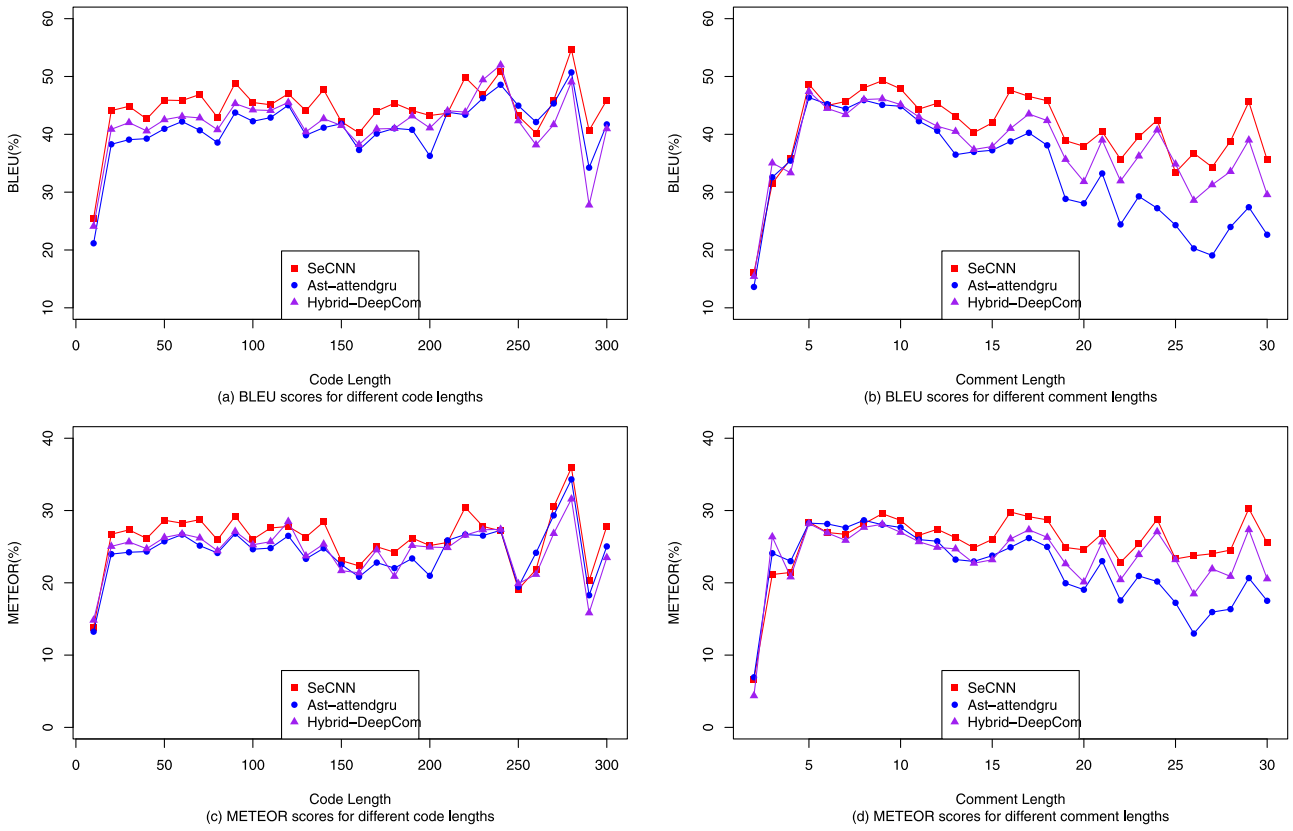

Fig. 8. *BLEU* and *METEOR* scores of different approaches with different code length and comment length.

Table 9
Cliff's delta under different code lengths and comment lengths.

	Approach	*BLEU*	*METEOR*
Code length	Hybrid-DeepCom	0.41	0.25
	AST-attendgru	0.49	0.17
Comment length	Hybrid-DeepCom	0.33	0.29
	AST-attendgru	0.47	0.29

in Table 9 indicate that the performance of SeCNN has a significant difference with other baselines under different code and comment lengths in terms of *BLEU* metric.

Summary for RQ2: When considering different lengths of code and comments, SeCNN achieves better performance than Hybrid-DeepCom and AST-attendgru in most cases. Besides, the experiment results also show that SeCNN performs the worst when code or comments are too short, and the reason is that short source code or comments are often incomplete, which will affect the performance of code comment generation techniques.

5.3. RQ 3: What is the difference between comments generated by SeCNN and human-written comments?

In this section, we want to analyze the difference between comments generated by code comment generation techniques and written by human. Since automatic evaluation metrics cannot fully reflect the actual quality of the results, we conducted a manual evaluation study to evaluate the quality of automatically generated code comments. As discussed in RQ1, we re-implement three code comment generation techniques, DeepCom, Hybrid-DeepCom and AST-attendgru, where Hybrid-DeepCom shows the best performance in these three techniques. So in this section, we only employ Hybrid-DeepCom as one baseline of code comment generation technique to perform comparison. Due to the high cost of manually analyzing all these code comments, we use a commonly-used sampling method (Singh and Mangat, 2013) to select the minimum number *MIN*[6] of code comments. The value of *MIN* is calculated by the following formula:

$$MIN = \frac{n_0}{1 + \frac{n_0 - 1}{populationsize}} \tag{15}$$

$$n_0 = \frac{Z^2 \times 0.25}{e^2} \tag{16}$$

where the value of *populationsize* is 8714, which is the total number of code comments. Z is a confidence level and e is the error margin. In our experiment, we set the value of Z to 95% and the value of e to 0.05. Then the calculated value of *MIN* is 384, which means we should obtain 384 groups of scores from human evaluation for SeCNN and Hybrid-DeepCom Dataset respectively.

