Automatic Generation of Source Code Comments Using Convolutional Neural Networks

Y. R. Zeng[1], C. Y. Huang[1]
[1]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

Abstract - **Source code is the key factor for understanding program behavior, and comments enable developers to maintain programs more easily. Producing comments is labor-consuming for developers; hence, automated tools for generating comments can considerably reduce the effort of developers. Recently, several studies have developed deep learning based tools to produce code comments. However, most of them face the out-of-vocabulary problem caused by diverse identifiers and the vanishing gradient problem caused by recurrent neural networks.**

To resolve the OOV and vanishing gradient problem, we proposed another deep learning based approach, namely ComCNN, to generate the comments for Java code. Convolutional neural networks and bidirectional long short-term memory networks were employed as our proposed model architecture. To achieve better semantic learning of identifiers and to resolve the OOV problem, the identifier splitting method was used with ComCNN to preprocess the training data. Eventually, ComCNN achieved 3%–7% higher prediction scores and was more lightweight when compared to comparative models.

Keywords – **Deep Learning, Comment Generation, Program Comprehension, Machine Translation, Software Engineering**

I. INTRODUCTION

Comments are the key point of understanding. Good comments can assist program maintainers to quickly understand the logic of programs, thus reducing the time cost for maintainers. Conversely, if the source code lacks understandable comments, the program maintainers may be unable to understand the behavior of the programs. One explanation for the lack of understandable comments may be the insufficient time to write comments. Moreover, writing comments for source code is time-consuming for developers. According to a survey [1], 78.2% of software maintainers believed that comments were crucial for software development. Therefore, applying an automated tool to help developers produce high-quality comments for the source code would not only considerably reduce comprehension issues but also save developers' time.

For comment generation, manually-designed approaches [2][3][4] mostly depend on the handcrafted features of the source code. However, the designs of handcrafted features are usually labor-intensive and are only able to capture a portion of the source code's information. By contrast, deep learning approaches [5][6][7][8] automatically learn the most needed features for comment generation, thus predicting the comments more precisely. Therefore, our proposed approach, namely ComCNN, uses deep learning techniques for comment generation.

Before training the models, the previous deep learning approaches [5][6][7] faced a data-preprocessing problem, namely, the out-of-vocabulary (OOV) problem. The OOV problem results from the sparse identifier (ID) names in the training set. The rare ID names in the vocabulary would not be learned properly and lead to poor performance [9][10]. To fix the OOV problem, we designed an ID-splitting approach. In terms of the model architecture, most of previous studies [8][9] employed long short-term memory network (LSTM) architecture [11]. However, LSTM can only learn the unidirectional memories of inputs. Instead of using LSTM, we proposed a novel architecture based on convolutional neural network (CNN) [12] and bidirectional long short-term memory network (Bi-LSTM) [13][14].

The dataset we used consisted of 119,233 data units and was divided into the training set and the testing set, which occupied 90% and 10% of all the data, respectively. After the training stage, the experiments leveraged some metrics to evaluate the comments generated by the trained models. Our source code for the project and dataset are available on our GitHub page[1]. The main contributions of this paper are listed below: (i) We propose an ID-splitting method to resolve the OOV problem and achieve better semantic learning (ii) We constructed the model architecture with CNN and Bi-LSTM to resolve the vanishing gradient problem and to learn the bidirectional memories of source code (iii) The integrated model, namely ComCNN, uses the ID-splitting method, the beam search algorithm, and our proposed model architecture and displays superior comment prediction performance.

The rest of this paper is organized as follows. Section II introduces studies regarding comment generation and the current challenges in comment generation. Section III presents the details of our comment predicting flow and explains our proposed model architecture. Section IV first introduces the criteria evaluating the comment prediction and then presents the experimental settings and results. Finally, Chapter V concludes our research and provides our future works.

II. RELATED WORK

A. *Manually-designed Comment Generation*

In the past, manually-designed comment generation methods mostly utilized complicated features and processes to predict comments. Some manually-designed methods are listed in this section. An open source tool,

[1] https://github.com/yurong0404/ComCNN

namely SmartComments [2], allows programmers to create comments from JavaScript source code. It creates the descriptions containing three pieces of information, such as method name, method parameters, and return type. However, it is not able to create the comments describing the behavior of the method.

Wong et al. [3] used NLP techniques to extract the code-description pairs from the Stack Overflow website and built a code-description database. Then, they leveraged code clone detection techniques to map the target source code to several similar source code in the database and further selected the best comment among the available comments of the selected source code as the predicted comment.

McBurney et al. [4] also proposed an approach to generate the summaries of the given code, and the summaries aimed to provide several pieces of information, including what the methods do internally, why the methods exist, and how to use the methods. However, according to their questionnaire, the programmers claimed that the generated summaries detailed inner information but provided no big picture summary.

Lastly, Iyer et al. [5] used an information retrieval (IR) approach as a representative of non-deep learning based models in a model comparison. The IR approach employs the Levenshtein distance algorithm [15] to find the code most similar to the input code among the training data and returns its corresponding comment as a predicted comment.

