SeCNN: A Semantic CNN Parser for Code Comment Generation

Zheng Li[a], Bin Peng[a], Xiang Chen[c], Zeyu Sun[b,*], Yong Liu[a,*], Yonghao Wu[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;*
Software Institute, Peking University, 100871, P. R. China
[c]*School of Information Science and Technology, Nantong University, Nantong, China*

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.

Keywords: Program Comprehension, Code Comment Generation, Convolutional Neural Network, Long Short-Term Memory Network

1. Introduction

During the process of software development and maintenance, developers spend nearly 60% time on source code comprehension [1]. 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tight project schedule [2], a large number of software projects do not provide complete comments, or some comments are outdated due to software updates.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 [3, 4, 5, 6]. Although significant progress has been made based on keywords and sentence templates selection, these template-based approaches still have limitations [7].

With the development of AI-based approaches, more studies tend to apply the encoder-decoder framework [8] 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 [9]. However, as a strict structural text, source code contains rich structural information, which is important for program modeling [10]. To address this issue, state-of-the-art approaches generate the comment based on the abstract syntax tree (AST) of the code via RNNs [2, 7].

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 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. **SeCNN** uses a CNN, which captures features effectively among different code blocks, to alleviate the long-dependency problem. 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 to alleviate the out-of-

[*]Corresponding authors
Email address: `lyong@mail.buct.edu.cn, szy_@pku.edu.cn` (Yong Liu)

vocabulary word 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 [11]. Experimental results show that SeCNN achieves better performance in terms of *BLEU* [12], *METEOR* [13], *Precision*, and *F-score* 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 word 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[1].

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.

2. Background

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

[1]The website will be given when this manuscript is accepted.

2.1. Language Model

Code comment generation techniques are inspired by the techniques used in text generation task of NLP field [14]. The language model used to generate code comments is learned from code corpus. More specifically, for a sequence $X = (x_1, x_2, \cdots, x_n)$, the language model aims to estimate the probability of this sequence, and the corresponding calculation formula [2] can be defined as follows:

$$P(X) = \prod_{i=1}^{n} P(x_i | x_1, \cdots, x_{i-1}) \tag{1}$$

In previous studies on code comment generation, Recurrent Neural Network (RNN) [2] and Long Short-Term Memory Model (LSTM) [15] 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 [2, 7, 16]. Standard RNN takes sequence data as the input, and then it will perform recursion in the evolution direction of the sequence. Finally, Standard RNN connects all nodes (i.e., cyclic units) in a chain. Figure 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 first calculates a hidden state h_t according to the previous hidden state h_{t-1}, where the formula is defined as follows:

$$h_t = tanh(Wx_t + Uh_{t-1} + b) \tag{2}$$

where W, U and b are parameters to be trained and will be updated during the training process, and *tanh* is an activation function.

Then, it estimates the probability $p(y_{t+1}|y_{1:t})$ of the following word, which can be calculated as follows:

$$p(y_{t+1}|y_{1:t}) = g(h_t) \tag{3}$$

where g is the output layer that generates the output token.

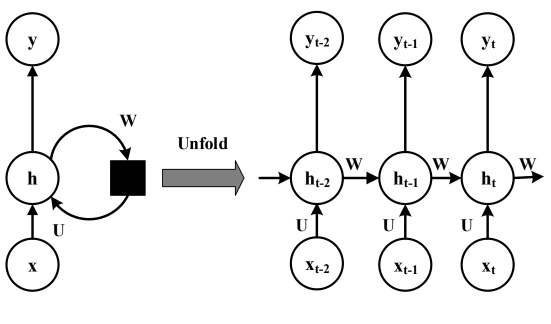


Figure 1: Standard RNN and its unfolded

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) [17, 18]. 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. Figure 2 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.

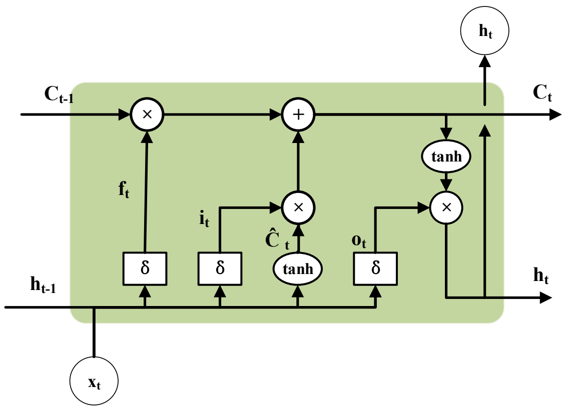


Figure 2: The structure of an LSTM unit

In Figure 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. In particular, σ can be computed by the following formula:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4}$$

f_t is generated by the forget gate, which decides what kind of information should be discarded or retained. The calculation formula of f_t is defined as follows:

$$f_t = \sigma(W^f x_t + U^f h_{t-1} + b^f) \tag{5}$$

where W^f, U^f and b^f are the parameters to be leaned that will be updated while training.

i_t is generated by the input gate, and it decides what kind of new information should be stored in the cell state. In particular, i_t can be computed as follows:

$$i_t = \sigma(W^i x_t + U^i h_{t-1} + b^i) \tag{6}$$

where W^i, U^i and b^i are the parameters to be learned that will be updated while training.