We recruited five volunteers with rich development experience (including teachers, professional master and doctoral students) to provide feedback for our comparison. Among them, we followed the guidelines of experimental design[7] and did a within-subjects experiment, because every volunteer will answer the same questions under the same circumstances.

Specifically, we randomly selected 384 pairs of prediction results and their references from the test set. Therefore, our questionnaire has a total of 384 pages, and each page consists of

6 https://www.surveysystem.com/sscalc.htm#one.
7 https://opentextbc.ca/researchmethods/chapter/experimental-design/.

Table 10
One page in the questionnaire of user study.

Answer questions for the following code:

```
private static int exitWithStatus(int status) {
    if (ToolIO.getMode() == ToolIO.SYSTEM){
        System.exit(status);
    }
    return status ;
}
```

Reference comment: if run in the system mode , exits the program , in tool mode returns the status.

Candidate 1: The method exists , which means the status is null and remove status.	
Please evaluate the naturalness of Candidate 1:	Score from 1 to 5 (5 is the best)
Please evaluate the relevance of Candidate 1:	Score from 1 to 5 (5 is the best)
Candidate 2: Returns a hash failure.	
Please evaluate the naturalness of Candidate 2:	Score from 1 to 5 (5 is the best)
Please evaluate the relevance of Candidate 2:	Score from 1 to 5 (5 is the best)

Table 11
Manual analysis.

Type	SeCNN	Hybrid-DeepCom
Naturalness	**3.78**	3.53
Relevance	**3.73**	3.51

an input source code, comments generated by SeCNN and Hybrid-DeepCom, and a hand-written reference comment. Then we send each copy of the 384-page questionnaire to each volunteer, and volunteers were required to evaluate two comments for each code. Furthermore, to ensure fairness, the order of the comments generated by the two methods is random on each page, and we delete their tags to ensure that volunteers cannot distinguish comments generated by Hybrid-DeepCom or SeCNN. Our manual evaluation allowed volunteers to search for relevant information and unfamiliar concepts on the Internet.

To allow volunteers to evaluate the quality of generated comments from different views, we followed Gao et al. (2020) to consider two modalities, naturalness and relevance. Naturalness refers to the grammatical correctness and fluency of the generated comments, that is, whether the text of a comment is easy for humans to read and understand; Relevance refers to the correlation between the generated comments and the input code, that is, can humans understand the intention of the code based on this comment. One page of our questionnaire as shown in Table 10, volunteers need to read the input code, reference comment, and two generated comments. Then score the two generated comments' naturalness and relevance, with a score ranging from 1 to 5 (5 is the best).

Finally, we calculated the average value of the five volunteers' feedback, and the results can be found in Table 11. For example, the value in the second row and the second column indicates that the average relevance score of Hybrid-DeepCom is 3.51. It can be found that the average naturalness and relevance score of SeCNN outperform Hybrid-DeepCom 0.25 and 0.22 respectively, which indicates that the volunteers have a higher agreement of comments generated by SeCNN methods. Besides, we also use Cliff's Delta to quantify the difference between SeCNN and Hybrid-DeepCom in terms of comments' naturalness and relevance. The Cliff's Delta value is 0.12 for naturalness and 0.11 for relevance, which means that the performance of SeCNN has only a weak advantage for volunteers.

Summary for RQ3:
We conducted a manual evaluation of code comments generated by SeCNN and Hybrid-DeepCom methods. Experimental results show that SeCNN can generate more comments with better naturalness and relevance.

5.4. Discussions

5.4.1. Discussion on out-of-vocabulary problem

Word embedding encodes the relationships between words through vector representations of the words. However, due to a large number of identifiers defined in the source code, the out-of-vocabulary problem is one of the challenges of neural network based code comment generation techniques (Hellendoorn and Devanbu, 2017). Specifically, in natural language processing or text processing, we usually have a vocabulary. This vocabulary is either loaded in advance, or defined by researchers, or extracted from the current data set. On the other hand, if there are some words in the data set but not in the existing vocabulary, then these words are out-of-vocabulary (IssamBazzi and Glass, 2000). In natural language processing area, an effective method to handle out-of-vocabulary problem is limiting vocabulary to the most common words during data processing. For example, if the size of words in vocabulary is more than 30,000, such method will only use the top 30,000 common words, and replace other words by a special token ⟨nuk⟩.

However, due to the large number of user-defined identifiers in the source code, which makes the number of unique tokens in source code very large, this method cannot be used directly in the source code manipulation area. Moreover, we find that the identifiers usually consist of several common words. Therefore, to alleviate the out-of-vocabulary problem, we use camel casing conversion to split the identifiers of both code tokens and AST nodes into several words. After that, the number of unique words in the code token vocabulary and AST node vocabulary become much smaller. More specifically, in our study, we decrease the number of unique words in the code token vocabulary from 308,920 to 32,901, and decrease the number of unique words in AST node vocabulary from 322,933 to 32,570.