The studies outlined the category of manually designed methods. The manually designed methods mostly utilize complicated features and processes to predict comments. They are labor intensive and are only able to extract a part of the features, which may lead to erroneous and inaccurate predictions [5].

B. Deep Learning Based Comment Generation

Some studies [5][6][7] have demonstrated that deep learning based approaches outperform the traditional approaches. For example, Iyer et al. [5] proposed a deep learning based approach to generate comments for source code, namely, CODE-NN, and it summarized the C# and SQL codes. CODE-NN employs LSTM architecture to train the model and uses the dataset from Stack Overflow.

Zhang et al. [16] proposed an approach to comment generation that combined the advantages of retrieval-based methods and deep learning based methods. Because the low-frequency tokens are disadvantaged in the deep learning based approaches, the retrieval-based method makes use of low-frequency tokens. The retrieval-based method finds the two most similar codes from the training data based on syntax and semantics, and it further passes these two pieces of code into the deep learning model. The results revealed that combining the retrieval-based method and the deep learning based method could improve comment generation tasks.

Hu et al. [6] proposed an approach to generate comments for Java code, namely DeepCom. DeepCom was implemented with LSTM architecture and their proposed abstract syntax tree (AST) traversal method: structure-based traversal (SBT). Following the DeepCom study [6], Hu et al. further enhanced DeepCom and proposed Hybrid-DeepCom [7], which leverages not only syntactical information but also lexical information to train the model. To pursue better learning of the lexical information from source code, Hu et al. split the camel case IDs in the preprocessing stage. Compared to DeepCom, Hybrid-DeepCom has larger model architecture and better comment generation performance but leads to slower training processes and much heavier training loads.

According to our survey of the comment generation tasks above, we addressed several issues for further improvement. Firstly, the deep learning models may encounter the OOV problem. Before the source code is passed into neural networks, each token should be transformed into a number, and the vocabulary records the corresponding numbers of each token. However, there are numerous unique tokens among source codes, and many IDs appear only a few times in the dataset. To avoid poor performance caused by rare IDs, rare IDs outside the vocabulary need to be replaced with <unk> tags [9][10]. However, if the OOV problem cannot be properly handled, such <unk> tags will occupy a large proportion of tokens and still lead to poor performance.

The previous studies on comment generations [5][6] aborted the semantic information of IDs to deal with the OOV problem. Hu et al. [7] exploited the ID-splitting method to split camel case IDs to resolve the OOV problem. In this study, we also exploited the ID-splitting method to deal with the OOV problem. However, in contrast to the Hybrid-DeepCom [7], our proposed ID-splitting method not only splits the camel case IDs but also the snake case IDs, preserving the semantic information of both the camel case and snake case IDs.

In terms of the prediction algorithm, some studies [5][6] have exploited the greedy search approach to generate comments in the prediction stage. The greedy search generates the word with the highest possibility at every timestep to predict a sentence of comment. However, if the greedy search always predicts the word with the highest probability at every timestep, the predicted sentence may not turn out to be the best one. Sometimes, selecting the words with the second or third highest probability would improve the comment predictions. Hence, to resolve this issue, we used an alternative algorithm, beam search [17], to predict comments.

Another challenge of the deep learning based approach is the vanishing gradient problem [18], which is often caused by long input sequences in recurrent neural networks. The inputs of comment generation tasks are long and usually include hundreds of tokens. Most of previous approaches used LSTM to learn the features of source code [5][6] and led to poor performances when the input snippet was long. To solve the vanishing gradient problem, we propose a novel model architecture based on CNN and Bi-LSTM in this study, and it significantly relieves the vanishing gradient problem.

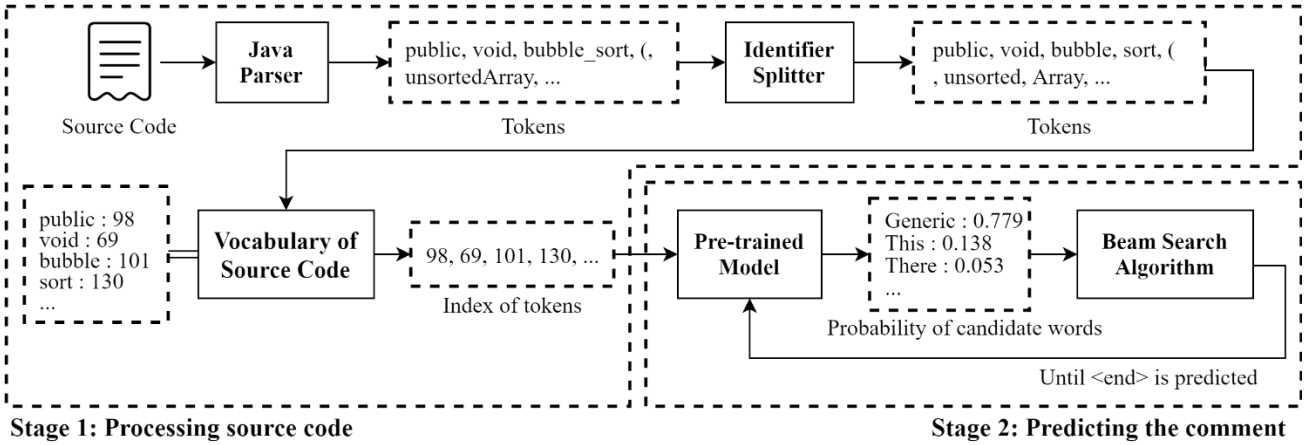

Fig. 1. The comment predicting flow of our proposed approach.