$\widehat{C}$ is the new information added in the current state. In particular, $\widehat{C_t}$ can be computed as follows:

$$\widehat{C_t} = tanh(W^c x_t + U^c h_{t-1} + b^c) \tag{7}$$

where W^c, U^c and b^c are the parameters, which will be updated while training.

o_t is generated by the output gate, which decides what kind of value should be set to the output. In particular, i_t can be computed as follows:

$$o_t = \sigma(W^o x_t + U^o h_{t-1} + b^o) \tag{8}$$

where W^o, U^o and b^o are the trainable parameters, which are updated while training.

C_t is the content vector of the current state. In particular, C_t can be computed as follows:

$$C_t = C_{t-1} * f_t + \widehat{C_t} * i_t \tag{9}$$

h_t is the hidden state of the previous state. h_t can be computed as follows:

$$h_t = C_t * o_t \tag{10}$$

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 [19] and computer vision [20]. Recent research shows that the CNN model is also effective in Natural language processing (NLP), and can perform very well in sentence classification [21] and web search [22] 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 [21, 23, 24] have been proposed, and different techniques have advantage of handling different problems. Due to space limitation,

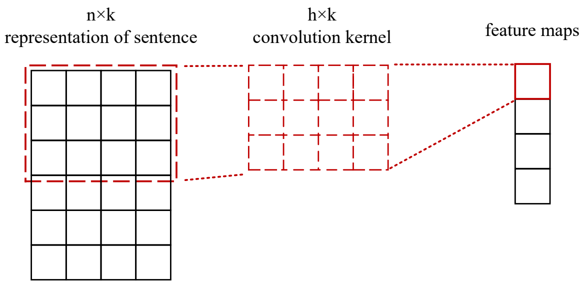


Figure 3: Convolutional layer of TextCNN

we use TextCNN as an example to introduce the basic idea of CNN. TextCNN [21] is a type of CNN that is designed to handle text related problems. Figure 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 the i-th word in the sentence. This convolution can be computed by the following formula:

$$c_i = f(W \cdot \boldsymbol{x_{i:i+h-1}} + b) \tag{11}$$

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 can not 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 [7] [25] 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 [10] [26] have proved CNN's capability of extracting lexical and syntactic information of source code. The reason of using LSTM is that previous studies [27] 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 [28, 29], 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.

3.1. *Framework of SeCNN*

Different from text data of natural languages, source code is a kind of structured data and the corresponding vocabulary is very large [2]. 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.

Figure 4 shows the framework of SeCNN. In Figure 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 word problem, we use camel casing conversion to split the identifiers from both code tokens and AST nodes into several words. In the second step, we use two CNNs to extract semantic information from 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 [2] [25] to convert source code into vectors. 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 $X = < x_1^{(code)}, \cdots, x_n^{(code)} >$, and we use word embedding to convert it into the vectors $< \boldsymbol{x_1}^{(code)}, \cdots, \boldsymbol{x_n}^{(code)} >$, where $\boldsymbol{x_i}^{(code)}$ is a k-dimensional vector indicating the i-th code token x_i.