Bojanowski et al. (2017) used subword information to solve the out-of-vocabulary problem, which method is very effective in natural language processing. However, this method cannot handle the problem of compound words in the code. Gao et al. (2020) used the copy mechanism to solve the problem of out-of-vocabulary words in output words. Different from these previous studies, we need to solve the problem of out-of-vocabulary words in input words. To verify that the identifier segmentation can improve the performance of SeCNN, we compare the performance of SeCNN with identifier segmentation and SeCNN without identifier segmentation. Table 13 shows using identifier segmentation can achieve better performance than not using identifier segmentation, and the improvement is 0.63% and 1.131% in terms of *BLEU* and *METEOR* respectively.

Table 12
Comparison of the running time of three epochs.

Approach	Time (minute)			
	First epoch	Second epoch	Third epoch	Average
DeepCom	18.34	18.23	18.21	18.26
Hybrid-DeepCom	18.56	18.48	18.50	18.51
AST-attendgru	62.55	61.39	61.29	61.74
SeCNN (Original SBT)	18.51	18.53	18.53	18.52
SeCNN	**14.11**	**13.98**	**14.15**	**14.08**

Table 13
The average performance of SeCNN with identifier segmentation and SeCNN without identifier segmentation.

Approach	*BLEU* (%)	*METEOR* (%)
Without identifier segmentation	44.06	25.57
With identifier segmentation	**44.69**	**26.88**

5.4.2. Discussion on training time of code comment generation techniques

We further discuss the training time of SeCNN and other baselines. Since Hybrid-eepcom and AST-attendgru use the code tokens and SBT sequences as the input like SeCNN, so in this subsection, we choose them as two baselines. Besides, to verify the effectiveness of ISBT, we also compare the training time of SeCNN with ISBT sequence replace and SeCNN with original SBT sequence. Table 12 shows the training time of three epochs of these four techniques. Since the training time of each epoch of the neural network is similar, we choose the first three epochs for comparison.

As shown in Table 12, SeCNN with ISBT has the least training time. Compared with SeCNN (original SBT), we find that using ISBT can reduce the training time of SeCNN by about 23.97%, which proves that using ISBT can obviously improve the efficiency of SeCNN.

Compared with Hybrid-DeepCom, the training time of SeCNN is reduced by about 23.93%. In the encoder stage of the Hybrid-DeepCom, it uses a layer of GRU network, while SeCNN (original SBT) uses a 9-layer CNN network. The experimental results show that CNN needs less training time than GRU. Compared with DeepCom, the training time of SeCNN is reduced by about 22.89%. The experimental results show that CNN needs less training time than GRU. Besides, it also can be seen that AST-attendgru has the longest training time, and the reason is that it predicts one word at a time and does not use RNN-based decoding to generate code comments.

5.4.3. Discussion on the influence of the hidden size on SeCNN

The hidden size is an important parameter of the neural network and has a significant impact on the performance of the trained neural network model. This parameter includes the hidden state of the LSTM and the size of the word embedding vector. In this section, we want to discuss the impact of this parameter on the performance of SeCNN. Fig. 9 shows the performance with different parameter values. To ensure the fair comparison of the experiment, the value of parameters is set to the same except the hidden size parameter.

Fig. 9 shows that the value of *BLEU* and *METEOR* increases with the increase of the hidden size. That means, increasing the value of hidden size can effectively improve the performance of our proposed method SeCNN. Moreover, We also find that as the value of the hidden size increases, the growth rate of *BLEU* and *METEOR* becomes slower, and when the value of hidden size increases to 500, the performance of the trained model almost converges. There is no doubt that the increase in the hidden size can increase the cost of model training. Therefore, we comprehensively set the value of the hidden size to 500 in our experimental study.

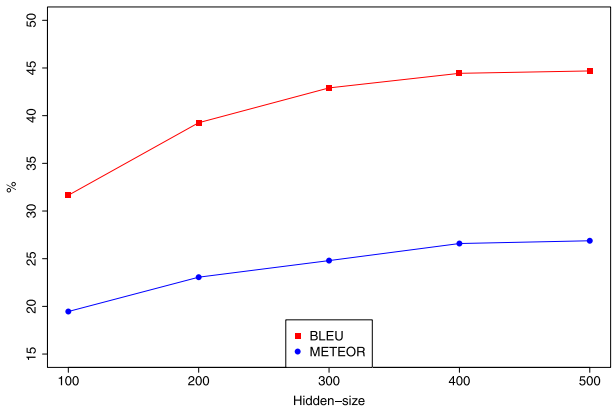

Fig. 9. Different hidden-size performance on SeCNN.

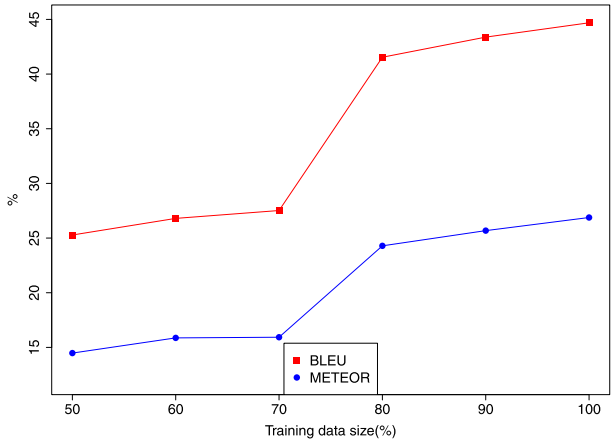

Fig. 10. SeCNN performance on different amounts of training data.