Lastly, the previous studies on comment generation all focused on the short source code [5][6][7]; however, functions may be longer in software development. In this study, we research the impact of using long source code on comment generation; therefore, the longer source codes are included in our dataset.

III. METHODOLOGY

A. *Working Flow of Comment Prediction*

To generate the comments of the given source code, we formulated a complete working flow of comment prediction in Fig. 1. The working flow includes the Java parser stage, the ID-splitting stage, the token-to-indices stage, and the beam search stage. In the beginning, the input source code is parsed by the Java parser, which applies the open source tool, Javalang [19]. Javalang provides a lexer and parser targeting Java. In the Java parser stage, we adopted tokenization mechanism [20] to split the snippet into tokens, including keywords, IDs, separators, and operators.

TABLE I:
STATISTICS FOR FREQUENCIES OF IDS IN OUR DATASET

# of all IDs	# of camel case	# of snake case
2,395,062	1,289,051	115,124

After the parsing stage, the source code turns out a bunch of tokens and then is passed to the ID-splitting stage. In the source code, IDs occupy a big proportion and are the key factor causing the OOV problem. Furthermore, many user-defined tokens are compound words following the naming conventions such as camel case and snake case [21][22]. TABLE I shows that more than half of the IDs in the training set follow the camel case and the snake case. Many of the compound word IDs are rare tokens leading to large vocabulary and causing the sparse ID problem. By contrast, the IDs with single words tend to be common.

Example 1: Camel case refers to the naming convention in which each word in the middle of a phrase begins with a capital letter, such as, “decToHex,” “getCurrentTime.” The token “getCurrentTime” is rare in the training set, but it is composed of three common tokens, such as “get,” “current,” and “time”.

Example 2: Snake case is the naming convention where the words are combined with an underscore, (e.g., “set_root_path” and “exec_command”). The rare token “set_root_path” consists of three common tokens, such as “set”, “root”, and “path.”

The embedding vectors of rare tokens would not precisely represent the semantics because they are too rare to train model. By contrast, the trained model can recognize the one-word IDs well because they are common during training processes. Hence, it is essential to split the IDs into common tokens, and the ID-splitting method splits them, particularly with the camel case and the snake case. After the ID-splitting stage, the tokens need to be transformed into vocabulary indices. The vocabulary of the source code records the corresponding index number of each token and so does the vocabulary of the comments. In term of prediction algorithm, ComCNN employed beam search algorithm [17] and the pseudo code of beam search is illustrated in Algorithm 2.

Algorithm 2 Beam search for comment generation

```
Input:
  code: the input source code
  model: the pre-trained model to generate comments
  B: determines the number of selected candidates
  maxLen: the maximum length of predicted comment
Output:
  The predicted comment for the input snippet
Function beam_search (code, model, B, maxLen)
  Res = {“<start>”}, iter = 0
  while (iter<maxLen) and !(all elements in Res end with“<end>”s)
    iter += 1
    Cands = {}  // stand for Candidates
    for r in Res
      if (r ends with a tag “<end>”)
        Cands = Cands ∪ {r}
      else
        topB_words = Predict_Top_B_Words(model, B, r)
        Cands = Cands ∪ {concat(r, w) | w ∈ topB_words}
    Res = Select_Top_B_Candidates(Candidates, B)
  return Top_One_Candidate(Res)
```

B. Model Architecture

The sequence-to-sequence (seq2seq) framework [23] is common and was developed by Google. It can translate one kind of sequence to another kind of sequence and is applicable for numerous different languages. The seq2seq model has dramatically improved several deep learning tasks [24][25]. Due to the powerful capability of seq2seq framework, we also based our proposed model on the seq2seq framework.

In Fig. 2, our proposed model architecture is primarily divided into two parts, an encoder and a decoder. The encoder is used to extract the features of input snippets, and the decoder is used to predict the comments of the snippets based on the information from the encoder. The encoder is composed of an embedding layer, a CNN layer, a max pooling layer, and a Bi-LSTM layer. The embedding layer converts the source code tokens into the vectors representing the semantic information. The CNN layer is used to extract the features of the tokens, and the max pooling layer further selects the significant features. The Bi-LSTM layer uses bidirectional memories to learn the full behavior of the input source code.