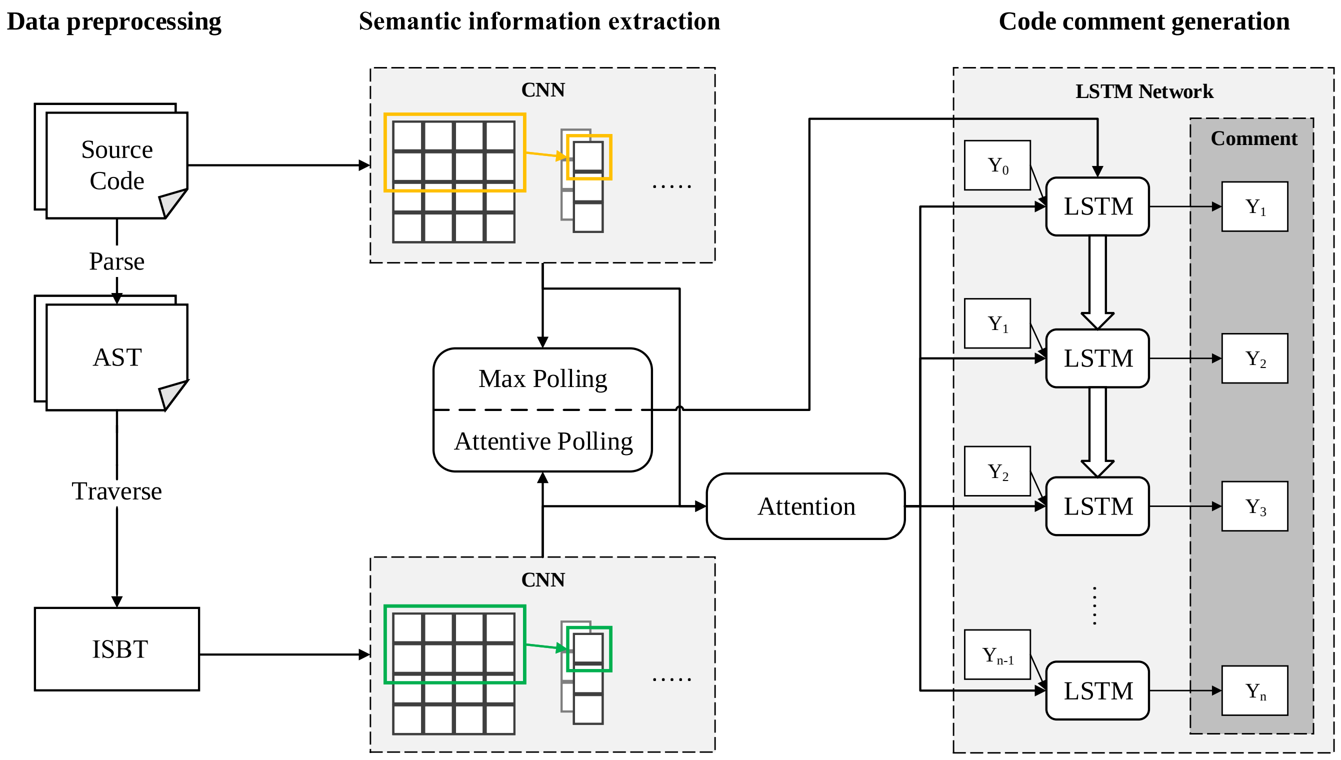


Figure 4: The framework of our proposed approach SeCNN

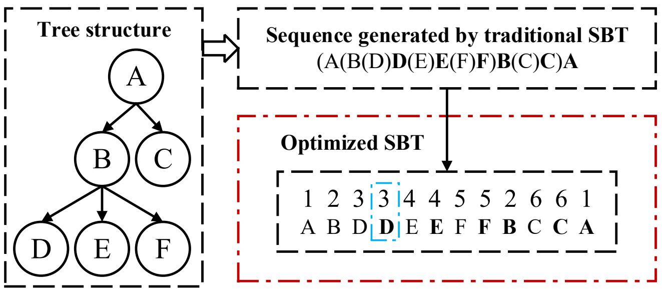

Tree structure
A
B
C
D
E
F
Sequence generated by traditional SBT
(A(B(D)**D**(E)**E**(F)**F**)**B**(C)**C**)**A**
Optimized SBT
1 2 3 3 4 4 5 5 2 6 6 1
A B D **D** E **E** F **F** **B** C **C** **A**

Figure 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)

3.2.2. Abstract Syntax Tree with Improved Structure-Based Traversal 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 [2], 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. Figure 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 the Figure 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 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,

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

for a serial number sequence $X = < x_1^{(\text{serial})}, \cdots, x_m^{(\text{serial})} >$, and a node sequence $\text{X} = < x_1^{(\text{node})}, \cdots, x_m^{(\text{node})} >$, we use word embedding to convert them into vectors $< \boldsymbol{x}_1^{(\text{serial})}, \cdots, \boldsymbol{x}_m^{(\text{serial})} >$ and $< \boldsymbol{x}_1^{(\text{node})}, \cdots, \boldsymbol{x}_m^{(\text{serial})} >$, 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:

$$\boldsymbol{x}_i^{(\text{isbt})} = \text{ReLU}(W^{(\text{isbt})}[\boldsymbol{x}_i^{(\text{node})}; \boldsymbol{x}_i^{(\text{serial})}]) \tag{12}$$

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.

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 $< \boldsymbol{x}_1^{(\text{code})}, \cdots, \boldsymbol{x}_n^{(\text{code})} >$, 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 $< \boldsymbol{y}_1^{(\text{code})}, \cdots, \boldsymbol{y}_n^{(\text{code})} >$. Such convolution can be calculated as follows:

$$\boldsymbol{y}_i^{\text{code}} = f(W^{\text{code}} \cdot \boldsymbol{x}_{i:i+h-1}^{\text{code}}) \tag{13}$$

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 $\boldsymbol{x}_{i:i+h-1}^{\text{code}}$.

Notice that our convolution slightly differs from TextCNN [21]. 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 $< \boldsymbol{x}_1^{(\text{isbt})}, \cdots, \boldsymbol{x}_m^{(\text{isbt})} >$ of ISBT information, we use the same convolution operation to extract feature vectors $< \boldsymbol{y}_1^{(\text{isbt})}, \cdots, \boldsymbol{y}_m^{(\text{isbt})} >$.

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

3.3.2. Polling

SeCNN uses polling 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). To construct the content vector with semantic information, SeCNN will firstly apply MAX polling [30] on the features $< \boldsymbol{y}_1^{(\text{code})}, \cdots, \boldsymbol{y}_n^{(\text{code})} >$ extracted from the code. Then, SeCNN uses a fixed-size controlling vector code_polling to compute the attention weights for ISBT Encoder. After that, SeCNN uses features $< \boldsymbol{y}_1^{(\text{isbt})}, \cdots, \boldsymbol{y}_m^{(\text{isbt})} >$ extracted from ISBT to do attentive pooling, which aims to make c_state contain lexical and grammatical information. Similarly, the features $< \boldsymbol{y}_1^{(\text{isbt})}, \cdots, \boldsymbol{y}_m^{(\text{isbt})} >$ are used to do MAX polling to get a controlling vector isbt_polling. Then we apply attentive pooling on the isbt_polling and features $< \boldsymbol{y}_1^{(\text{code})}, \cdots, \boldsymbol{y}_n^{(\text{code})} >$ 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. [28] as the attention mechanism in SeCNN.