5.4.4. Discussion on the impact of the size of training data on SeCNN

Deep learning often requires a reasonable size of the training data. If the size of the training data is too small, the neural network model may not achieve dependable performance, which will lead to inaccurate comments generation. To know how much data is needed before SeCNN can work, we further discuss the impact of the training set size on the performance of SeCNN. Fig. 10 shows the performance of SeCNN for different sizes of the training data. To ensure the fair comparison in our study, we set all parameters the same value except for the size of the training data.

In Fig. 10, we can find that with the increase of the training data, the performance of SeCNN is getting better. Because the more training data, the easier it is for SeCNN to learn the relationship between code and comments. Moreover, we find that when the size of the training data is less than 80% (55,766), the performance of SeCNN is very poor, which also shows that at least 80% (55,766) of the training data is required before SeCNN can work efficiently.

5.4.5. Discussion on the impact of the network depth

Furthermore, we also conduct empirical studies to investigate the impact of the setting of network depth parameter on the model performance, since recent studies show that the network depth is of crucial importance (He et al., 2016; Simonyan and Zisserman, 2014; Szegedy et al., 2015). Fig. 11 lists the performance results of SeCNN with different network depths. From Fig. 11, it can be found that the setting of the network depth parameter has a slight impact on the performance of SeCNN.

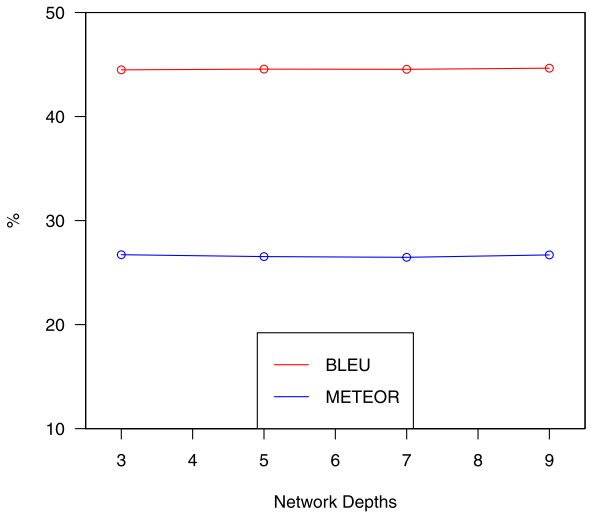

Fig. 11. Different network depths performance on SeCNN.

6. Threats to validity

In this section, we discuss the potential threats to our study.

6.1. Internal validity

One threat to internal validity is the implementation errors of our code. To alleviate this issue, we have carefully performed code inspection and software testing on our code. Moreover, we also release our source code for other researchers to replicate our study. Another threat to internal validity is the bias in the re-implementation of the baselines. To alleviate this threat, we re-implemented some baselines by using the same experimental setting as the baselines. However, their settings may not be suitable for the dataset we use.

Besides, threats concern factors internal to our study that could affect the empirical results. The performance of our method may depend on the hyperparameter configuration. Hyperparameter settings of our proposed method mainly come from optimization on the validation set and baselines' hyperparameter value, which we discussed in Section 4.6.

6.2. Construct validity

The first construct validity related to metrics. To reduce the impact of evaluation measures, we used two machine translation metrics that are *BLEU* and *METEOR*. *BLEU* and *METEOR* have been widely used in the machine translation tasks.

The second construct validity related to the dataset. The dataset used in the paper has 9.7k items, and each item is a ⟨code, comment⟩ pair. However, we cannot trace back items to the projects due to data limitations. Therefore, the proposed approach might work remarkably well for certain projects and poorly for others.

The third construct validity is related to the quality of comments. To mitigate the impact of code quality, we just select the comment about the Java method in the first sentence of Javadoc, just like the previous study on code comment generation. However, explaining a piece of code often requires a lot of comments. The chosen dataset was collected by Hu et al. (2018b) on GitHub. Hu et al. (2018b) used heuristic rules to reduce noise in comments. Since the comments were not updated in time after the code was updated, there are still some code and comments that do not match the dataset.

6.3. Conclusion threats

The conclusion threat is related to the dataset splitting method. We split the dataset into the training set, the validation set and the test set and the percentage are 80%, 10%, and 10%. The split strategy is consistent with the previous code comment generation studies (Hu et al., 2018a,b; LeClair et al., 2019; Wei et al., 2019). Moreover, to alleviate possible sampling bias in random shuffle and sampling, we conducted three experiments independently, and we redivided each experiment randomly.

The second conclusion validity is related to our experiment times. Especially, we run the experiment three times. Conducting more experiments can increase the generalization of the experimental results, but it will take too much time to conduct a ten-fold cross-validation technique with our existing GPU resources. Therefore, our experiment settings are consistent with many existing studies in this field (Hu et al., 2018a; LeClair et al., 2019; Hu et al., 2019), so the threat of not using the ten-fold cross-validation model validation technique is limited.

7. Related work

7.1. Deep code representation

Deep learning algorithms are commonly used in the field of image processing (Ren et al., 2015) and natural language processing (Bao et al., 2019). In recent years, many researchers try to employ deep learning algorithms to solve software engineering problems, where deep code representation is one of the main challenge (Zhang et al., 2019).