The decoder is composed of an embedding layer, an LSTM layer, an attention layer, and a full-connected layer. The embedding layer is used to convert comment tokens into the vectors representing their semantic information. The LSTM layer is used to keep the memories of the comment sentences, and the attention layer assigns weights to the output of the encoder to improve the prediction. Finally, the full-connected layer transforms the complicate vectors into the possibility distribution of candidate words.

The primary improvement of our proposed model architecture compared to other studies [5][6][7] is that we use the CNN layer and max pooling layer to extract the features of the code. The advantage of using the CNN layer and max pooling layer is that the number of code features is greatly reduced and the vanishing gradient problem is avoided. For the learning behavior of the code, we used Bi-LSTM. Compared with the pure LSTM used by other studies [5][6], Bi-LSTM can learn the behavior of code more effectively.

CNN has the strength to extract the features of input data and hence trains the model more effectively [26][27]. The second strength of CNN is that it can reduce the dimension of data [26][27]. In our approach, 1D CNN was used to fetch features of the source code and to reduce the data's dimension. In the encoder, the CNN parses a snippet into a sequence of features. In Fig. 3, the kernel maps, consist of the weights to perform convolutional computations with the input, which can learn the relationships between the neighboring tokens in the source code.

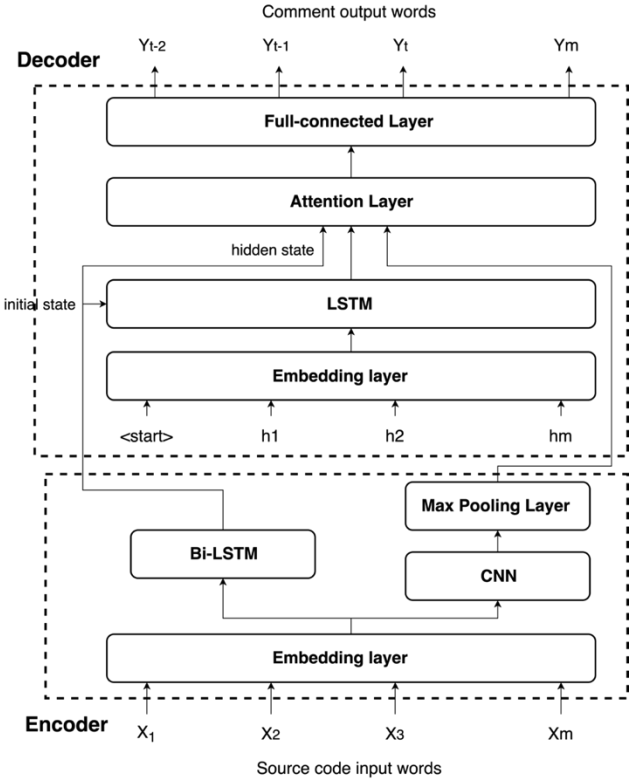

Fig. 2. The model architecture of our proposed approach.

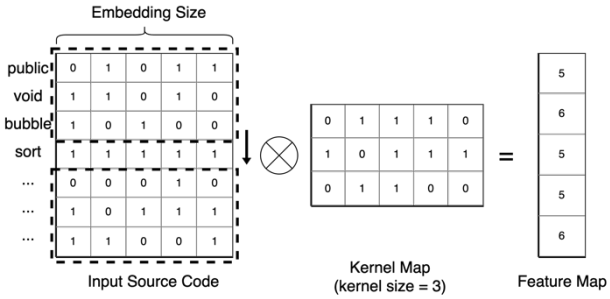

Fig. 3. The structure of 1D convolutional neural networks.

Pooling layers are used to extract important features and further shorten the length of output from CNN. There are two types of pooling layers, namely, mean pooling [28] and max pooling [29], and they have different usages. Our approach employs the max pooling layer because it can pick out more important features and discard insignificant features to achieve more effective learning [29].

LSTM can retain long-term memories which is applied to many previous studies [5][6]. However, it can only keep unidirectional memories. To learn bidirectional memories, bidirectional recurrent neural network (BRNN) was published [14]. Bi-LSTM is a kind of BRNN, and it not only has the strength of learning bidirectional memories but also can learn longer memories than vanilla RNN. Previous studies [5][6][7] all employed unidirectional memories in their models. To further enhance its learning effectiveness, our proposed approach leverages the merit of Bi-LSTM architecture on comment generation.

The Bi-LSTM in the encoder computes the cell state and hidden state of the final timestep, and the states are used as the initial states of LSTM in the decoder. After the iterations of Bi-LSTM, the states of the final timestep can represent the behavior of snippets. Thus, passing the states to the LSTM of the decoder can help predict the comments representing source code behavior.

Example 3: Take Fig. 4 for an example. The Bi-LSTM layer extracts the information of the bubble sort function from the forwarding side and the backwarding side. The forwarding information and backwarding information both consist of the cell state and hidden state. The two cell states and the two hidden states are then combined into one cell state and one hidden state through a full-connected layer. The combined cell and hidden states are then used as the initial states of LSTM in the decoder.