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

$$\boldsymbol{c}_i = \sum_{j=1}^{m+n} a_{ij} \boldsymbol{y}_j^{(sem)} \tag{14}$$

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:

$$e_{ij} = \boldsymbol{a}(\boldsymbol{h}_{i-1}, \boldsymbol{y}_j^{(sem)}) \tag{15}$$

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:

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{m} exp(e_{ik})} \tag{16}$$

3.4.2. Sequence Output

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

$$y_i = P(y_i|y_1, \cdots, y_{i-1}, x) = g(y_{i-1}, h_i, c_i) \tag{17}$$

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 Figure 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 6 as an example, LSTM receives a serials of prior words as the input [31] during the model training process. Figure 6 shows the predicted result word by word from the decoder component in the training mode, and Figure 7 shows decoder in the testing mode, which takes the previous predicted words as a part of input rather than labeled text.

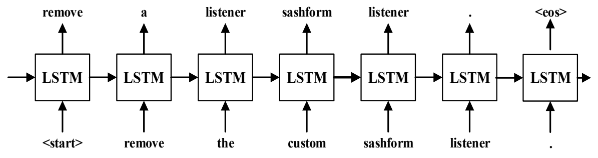


Figure 6: Decoder in the training mode

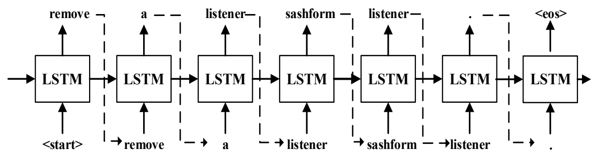


Figure 7: Decoder in the testing mode

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 serials of empirical studies and compare SeCNN with other state-of-the-art code comment generation baselines, including DeepCom [2], TL-CodeSum [11], Hybrid-DeepCom [25], Dual Model [16] and AST-attendgru [7].

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.

4.2. Dataset

In the Previous study, Hu et al. [11] collected a large corpus by matching Java methods and comments in 9,732 projects from GitHub [3]. 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. Table 1 and Table 2 show the statistics information of the dataset used in our study.

Table 1: Statistics for code snippets in the dataset

# Projects	# Files	# Lines	# Items
9,732	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%

Table 3: Splits of dataset

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

[3]https://github.com/

As shown in Table 1 and Table 2, the dataset used in our study contains a total of 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. Consistent with previous research [11], we split the dataset into the training set, test set, and validation set by 8:1:1, and the detailed split information can be found in Table 3.

4.3. Performance Metrics

To evaluate the effectiveness of SeCNN, we use two types of metrics: IR-based Metrics (*Precision*, *Recall*, F-sore) and MT-based Metrics (*BLEU*, *METEOR*). These chosen evaluation metrics have been widely used in previous code comment generation studies [11, 25, 16].

4.3.1. IR-based Metrics

***Precision*.** *Precision* is the proportion of the matched n-grams out of the total number of n-grams in the evaluated translation [32].
***Recall*.** *Recall* is the proportion of the matched n-grams out of the total number of n-grams in the reference translation [32].
***F-score*.** *F-score* is a summary measure that combines both *Precision* and *Recall*, and it evaluates if an increase in *Precision* (or *Recall*) outweighs a reduction in *Recall* (or *Precision*) [25].

There is a positive correlation between the above three IR-based metrics and the performance of code comment generation techniques, which means a higher *Precision*, *Recall*, and *F-score* value indicates a better code comment generation technique.

4.3.2. MT-based Metrics

***BLEU*.** *BLEU* [12] 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:

$$BLEU = \mathrm{BP} \cdot (\sum_{i=0}^{n} w_n log p_n) \tag{18}$$

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:

$$\mathrm{BP} = \begin{cases} 1 & c > r \\ e^{1-r/c} & c \leq r \end{cases} \tag{19}$$

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 [2, 25, 16].

***METEOR*.** *METEOR* [13] is explicitly designed to improve correlation with human judgments of machine translation quality at the segment level. The value can be computed as follows:

$$METEOR = (1 - Pen) \cdot F_{mean} \tag{20}$$

where Pen is a penalty coefficient, and F_{mean} is a parameterized harmonic mean.

Pen can be computed as follows:

$$Pen = \gamma(\frac{ch}{m})^{\beta} \tag{21}$$

where ch is the number of chunks. 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.

F_{mean} can be computed as:

$$F_{mean} = \frac{P \cdot R}{\alpha \cdot P + (1 - \alpha) \cdot R} \tag{22}$$

where P is the unigram *Precision* and R is the unigram *Recall*.

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. 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 [2] 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 [2].
Baseline 2: TL-CodeSum [11] 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 [11] to perform comparison.
Baseline 3: Hybrid-DeepCom [25] 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 [25] to perform comparison.
Baseline 4: AST-attendgru [7] 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. [2]. In our

study, we re-implement AST-attendgru model according to the same parameter settings used in the corresponding study [7] to perform comparison.

Baseline 5: Dual Model [16] 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 [16] to perform comparison.

4.5. Experimental Settings

We replace the constant numbers and strings in the source code with special marks <unm> and <str> respectively. 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. 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. 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.

• 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.

• The decoder model of SeCNN uses one-layer LSTM, where the hidden state is 500 dimension, and the embedded word is 500 dimension.

• 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.

• SeCNN clips the gradients norm by 5. We use dropout strategy during the training process and set dropout to 0.8.