Many researchers have proposed a set of approaches to do deep code representation, which include attention unit (Iyer et al., 2016), sequence-to-sequence (Gu et al., 2016), and CNN (Mou et al., 2016), etc. For example, Iyer et al. (2016) used attention unit to calculate the distribution representation of code segment and then employed neural networks to generate code comments. Gu et al. (2016) used a sequence-to-sequence model to learn medium vector representations of code-related natural language queries, which are further utilized with neural networks to predict related API sequences. Li et al. (2015) used heap nodes to learn distributed vector representations, and then utilized them to synthesize candidate formal specifications for the code that generates the heap. Mou et al. (2016) employed a custom CNN to learn features form code fragments and obtained a distributed vector representation, and then they tried to restore the solution to the problem mapping by classification.

These previous approaches are transferred from natural language processing approaches. Different with these problems, code representation is a kind of long-dependency problem, where traditional natural language processing approaches have limitations on solving it. In our study, we use source code tokens-based CNN and AST-based CNN to capture semantic information from source code, which can effectively alleviate the long-dependency problem.

7.2. Language models for source code

In recent years, language models for source code become the fundamental part and have been successfully used in many software engineering tasks, including fault detection (Ray et al., 2016), code summarization (Iyer et al., 2016), code clone detection (Yu et al., 2019; White et al., 2016), program repair (Gupta et al., 2017), and code generation (Rabinovich et al., 2017).

Hindle et al. (2012) are the first to propose a n-gram based language model for source code, and they also proved that such model can be used in most software. Allamanis et al. (2014)

developed a framework that learns the code rules of a code base and then used the n-gram model to name Java identifiers. Gu et al. (2016) used sequence-to-sequence deep neural networks to learn medium vector representations of natural language queries, which are used to predict related API sequences. Yin and Neubig (2017) established a grammar-based neural network model for automatic code generation.

In our study, we propose a novel approach SeCNN, which designs several novel components to learn the semantic information from source code and form an effective neural network-based language model to generate comments for Java methods.

7.3. Code summarization

Code Summarization is a novel task in software engineering, which aims to generate natural language descriptions for source code (Wong et al., 2015; Sridhara et al., 2010; LeClair et al., 2020; Fowkes et al., 2017; Movshovitz-Attias and Cohen, 2013; Shido et al., 2019). In recent years, many researchers pay a lot of attention and propose many approaches to handle this problem. These approaches can be divided into two groups, template-based code summarization (Haiduc et al., 2010b,a; Rodeghero et al., 2015; Mcburney and Mcmillan, 2016; Hill et al., 2009) and AI-based code summarization (Iyer et al., 2016; Wan et al., 2018; Hu et al., 2018a,b, 2019; Wei et al., 2019; Ye et al., 2020).

In template-based automatic code comment generation approaches, the researchers defined a set of templates and populated them by the type of target code segment and other information. Haiduc et al. (2010b,a) attempted to generate code comments by calculating the top-n keywords with metrics such as TF/IDF. Rodeghero et al. (2015) further improved the content extraction of heuristics and templating solutions by modifying heuristics to mimic how human developers read code with their minds. Mcburney and Mcmillan (2016) proposed a Software Word Usage Model (SWUM) to generate code comments (Hill et al., 2009).

AI-based code comment generation approaches use neural network algorithms and machine translation models. Iyer et al. (2016) are the first to use neural networks to generate code comments. They developed a new method called CODE-NN that utilizes RNNs and distributes annotated words directly to code tokens. CODE-NN can successfully recommend natural language descriptions corresponds to source code snippets collected from Stack Overflow. Wan et al. (2018) employed deep reinforcement learning framework to handle code representation and exposured bias problems during the code summarization process. Hu et al. (2018a) proposed a novel approach DeepCom to generate descriptive comments for Java methods. DeepCom uses a new Structure-Based Traversal method to traverse AST and employs Neural Machine Translation (NMT) to train the code comment generation model. Hu et al. (2018b) proposed an approach TL-CodeSum, which generates summaries for Java methods with the assistance of transferred API knowledge learned from another task of API sequences summarization. Moreover, Hu et al. (2019) extended their work and proposed a deep neural network to generate code comments for Java methods. Their proposed approach Hybrid-DeepCom combines the lexical and structure information of source code and shows better performance than other techniques. Liu et al. (2019) extracted call dependencies from source code and its related code, and used a model based on Seq2Seq to generate code summarization from source code and call dependencies. Haque et al. (2020) combined the file context of the subroutine and the code of the subroutine to enhance the automatic summary method of the subroutine. The latest approach CO3 was proposed by Ye et al. (2020). This approach is based on an end-to-end model, which employs dual learning and multi-task learning to improve code summarization and code retrieval.

Our proposed approach SeCNN is a kind of AI-based code comment generation approach. However, different from the previous approaches, SeCNN uses two CNNs (i.e., source code-based CNN and AST-based CNN) to alleviate long-term dependency problems and learn semantic information of source code. Empirical results also verify the effectiveness of our proposed approach.

8. Conclusion and future work

In this paper, we propose a novel neural network based technique, SeCNN, for generating code comment of Java methods. Compared with the existing researches, our method has improved the quality of generated comments. However, in this paper, the accomplishments of our proposed approach cannot make it ready in practice. In the future, we will try to further improve the quality of automated generated code comments. SeCNN uses CNN to alleviate the long-dependency problem of source code manipulation, and contains several novel components to capture the semantic information of source code. In particular, we design a source code-based CNN component to learn the lexical information and an AST-based component to learn the syntactical information. Then, we use LSTM with an attention mechanism to decoder and generate code comments. Comprehensive experiments are conducted on a widely-studied large-scale dataset of 87,136 Java methods, and the results show that SeCNN performs better than five state-of-the-art baselines. More specifically, SeCNN achieves the best performance in terms of *BLEU* and *METEOR* metrics. Compared with other two similar baselines, Hybrid-DeepCom and AST-attendgru, which also use source code and AST, SeCNN shows better performance in terms of efficiency, whose execution time cost is apparently lower than other two baselines.