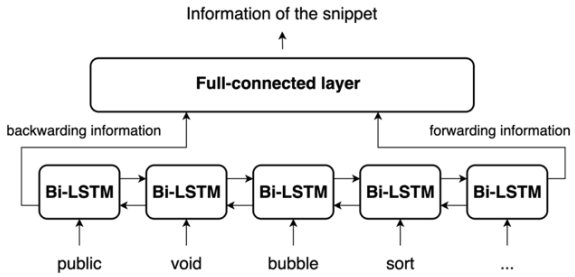

Fig. 4. An example of Bi-LSTM layer.

The final hidden state of Bi-LSTM is used in the attention mechanism. Due to the advantage of attention mechanism, numerous kinds of deep learning tasks have improved [30][31]. Thus, our proposed model also uses the attention mechanism to help generate comments. In our attention mechanism, the computations are listed as Equations (1)-(3):

$$score(e_s, h_t) = {W_1}^{\mathrm{T}} \exp(e_s * h_t)\,, \tag{1}$$

$$\alpha_{ts} = \frac{\exp(score(e_s, h_{t-1}))}{\sum_{s'=1}^{S} \exp(score(e_s, h_t - 1))}, \tag{2}$$

and

$$con_t = \sum_s \alpha_{ts}\, e_s\,, \tag{3}$$

Where e_s is the *s-th* feature from the maximum pooling layer of the encoder, h_t is the hidden state of the LSTM of decoder at the *t-th* iteration, $score(e_s, h_t)$ is the function to compute the score of e_s at the *t-th* timestep of decoder, W_1 represents the weights of the dense layer, α_{ts} represents the attention weights to the e_s at *t-th* timestep of the decoder, and con_t represents the context vector which sums the weighted features of the maximum pooling layer.

The rest of the computations in the decoder are listed as Equations (4)-(6). After the context vector is computed, con_t and h_t are combined into the final output of attention layer, out_t. Then, out_t passes through a full-connected layer, and it turns out to be p_t representing the possibilities for all candidate tokens to become the next predicted word. Then, the token chosen by the beam search algorithm is the predicted word Y_t.

$$out_t = \tanh({W_2}^{\mathrm{T}} h_t + {W_3}^{\mathrm{T}} con_t) \tag{4}$$

$$p_t = {W_4}^{\mathrm{T}} out_t \tag{5}$$

and

$$Y_t = \text{Beam Search } (p_t) \tag{6}$$

IV. EXPERIMENTAL RESULT

A. Experimental Setup

The dataset we used was derived from DeepCom [6], which can be downloaded from GitHub [32]. It consists of 588,108 pairs of snippets and comments. In this dataset, the lines of code (LOC) of some snippets are too large. Long snippets could make the training process time-consuming and result in vanishing gradients [18], which can in turn lead to an unlearnable model. Furthermore, the coding style guidelines for Google code [33] indicates that functions that exceed 40 lines are not easy to maintain and would probably become defective in the future. In the book, *Refactoring in Large Software Projects* [34], Lippert stated that "Methods should not have more than an average of 30 code lines." According to these guidelines and to avoid heavy loading for the training process, we only trained data with acceptable length. Hence, we removed data whose snippets were over 350 tokens or 40 lines. According to the analysis of our dataset, the average token length of 40-line data was 349.05, so we set the token-length threshold as 350 tokens.

Some comments in DeepCom's dataset consist of multiple sentences. Lengthy comments may make it difficult to learn the resulting long sequences, so the comments should be brief [7]. Hence, we removed the data that had comments exceeding one sentence or 30 words. The simplified dataset contained 119,233 pairs of snippets and comments, and the statistics for our simplified dataset is presented in TABLE II. Because the extremely long snippets were removed, the maximum length of snippets declined from 2,860 to 40 lines, and the average snippet length declined from 12.93 to 9.45 lines. Thus, the training processes was quickened significantly.

TABLE II:
THE STATISTICS OF THE SIMPLIFIED DATASET

Line of Code					
Max	Avg	0-10	10-20	20-30	30-40
40	9.45	69.73%	20.81%	7.03%	2.42%

In the training stage, we used TensorFlow[2] (a widely used open source library for deep learning tasks) to construct the models, and all experiments were conducted on a server with NVIDIA GPU GeForce GTX 1080 and 32GB RAM. The hyperparameters were set as follows:

1) To compare the different models fairly, the output dimensions for LSTM, embedding, and CNN layers were all set to 256.

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

2) Adam [35] optimizer was used.
3) We used cross-entropy as the loss function, and the learning rate was set to 0.001.
4) The kernel size and strides of CNN were set to 3 and 1, respectively.
5) Both the pool size of the maximum pooling layer and the strides were set to 2 to reduce the number of features.