• SeCNN uses cross-entropy minimization as the cost function.

We use python 3.6 and Tensorflow [4] framework to implement SeCNN. All experiments conducted in our study run on a Linux Server (Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz CPU and a Tesla V100 GPU with 32 GB memory).

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

5. Result Analysis

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

As introduced in Section 4.4, 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).

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**

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.

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.

Table 5: *Precision*, *Recall*, and *F-score* of different approaches

Approaches	*Precision*(%)	*Recall*(%)	*F-score*(%)
DeepCom	46.82	46.52	45.72
Hybrid-DeepCom	54.37	53.20	52.45
AST-attendgru	54.19	**54.71**	53.08
SeCNN	**56.31**	54.38	**54.18**

Besides, we also compare the performance of SeCNN and other baselines in terms of three IR-based metrics, *Precision* and *Recall*, *F-score*. Table 5 shows the experimental results, where only three baselines, DeepCom, Hybrid-DeepCom and AST-attendgru are listed. We have not re-implemented TL-CodeSum and Dual Model, but directly use the results in the corresponding paper which do not provide IR-based metrics related results. In Table 5, it can be seen that SeCNN performs the best in terms of *Precision* and *F-score* metrics, but slightly lower than AST-attendgru in terms of *Recall* metric. Take a deep analyse, AST-attendgru usually generates longer comments, so it shows advantage in terms of *Recall* metric. On the other hand, AST-attendgru generates unnecessary comments for shorter comments, which causes the *Precision* of AST-attendgru is lower than SeCNN and Hybrid-DeepCom. SeCNN learns the semantic information of the source code, and the results listed in Table 5 demonstrate that semantic information can be helpful for generating more comprehensive and accurate comments.

Summary for RQ1: Experimental results show that SeCNN achieves better performance than five state-of-the-art baselines. In terms of MT-based metrics, SeCNN can achieve the best performance in terms of *BLEU* and *METEOR*. In terms of IR-based metrics, SeCNN can achieve the best performance in terms of *Precision* and *F-score*, and slightly lower than AST-attendgru in terms of *Recall*.

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. Therefore, due to space limitation, 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 Figure 8.

Figure 8(a) and Figure 8(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:

$$F(x) = \begin{cases} 300 & x \geq 300 \\ (\lfloor x/10 \rfloor + 1) * 10 & x < 300 \end{cases} \tag{23}$$

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

From Figure 8(a) and Figure 8(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. Since short code is often incomplete, so such performance is reasonable.

Figure 8(b) and Figure 8(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. 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.

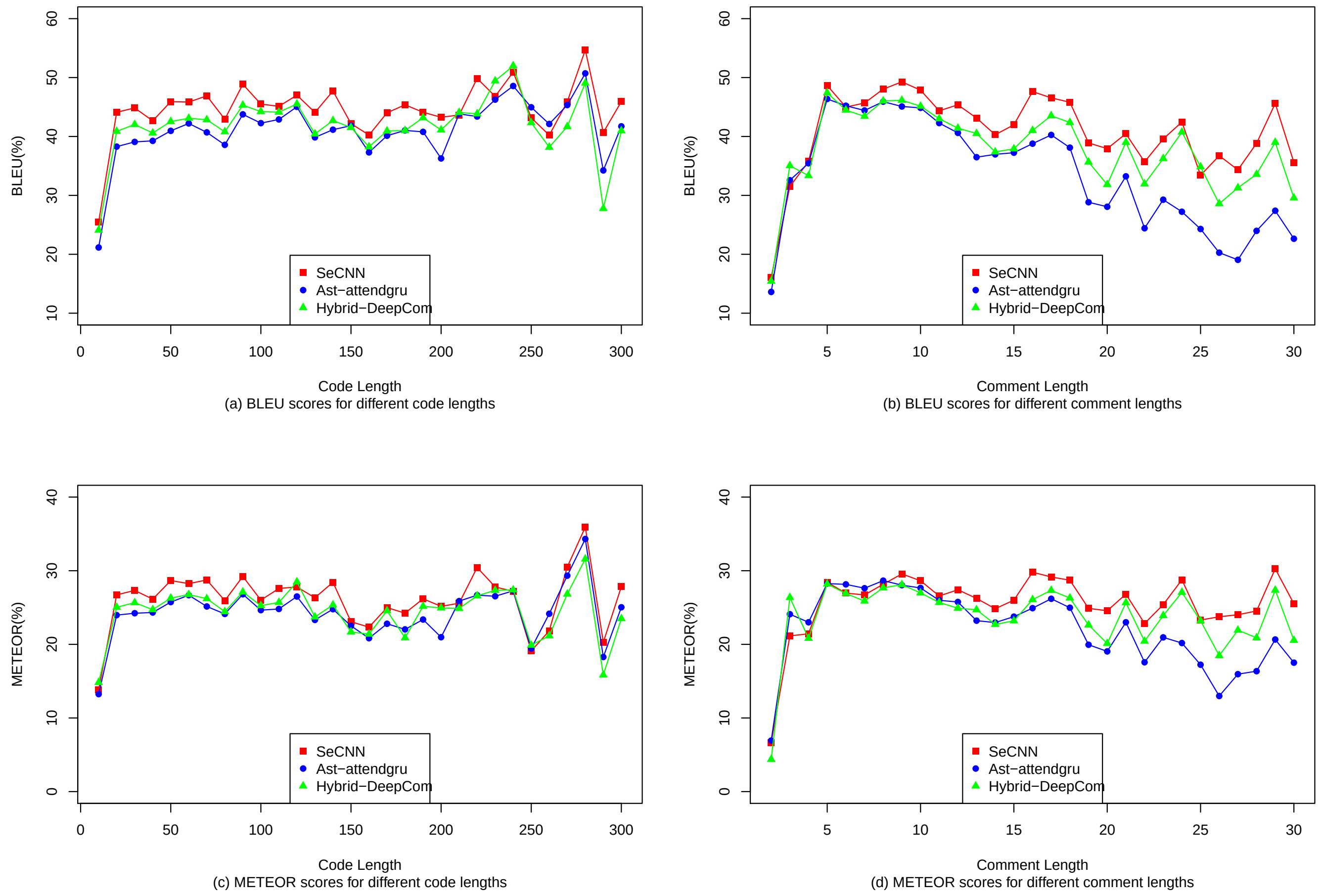
Summary for RQ2: When considering different lengths of code and comments, SeCNN achieves better performance than Hybrid-DeepCom and AST-attendgru in most cases. Besides, the experiment results also show that SeCNN performs the worst when code or comments are too short, and the reason is that short source 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. 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, due to space limitation, we only employ Hybrid-DeepCom as one baseline of code comment generation technique to perform comparison.