In the future, we want to employ program analysis tools to gather richer information and combine other deep learning-based approaches to further improve the effectiveness of our proposed code comment generation approach.

CRediT authorship contribution statement

Zheng Li: Conceptualization, Supervision, Project administration, Funding acquisition. **Yonghao Wu:** Formal analysis, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Bin Peng:** Software, Validation, Formal analysis, Investigation, Data curation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Xiang Chen:** Methodology, Resources, Writing - Original Draft, Writing - Review & Editing, Supervision, Project administration, Funding acquisition. **Zeyu Sun:** Resources, Writing - Original Draft, Writing - Review & Editing. **Yong Liu:** Methodology, Resources, Writing - Original Draft, Writing - Review & Editing, Supervision, Project administration, Funding acquisition. **Deli Yu:** Investigation, Writing - Original Draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work described in this paper is supported by the National Natural Science Foundation of China (Grant nos. 61902015 and 61872026), the Nantong Application Research Plan, China (Grant no. JC2019106), and the Open Project of Key Laboratory of Safety-Critical Software for Nanjing University of Aeronautics and Astronautics, China, Ministry of Industry and Information Technology, China (Grant No. NJ2020022).

References

Allamanis, M., Barr, E.T., Bird, C., Sutton, C.A., 2014. Learning natural coding conventions. In: Cheung, S., Orso, A., Storey, M.D. (Eds.), Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. ACM, pp. 281–293. http://dx.doi.org/10.1145/2635868.2635883.

Allamanis, M., Peng, H., Sutton, C.A., 2016. A convolutional attention network for extreme summarization of source code. In: Balcan, M., Weinberger, K.Q. (Eds.), Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. In: JMLR Workshop and Conference Proceedings, vol. 48, JMLR.org, pp. 2091–2100, URL http://proceedings.mlr.press/v48/allamanis16.html.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. Comput. Sci. 1409.

Banerjee, S., Lavie, A., 2005. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: Goldstein, J., Lavie, A., Lin, C., Voss, C.R. (Eds.), Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/Or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005. Association for Computational Linguistics, pp. 65–72, URL https://www.aclweb.org/anthology/W05-0909/.

Bao, L., Lambert, P., Badia, T., 2019. Attention and lexicon regularized LSTM for aspect-based sentiment analysis. In: Alva-Manchego, F.E., Choi, E., Khashabi, D. (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28 - August 2, 2019, Volume 2: Student Research Workshop. Association for Computational Linguistics, pp. 253–259. http://dx.doi.org/10.18653/v1/p19-2035.

Bengio, Y., Simard, P.Y., Frasconi, P., 1994. Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. 5 (2), 157–166. http://dx.doi.org/10.1109/72.279181.

Bojanowski, P., Grave, E., Joulin, A., Mikolov, T., 2017. Enriching word vectors with subword information. Trans. Assoc. Comput. Linguist. 5, 135–146, URL https://transacl.org/ojs/index.php/tacl/article/view/999.

Chung, J., Gülçehre, Ç., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR abs/1412.3555. arXiv:1412.3555 URL http://arxiv.org/abs/1412.3555.

Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions.. Psychol. Bull. 114 (3), 494.

Denkowski, M.J., Lavie, A., 2014. Meteor universal: Language specific translation evaluation for any target language. In: Proceedings of the Ninth Workshop on Statistical Machine Translation, WMT@ACL 2014, June 26-27, 2014, Baltimore, Maryland, USA. The Association for Computer Linguistics, pp. 376–380. http://dx.doi.org/10.3115/v1/w14-3348.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arxiv preprint arxiv:1810.04805.

Engelken, R., Wolf, F., Abbott, L., 2020. Lyapunov Spectra of chaotic recurrent neural networks. arxiv preprint arxiv:2006.02427.

Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., Sutton, C., 2017. Autofolding for source code summarization. IEEE Trans. Softw. Eng. 43 (12), 1095–1109.

Gao, Z., Xia, X., Grundy, J.C., Lo, D., Li, Y., 2020. Generating question titles for stack overflow from mined code snippets. CoRR abs/2005.10157. arXiv:2005.10157 URL https://arxiv.org/abs/2005.10157.

Gehring, J., Auli, M., Grangier, D., Dauphin, Y.N., 2017. A convolutional encoder model for neural machine translation. In: Barzilay, R., Kan, M. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers. Association for Computational Linguistics, pp. 123–135. http://dx.doi.org/10.18653/v1/P17-1012.

Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep API learning. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (Eds.), Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. ACM, pp. 631–642. http://dx.doi.org/10.1145/2950290.2950334.

Gupta, R., Pal, S., Kanade, A., Shevade, S.K., 2017. Deepfix: Fixing common c language errors by deep learning. In: Singh, S.P., Markovitch, S. (Eds.), Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. AAAI Press, pp. 1345–1351, URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603.

Hahnloser, R.H., Sarpeshkar, R., Mahowald, M.A., Douglas, R.J., Seung, H.S., 2000. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature 405 (6789), 947–951.