B. *Numerical Result*

To evaluate the quality of generated comments from the trained model, several common metrics for machine translation, such as BLEU [36], CIDEr [37], and ROUGE-L [38] were employed in our research. These metrics measure the similarity between the generated comments and the ground truth comments. The evaluation results were categorized according to LOC levels. The LOC levels consist of five ranges: 0–10 lines, 10–20 lines, 20–30 lines, 30–40 lines, and 0–40 lines (overall level). The reason for this category was to observe the differences and comparisons of the model performance between short snippets and long snippets.

The first experiment aimed to verify the effectiveness of ID-splitting. We analyzed the differences in vocabulary between ID-splitting and non-ID-splitting. As TABLE III shows, the total size of the ID-splitting vocabulary was much lower than that without ID-splitting. Furthermore, ID-splitting notably reduced the percentage of rare IDs. Rare IDs here are defined as IDs whose frequencies were less than three in the training set, and this threshold of 3 setting was based on [5]. The results revealed that the IDs with compound words can be split into more common words using the ID-splitting method.

TABLE III
THE COMPARISON OF VOCABULARIES BETWEEN THE ID-SPLITTING AND THE NON-ID-SPLITTING METHOD

	With ID-splitting	Without ID-splitting
# of rare IDs	34,339	166,900
# of total IDs	69,334	265,953
% of rare IDs	49.5%	62.8%

To verify the performance of the comment prediction, we trained the models with ID-splitting and without ID-splitting, respectively, and both models were employed with $M_{CNN+BiLSTM}$[3] architecture. As Fig. 5 and TABLE IV show, the results demonstrated that ID-splitting performed better in all metrics than non-ID-splitting when the LOC was under 10 lines. When the LOC level increased, the ID-splitting method sometimes performed worse than the non-ID-splitting method. It is because ID-splitting method splits an identifier into several tokens, making the input code longer and leading to vanishing gradient problem. Despite that, the ID-splitting method still attained higher scores according to 0-40 LOC level.

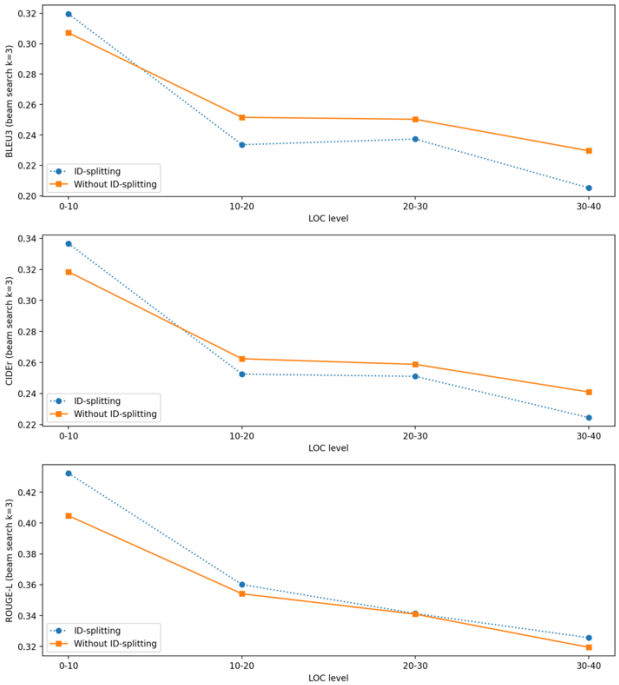

Fig. 5. The model performances of ID-splitting and non-ID-splitting.

TABLE IV
THE SCORES OF ID-SPLITTING AND NON-ID-SPLITTING METHODS AT THE 0-40 LOC LEVEL

Models	BLEU3	CIDEr	ROUGE-L
ID-splitting	**0.2930**	**0.3102**	**0.4081**
Without ID-splitting	0.2896 (-0.3%)	0.3006 (-1%)	0.3876 (-2.1%)

In this section, we compare the performance of different model architectures. To know how the different model architectures affect performance, we selected various encoder architectures and fixed the decoder to LSTM. The experiment sought to answer two questions: (i) To what extent does CNN improve the model by leveraging the dimensionality reduction and (ii) how much improvement could be obtained when the model uses bidirectional LSTM rather than pure LSTM? To present the advantages of CNN and Bi-LSTM, the experiments compared multiple models, including M_{LSTM}, $M_{CNN+LSTM}$, and $M_{CNN+BiLSTM}$. The $M_{CNN+LSTM}$ model was similar to the $M_{CNN+BiLSTM}$ model we proposed in Fig. 2; however, Bi-LSTM is replaced with pure LSTM. The M_{LSTM} model is also similar to $M_{CNN+LSTM}$ model and the difference is replacing the role of the CNN's output[4] with the hidden states of LSTM.

The performances of all model architectures are presented in TABLE V and Fig. 6. Inspecting the results, we found that the $M_{CNN+LSTM}$ model performed better than

[3] For simplicity, the model names are abbreviated in the $M_{\{encoder\}}$ format. All the models we trained in the experiments used LSTM as the architecture of the decoder. Thus, decoder was not specified in the model names. In this case, the encoder used CNN and Bi-LSTM.