Table 6 lists four example cases to perform comparison. In each case, Table 6 shows the source code of one Java method and the corresponding comments written by human and generated by two techniques, SeCNN and Hybrid-DeepCom. To further evaluate the effectiveness of code comment generation techniques, we also ask two authors to manually analyze 50 randomly selected examples independently. For ambiguous examples, the final classification results are determined through the discussion of these two authors. To measure the consistency of results, we employ a commonly-used kappa [33] method to calculate a coefficient score. The value of kappa score is changed from 0 to 1, and a higher kappa score indicates a better agreement of two authors. In Table 7, it can be seen that the kappa score of SeCNN and Hybrid-DeepCom are 0.94 and 0.91 respectively, which indicates that the two authors have a high agreement of classification on comments generated by these two techniques.

As shown in Table 6 and Table 7, we analyze these examples and achieve the following findings.



(a) BLEU scores for different code lengths
(b) BLEU scores for different comment lengths
(c) METEOR scores for different code lengths
(d) METEOR scores for different comment lengths

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

Correct comments: Correct comments are the comments generated by automatic techniques are almost the same as Human-written comments, such as case 1 listed in Table 6. In Table 7, it can be seen that SeCNN can generate more correct comments than Hybrid-DeepCom. In specifically, SeCNN can generate 40% correct comments, while Hybrid-DeepCom only generated 32% correct comments.
More detailed comments: More detailed comments means that the auto generated comments have better quality than human-written comments, as case 2 listed in Table 6. Due to tight project schedule, there are many incomplete comments in projects. So generating more detailed comments can be helpful to fix this problem. Table 7 shows that both SeCNN and Hybrid-DeepCom can generate some more detailed comments, and SeCNN performs slightly better than Hybrid-DeepCom.
Similar comments: Similar comments mean that the generated comments have the same meaning as the human-written comments, like case 3 shown in Table 6. In Table 7, it can be seen that SeCNN and Hybrid-DeepCom have similar performance on generating similar comments.
Wrong comments: Wrong comments mean the generated comments cannot explain the meaning of the source code, and may misleas the developer's understanding, like case 4 shown in Table 6. In Table 7, it can be seen that although SeCNN performs better than Hybrid-DeepCom, both of these two technique will generate a number of wrong comments, which indicates that we need to conduct more research to further improve the performance of code comment generation techniques.

Summary for RQ3: We divide the comments generated by SeCNN and Hybrid-DeepCom into four categories: correct comments, more detailed comments, similar comments, and wrong comments, and conduct a user study to analyze the performance. The results show that SeCNN can generate more high-quality comments and less wrong comments than Hybrid-DeepCom.

5.4. Discussions

5.4.1. Discussion on out-of-vocabulary word problem

Due to a large number of identifiers defined in the source code, the out-of-vocabulary word problem is one of the challenges of neural network based code comment generation techniques [34]. In natural language processing area, an effective method to handle out-of-vocabulary word problem is limiting

Table 6: Examples of code comment generated by SeCNN

Case ID	Examples
1	`public OMScalingRaster(double ullat, double ullon, double lrlat, double lrlon, ImageIcon ii){ this(ullat, ullon, lrlat, lrlon, ii.getImage()); }` **Human-written:** create an omraster , lat / lon placement with an imageicon . **SeCNN:** create an omraster , lat / lon placement with an imageicon . **Hybrid-DeepCom:** constructs an omraster , with placement size .
2	`public java.lang.StringBuffer insert (int offset, java.lang.String str){ internal.insert(offset, str); return this; }` **Human-written:** inserts the string into this string buffer . **SeCNN:** inserts the string representation of the char argument into this string buffer . **Hybrid-DeepCom:** inserts the string representation of the boolean argument into this string buffer .
3	`public static File toFile(JavaFileObject javaFileObject){ return new File(javaFileObject.getName()); }` **Human-written:** gets the file from a java file object . **SeCNN:** create a file from a java object **Hybrid-DeepCom:** create a file from the source file descriptors
4	`private static int exitWithStatus(int status) { if (ToolIO.getMode() == ToolIO.SYSTEM){ System.exit(status); } return status ; }` **Human-written:** if run in the system mode , exits the program , in tool mode returns the status **SeCNN:** the method exists , which means the status is null and remove status . **Hybrid-DeepCom:** returns a hash failure .