Haiduc, S., Aponte, J., Marcus, A., 2010a. Supporting program comprehension with source code summarization. In: Proceedings - International Conference on Software Engineering.

Haiduc, S., Aponte, J., Moreno, L., Marcus, A., 2010b. On the use of automated text summarization techniques for summarizing source code. In: 2010 17th Working Conference on Reverse Engineering. IEEE, pp. 35–44.

Hansel, D., Van Vreeswijk, C., 2002. How noise contributes to contrast invariance of orientation tuning in cat visual cortex. J. Neurosci. 22 (12), 5118–5128.

Haque, S., LeClair, A., Wu, L., McMillan, C., 2020. Improved automatic summarization of subroutines via attention to file context. MSR 2020.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, pp. 770–778. http://dx.doi.org/10.1109/CVPR.2016.90.

Hellendoorn, V.J., Devanbu, P.T., 2017. Are deep neural networks the best choice for modeling source code?. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (Eds.), Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. ACM, pp. 763–773. http://dx.doi.org/10.1145/3106237.3106290.

Hill, E., Pollock, L.L., Vijay-Shanker, K., 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. IEEE, pp. 232–242. http://dx.doi.org/10.1109/ICSE.2009.5070524.

Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software. In: 2012 34th International Conference on Software Engineering (ICSE). Proceedings - International Conference on Software Engineering 837–847.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018a. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension. In: ICPC '18, Association for Computing Machinery, New York, NY, USA, pp. 200–210. http://dx.doi.org/10.1145/3196321.3196334.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2019. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 1–39.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018b. Summarizing source code with transferred API knowledge. In: Lang, J. (Ed.), Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. ijcai.org, pp. 2269–2275. http://dx.doi.org/10.24963/ijcai.2018/314.

IssamBazzi, Glass, J., 2000. Modeling out-of-vocabulary words for robust speech recognition. In: Proc Icslp Oct.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2073–2083. http://dx.doi.org/10.18653/v1/P16-1195.

Kadmon, J., Sompolinsky, H., 2015. Transition to chaos in random neuronal networks. Phys. Rev. X 5 (4), 041030.

Kalchbrenner, N., Grefenstette, E., Blunsom, P., 2014. A convolutional neural network for modelling sentences. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 1: Long Papers. The Association for Computer Linguistics, pp. 655–665. http://dx.doi.org/10.3115/v1/p14-1062.

Karampatsis, R., Babii, H., Robbes, R., Sutton, C.A., Janes, A., 2020. Big code != big vocabulary: Open-vocabulary models for source code. CoRR abs/2003.07914. arXiv:2003.07914 URL https://arxiv.org/abs/2003.07914.

Kim, Y., 2014. Convolutional neural networks for sentence classification. Eprint Arxiv.

Klein, G., Kim, Y., Deng, Y., Senellart, J., Rush, A.M., 2017. Opennmt: Open-source toolkit for neural machine translation. In: Bansal, M., Ji, H. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, System Demonstrations. Association for Computational Linguistics, pp. 67–72. http://dx.doi.org/10.18653/v1/P17-4012.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2017. Imagenet classification with deep convolutional neural networks. Commun. ACM 60 (6), 84–90. http://dx.doi.org/10.1145/3065386, URL http://doi.acm.org/10.1145/3065386.

LeClair, A., Haque, S., Wu, L., McMillan, C., 2020. Improved code summarization via a graph neural network. arxiv preprint arxiv:2004.02843.

LeClair, A., Jiang, S., McMillan, C., 2019. A neural model for generating natural language summaries of program subroutines. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, pp. 795–806. http://dx.doi.org/10.1109/ICSE.2019.00087.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015. Gated graph sequence neural networks. Comput. Sci..

Libovický, J., Helcl, J., 2018. End-to-end non-autoregressive neural machine translation with connectionist temporal classification. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018. Association for Computational Linguistics, pp. 3016–3021. http://dx.doi.org/10.18653/v1/d18-1336.

Liu, B., Wang, T., Zhang, X., Fan, Q., Yin, G., Deng, J., 2019. A neural-network based code summarization approach by using source code and its call dependencies. In: Proceedings of the 11th Asia-Pacific Symposium on Internetware, pp. 1–10.

Luong, T., Pham, H., Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. In: Màrquez, L., Callison-Burch, C., Su, J., Pighin, D., Marton, Y. (Eds.), Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015. The Association for Computational Linguistics, pp. 1412–1421. http://dx.doi.org/10.18653/v1/d15-1166.

Mcburney, P.W., Mcmillan, C., 2016. Automatic source code summarization of context for java methods. IEEE Trans. Softw. Eng. 42 (2), 103–119.

Moreno, L., Marcus, A., Pollock, L.L., Vijay-Shanker, K., 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In: IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013. IEEE Computer Society, pp. 230–232. http://dx.doi.org/10.1109/ICPC.2013.6613855.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Schuurmans, D., Wellman, M.P. (Eds.), Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA. AAAI Press, pp. 1287–1293, URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775.

Movshovitz-Attias, D., Cohen, W., 2013. Natural language models for predicting programming comments. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pp. 35–40.

Ott, R.L., Longnecker, M.T., 2015. An Introduction to Statistical Methods and Data Analysis. Nelson Education.

Papineni, K., Roukos, S., Ward, T., Zhu, W., 2002. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA. ACL, pp. 311–318, URL https://www.aclweb.org/anthology/P02-1040/.