[4] For simplicity, the CNN we mention here includes the max pooling layer and the CNN layer.

M_{LSTM} when the LOC was above 10 lines, and M_{LSTM} performed better when LOC was under 10 lines. This result indicated that M_{LSTM} encountered the vanishing gradient problem when LOC was above 10 lines. Furthermore, the vanishing gradient problem is not serious enough when LOC is below 10 lines, hence M_{LSTM} performs better. For Bi-LSTM, $M_{CNN+BiLSTM}$ gained some improvement over $M_{CNN+LSTM}$. The results revealed that Bi-LSTM could improve the comment generation task. The model combining CNN and Bi-LSTM was superior to the M_{LSTM} model in terms of scores.

TABLE V
THE PREDICTION SCORES OF DIFFERENT MODEL ARCHITECTURES AT THE 0-40 LOC LEVEL

Models	BLEU3	CIDEr	ROUGE-L
LSTM	0.2849 (-0.9%)	0.3055 (-0.5%)	**0.4120(+0.4%)**
CNN+ LSTM	0.2787 (-1.5%)	0.2973 (-1.3%)	0.4001 (-0.8%)
CNN+ BiLSTM	**0.2939**	**0.3102**	0.4081

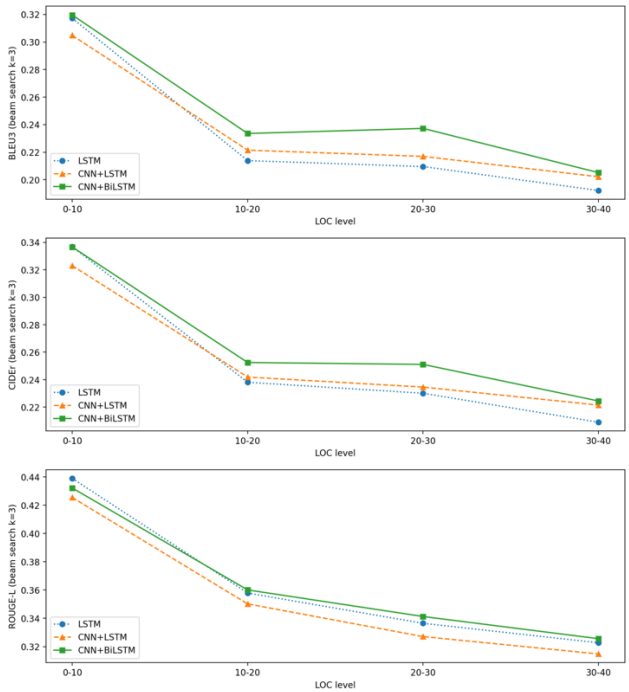

Fig. 6. The comparison of different model architectures.

According to these results, the $M_{CNN+BiLSTM}$ model is the one with the best performance among the models we trained; hence, we chose the $M_{CNN+BiLSTM}$ model as our proposed model, namely ComCNN. To compare our proposed model with other state-of-art approaches, we rebuilt the CODE-NN [5], DeepCom [6], and Hybrid-DeepCom [7] models with our proposed dataset. Although CODE-NN and DeepCom both employed greedy search as the prediction algorithm in their studies and our approach used beam search instead, we still employed beam search (B=3) on all the comparative models. We focused on the performance of different model architectures; therefore, the same prediction algorithm was applied to these models. The comparisons of the greedy search and beam search are presented later.

TABLE VI
THE PREDICTION SCORES OF COMCNN AND COMPARATIVE MODELS AT THE 0–40 LOC LEVEL

Models	BLEU3	CIDEr	ROUGE-L
ComCNN	**0.2930**	**0.3102**	**0.4081**
CODE-NN	0.2624 (-3.1%)	0.2752 (-3.6%)	0.3504 (-5.8%)
DeepCom	0.2505 (-4.3%)	0.2636 (-4.7%)	0.3444 (-6.4%)
Hybrid-DeepCom	0.2407 (-5.2%)	0.2679 (-4.2%)	0.3650 (-4.3%)
IR	0.0033 (-28.9%)	0.0188 (-29.1%)	0.0530 (- 35.5%)