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 word 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.

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-DeepCom 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 8 shows the training time of one epoch of these four techniques.

As shown in Table 8, SeCNN with ISBT has the least training time. Compared with SeCNN (original SBT), we find that

Table 7: Manual analysis on 50 examples

Type	SeCNN		Hybrid-DeepCom	
	The first author	The second author	The first author	The second author
Correct comments	20(40%)	20(40%)	16(32%)	16(32%)
More detailed comments	2(4%)	1(2%)	1(2%)	1(2%)
Similar comments	14(28%)	14(28%)	15(30%)	12(24%)
Wrong comments	14(28%)	15(30%)	18(36%)	21(42%)
kappa score	0.94		0.91	

Table 8: Comparison of the running time of an epoch

Approaches	Time (s)
Hybrid-DeepCom	1206.25
AST-attendgru	4107.82
SeCNN(Original SBT)	1155.34
SeCNN(ISBT)	**873.70**

using ISBT can reduce the training time of SeCNN by 24.38%, which proves that using ISBT can obviously improve the efficiency of SeCNN.

Compared with Hybrid-DeepCom, the running time of SeCNN (original SBT) is slightly reduced. 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 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.

6. Threats to Validity

In this section, we discuss the potential threats to our study.
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.
External Validity. Threats to external validity relates 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 code comment generation work. However, explaining a piece of code often requires a lot of comments. The dataset we used was collected by Hu et al. [11] on GitHub. Although Hu et al. [11] uses heuristic rules to reduce noise in comments. We believe that there are still some code and comments that do not match in the dataset.
Construct Validity. The construct validity relates to metrics. To reduce the impact of evaluation measures, we used two machine translation metrics that are *BLEU* and *METEOR*, And three IR-based Metrics that are *Precision* and *Recall* and *F-score*. *BLEU* and *METEOR* have been widely used in the machine translation tasks, and *Precision* and *Recall* and *F-score* have been widely used in software engineering tasks [25].

7. Related Work

7.1. Deep Code Representation

Deep learning algorithms are commonly used in the field of image processing [20] and natural language processing [35]. 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 [36].

Many researchers have proposed a set of approaches to do deep code representation, which include attention unit [9], sequence-to-sequence [37], and CNN [38], etc. For example, Iyer et al. [9] used attention unit to calculate the distribution representation of code segment and then employed neural networks to generate code comments. Gu et al. [37] 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. [39] 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. [38] 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 [40], code summarization [9], code clone detection [41, 42], program repair [43], and code generation [44].

Hindle et al. [45] 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. [46] 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. [37] 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 et al. [47] 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 [48, 49, 50, 51, 52, 15]. 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 [3, 4, 5, 53, 54] and AI-based code summarization [9, 55, 2, 11, 25, 16, 56].

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. [3, 4] attempted to generate code comments by calculating the top-n keywords with metrics such as TF/IDF. Rodeghero et al. [5] further improved the content extraction of heuristics and templating solutions by modifying heuristics to mimic how human developers read code with their minds. McBurney et al. [53] proposed a software word usage model (SWUM) to generate code comments [54].

AI-based code comment generation approaches use neural network algorithms and machine translation models. Iyer et al. [9] 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. [55] employed deep reinforcement learning framework to handle code representation and exposured bias problems during the code summarization process. Hu et al. [2] 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. [11] 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. [25] 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. [57] 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. [58] 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 Wei et al. [56]. 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. 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.

Acknowledgement

The work described in this paper is supported by the National Natural Science Foundation of China (Grant nos. 61902015, 61872026 and 61672085) and the Nantong Application Research Plan (Grant no. JC2019106).

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li, Measuring program comprehension: A large-scale field study with professionals, IEEE Transactions on Software Engineering 44 (10) (2017) 951–976.

[2] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 200–210. doi:10.1145/3196321.3196334.
URL https://doi.org/10.1145/3196321.3196334

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

[4] S. Haiduc, J. Aponte, A. Marcus, Supporting program comprehension with source code summarization, Proceedings - International Conference on Software Engineering (2010).

[5] P. Rodeghero, C. Liu, P. W. McBurney, C. McMillan, An eye-tracking study of java programmers and application to source code summarization, IEEE Transactions on Software Engineering 41 (11) (2015) 1038–1054.