Rabinovich, M., Stern, M., Klein, D., 2017. Abstract syntax networks for code generation and semantic parsing. In: Barzilay, R., Kan, M. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers. Association for Computational Linguistics, pp. 1139–1149. http://dx.doi.org/10.18653/v1/P17-1105.

Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P.T., 2016. On the "naturalness" of buggy code. In: Dillon, L.K., Visser, W., Williams, L. (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. ACM, pp. 428–439. http://dx.doi.org/10.1145/2884781.2884848.

Ren, S., He, K., Girshick, R.B., Sun, J., 2015. Faster r-CNN: towards real-time object detection with region proposal networks. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. pp. 91–99.

Rodeghero, P., Liu, C., McBurney, P.W., McMillan, C., 2015. An eye-tracking study of java programmers and application to source code summarization. IEEE Trans. Softw. Eng. 41 (11), 1038–1054.

Shen, Y., He, X., Gao, J., Deng, L., Mesnil, G., 2014. Learning semantic representations using convolutional neural networks for web search. In: Chung, C., Broder, A.Z., Shim, K., Suel, T. (Eds.), 23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume. ACM, pp. 373–374. http://dx.doi.org/10.1145/2567948.2577348.

Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., Matsumura, T., 2019. Automatic source code summarization with extended tree-LSTM. In: International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019. IEEE, pp. 1–8. http://dx.doi.org/10.1109/IJCNN.2019.8851751.

Simonyan, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arxiv preprint arxiv:1409.1556.

Singh, R., Mangat, N.S., 2013. Elements of Survey Sampling, Vol. 15. Springer Science & Business Media.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K., 2010. Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 43–52.

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L., 2019a. A grammar-based structural cnn decoder for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33, pp. 7055–7062.

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L., 2019b. A grammar-based structural CNN decoder for code generation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, the Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, the Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press, pp. 7055–7062. http://dx.doi.org/10.1609/aaai.v33i01.33017055.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada. pp. 3104–3112, URL http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2015. Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. IEEE Computer Society, pp. 1–9. http://dx.doi.org/10.1109/CVPR.2015.7298594.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, pp. 2818–2826. http://dx.doi.org/10.1109/CVPR.2016.308.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Huchard, M., Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. ACM, pp. 397–407. http://dx.doi.org/10.1145/3238147.3238206.

Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z., 2019. Code generation as a dual task of code summarization. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. pp. 6559–6569, URL http://papers.nips.cc/paper/8883-code-generation-as-a-dual-task-of-code-summarization.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: Lo, D., Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. ACM, pp. 87–98. http://dx.doi.org/10.1145/2970276.2970326.

Williams, R.J., Zipser, D., 1989. A learning algorithm for continually running fully recurrent neural networks. Neural Comput. 1 (2), 270–280. http://dx.doi.org/10.1162/neco.1989.1.2.270.

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

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2017. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. Softw. Eng. 44 (10), 951–976.

Ye, W., Xie, R., Zhang, J., Hu, T., Wang, X., Zhang, S., 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In: Proceedings of the Web Conference 2020, pp. 2309–2319.

Yin, P., Neubig, G., 2017. A syntactic neural model for general-purpose code generation. In: Barzilay, R., Kan, M. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers. Association for Computational Linguistics, pp. 440–450. http://dx.doi.org/10.18653/v1/P17-1041.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: Guéhéneuc, Y., Khomh, F., Sarro, F. (Eds.), Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, pp. 70–80. http://dx.doi.org/10.1109/ICPC.2019.00021.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zhou, Y., Yan, X., Yang, W., Chen, T., Huang, Z., 2019. Augmenting java method comments generation with context information based on neural networks. J. Syst. Softw. 156, 328–340.

Zheng Li is a professor at BUCT. He obtained his Ph.D. from King's College London, CREST center in 2009 under the supervision of Mark Harman. He has worked as a research associate at King's College London and University College London. He has worked on program testing, source code analysis and manipulation. More recently, he is interested in search-based software engineering and slicing state-based model.

Yonghao Wu is a third-year graduate student at Beijing University of Chemical Technology (BUCT). He has worked on program fault localization. His research interests are software testing, multiple fault localization and fault understanding.

Bin Peng is a second-year graduate student at Beijing University of Chemical Technology (BUCT). He has worked on source code analysis. His research interests are software testing, code comments generation and software defect prediction.

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received the M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is with the Department of Information Science and Technology at Nantong University as an associate professor. His research interests are mainly in software engineering. In particular, he is interested in empirical software engineering, mining software repositories, software maintenance and software testing. In these areas, he has published over 60 papers in referred journals or conferences, such as IEEE Transactions on Software Engineering, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software Quality Journal, Journal of Computer Science and Technology, ASE, ICSME, SANER and COMPSAC. He is a senior member of CCF, China and a member of IEEE and ACM.

Zeyu Sun is a Ph.D. student in School of Electronics Engineering and Computer Science (EECS), Peking University. His research interests are Software Testing, Code Generation, Code Representation, and Deep Learning Testing.

Yong Liu is an assistant professor at Beijing University of Chemical Technology (BUCT). He obtained his PhD from BUCT in 2018 under the supervision of Professor Zheng Li. His research interests include software testing, fault localization, fault understanding, and mutation testing.

Deli Yu is a second-year graduate student at Beijing University of Chemical Technology (BUCT). He has worked on student program analysis. His research interests are software testing, fault localization and fault understanding.