[6] L. Moreno, A. Marcus, L. L. Pollock, K. Vijay-Shanker, 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, 2013, pp. 230–232. doi:10.1109/ICPC.2013.6613855.
URL https://doi.org/10.1109/ICPC.2013.6613855

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

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

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

[10] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, L. Zhang, 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, 2019, pp. 7055–7062. doi:10.1609/aaai.v33i01.33017055.
URL https://doi.org/10.1609/aaai.v33i01.33017055

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

[12] K. Papineni, S. Roukos, T. Ward, W. Zhu, 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, 2002, pp. 311–318.
URL https://www.aclweb.org/anthology/P02-1040/

[13] S. Banerjee, A. Lavie, METEOR: an automatic metric for MT evaluation with improved correlation with human judgments, in: J. Goldstein, A. Lavie, C. Lin, C. R. Voss (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, 2005, pp. 65–72.
URL https://www.aclweb.org/anthology/W05-0909/

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

[15] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, T. Matsumura, Automatic source code summarization with extended tree-lstm, in: International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019, IEEE, 2019, pp. 1–8. doi:10.1109/IJCNN.2019.8851751.
URL https://doi.org/10.1109/IJCNN.2019.8851751

[16] B. Wei, G. Li, X. Xia, Z. Fu, Z. Jin, Code generation as a dual task of code summarization, in: H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, R. Garnett (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, 2019, pp. 6559–6569.
URL http://papers.nips.cc/paper/8883-code-generation-as-a-dual-task-of-code-summarization

[17] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780. doi:10.1162/neco.1997.9.8.1735.
URL https://doi.org/10.1162/neco.1997.9.8.1735

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

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, 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, 2015, pp. 1–9. doi:10.1109/CVPR.2015.7298594.
URL https://doi.org/10.1109/CVPR.2015.7298594

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

[21] Y. Kim, Convolutional neural networks for sentence classification, Eprint Arxiv (2014).

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

[23] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, 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, 2016, pp. 2818–2826. doi:10.1109/CVPR.2016.308.
URL https://doi.org/10.1109/CVPR.2016.308

[24] N. Kalchbrenner, E. Grefenstette, P. Blunsom, 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, 2014, pp. 655–665. doi:10.3115/v1/p14-1062.
URL https://doi.org/10.3115/v1/p14-1062

[25] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation with hybrid lexical and syntactical information, Empirical Software Engineering (2019) 1–39.

[26] M. Allamanis, H. Peng, C. A. Sutton, A convolutional attention network for extreme summarization of source code, in: M. Balcan, K. Q. Weinberger (Eds.), Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, Vol. 48 of JMLR Workshop and Conference Proceedings, JMLR.org, 2016, pp. 2091–2100.

URL http://proceedings.mlr.press/v48/allamanis16.html

[27] J. Gehring, M. Auli, D. Grangier, Y. N. Dauphin, A convolutional encoder model for neural machine translation, in: R. Barzilay, M. Kan (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, 2017, pp. 123–135. doi:10.18653/v1/P17-1012.
URL https://doi.org/10.18653/v1/P17-1012

[28] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, Computer Science 1409 (09 2014).

[29] T. Luong, H. Pham, C. D. Manning, Effective approaches to attention-based neural machine translation, in: L. Màrquez, C. Callison-Burch, J. Su, D. Pighin, Y. Marton (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, 2015, pp. 1412–1421. doi:10.18653/v1/d15-1166.
URL https://doi.org/10.18653/v1/d15-1166

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

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

[32] W. Zheng, H. Zhou, M. Li, J. Wu, Codeattention: translating source code to comments by exploiting the code constructs, Frontiers Comput. Sci. 13 (3) (2019) 565–578. doi:10.1007/s11704-018-7457-6.
URL https://doi.org/10.1007/s11704-018-7457-6

[33] A. J. Viera, J. M. Garrett, Understanding interobserver agreement: The kappa statistic, Family Medicine 37 (5) (2005) 360–363.

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

[35] L. Bao, P. Lambert, T. Badia, Attention and lexicon regularized LSTM for aspect-based sentiment analysis, in: F. E. Alva-Manchego, E. Choi, D. Khashabi (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, 2019, pp. 253–259. doi:10.18653/v1/p19-2035.
URL https://doi.org/10.18653/v1/p19-2035

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

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

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

[39] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, Computer Science (2015).

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

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

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

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

[44] M. Rabinovich, M. Stern, D. Klein, Abstract syntax networks for code generation and semantic parsing, in: R. Barzilay, M. Kan (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, 2017, pp. 1139–1149. doi:10.18653/v1/P17-1105.
URL https://doi.org/10.18653/v1/P17-1105

[45] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, Proceedings - International Conference on Software Engineering (2012) 837–847.

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

[47] P. Yin, G. Neubig, A syntactic neural model for general-purpose code generation, in: R. Barzilay, M. Kan (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, 2017, pp. 440–450. doi:10.18653/v1/P17-1041.
URL https://doi.org/10.18653/v1/P17-1041

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

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

[50] A. LeClair, S. Haque, L. Wu, C. McMillan, Improved code summarization via a graph neural network, arXiv preprint arXiv:2004.02843 (2020).

[51] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, C. Sutton, Autofolding for source code summarization, IEEE Transactions on Software Engineering 43 (12) (2017) 1095–1109.

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

[53] P. W. Mcburney, C. Mcmillan, Automatic source code summarization of context for java methods, IEEE Transactions on Software Engineering 42 (2) (2016) 103–119.

[54] E. Hill, L. L. Pollock, K. Vijay-Shanker, 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, 2009, pp. 232–242.

doi:10.1109/ICSE.2009.5070524.
URL https://doi.org/10.1109/ICSE.2009.5070524

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

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

[57] B. Liu, T. Wang, X. Zhang, Q. Fan, G. Yin, J. Deng, 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, 2019, pp. 1–10.

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