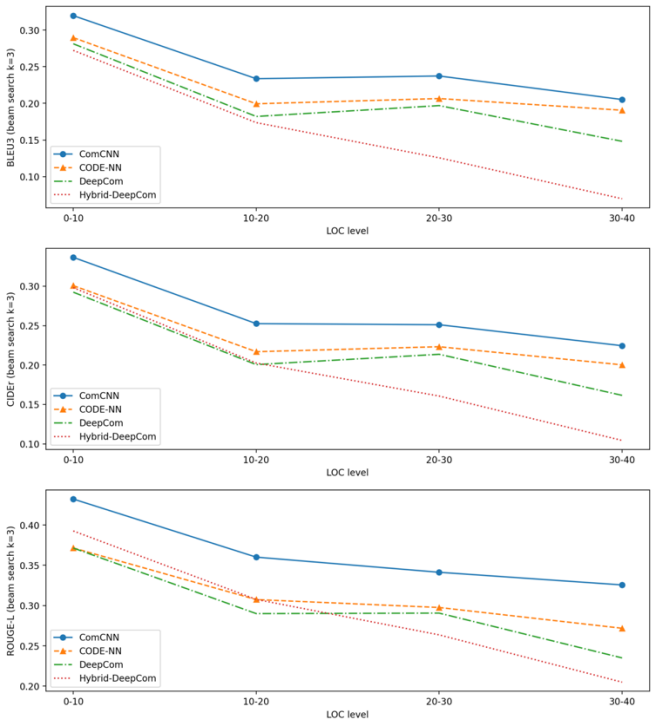

Fig. 7. The comparisons between our proposed model and comparative models.

The comparisons of our proposed model and other state-of-art models are depicted in TABLE VI[5] and Fig. 7[6]. The results indicated that ComCNN outperformed other models at every LOC level. The results indicated that ComCNN significantly improved the quality of comment prediction with other deep learning based approaches. Furthermore, the deep learning based models obviously outperformed the IR approach. The result thus confirms the strength of the deep learning models when compared to the manual-designed approach.

[5] The models are evaluated when the overall level (LOC 0–40) is considered.

[6] We did not present the performance of the IR approach in the line graph, because the lower metric scores of the IR approach would make the range of the y-axis larger and thus make it difficult for the line graph to depict the differences in scores among the deep learning based models.

To compare the different prediction algorithms, the performances of both greedy search and beam search were evaluated for every model we trained in the experiments. TABLE VII presents the results for the ComCNN model. The scores for beam search was higher in every metric than scores for greedy search, proving that the beam search could resolve the hidden defect of greedy search. In the beam search algorithm, the "*B*=3" setting performed slightly better than the "*B*=5" setting in almost all evaluations.

TABLE VII
THE COMPARISON OF GREEDY SEARCH AND BEAM SEARCH AT THE 0–40 LOC LEVEL

ComCNN	BLEU3	CIDEr	ROUGE-L
Greedy Search	0.2874	0.3045	0.4028
Beam Search (*B* = 3)	**0.2930**	**0.3102**	**0.4081**
Beam Search (*B* = 5)	0.2925	0.3093	0.4070

In addition to the performance for predicting comments, the time and memory consumption should also be considered. As TABLE VIII shows, the training time cost of ComCNN was much less than that of the comparative models. The training time was calculated by maximizing the batch size to fully use the 8-GB memory of our GPU. The GPU memory costs are also demonstrated in TABLE VIII, and the evaluation was based on the same batch size, 10. The results indicated that ComCNN costs much less GPU memory compared to the comparative models.

TABLE VIII
THE COMPARISONS OF TIME AND GPU MEMORY COST FOR THE TRAINING PROCESS

Models	Training time per epoch	GPU memory cost
ComCNN	17 mins	1423 MiB
CODE-NN	47 mins	2449 MiB
DeepCom	67 mins	4495 MiB
Hybrid-DeepCom	424 mins	4505 MiB

In TABLE IX, the predicted comments of the source code from our proposed model and other state-of-art models are presented. These examples came from our testing set, and some interesting cases were listed.

TABLE IX
THE OUTPUT EXAMPLE OF OUR PROPOSED MODEL AND COMPARATIVE MODELS

#	Java Code and Predicted Comments	

1	Code	public static void e(String msg){
		if (LOG_ENABLE) {
		Log.e(TAG,buildMsg(msg));
		}
		}

	ComCNN	Utility method to parse an error message
	CODE-NN	verbose log
	DeepCom	<UNK> warn message
	Hybrid-DeepCom	the debug log message
	IR	Replaces the cached tray icons with <UNK> <UNK> ones , fixing a Java bug which gives them <UNK> transparency

#	Java Code and Predicted Comments	

2	Code	public static int max(final int a,final int b){
		return (a <= b) ? b : a;
		}

	ComCNN	Returns the maximum int in integer
	CODE-NN	Return the greatest common divisor of two polynomials
	DeepCom	Returns the maximum of two elements
	Hybrid-DeepCom	Compute the maximum of two values
	IR	Creates a new instance of <UNK>

#	Java Code and Predicted Comments	

3	Code	public void write(int a) throws IOException {
		outs.write(a);
		position++;
		}

	ComCNN	Writes a text into the output stream
	CODE-NN	Writes a number of type long in little endian
	DeepCom	Write a String to an output stream
	Hybrid-DeepCom	Append a single output to the output stream
	IR	Clears the back edges for the basic block passed

C. Research Questions

RQ1: How much improvement does ID-splitting contribute?

This RQ aims to verify the extent to which vocabulary size is reduced and whether this textural processing to input data can improve prediction performance. TABLE III and Fig. 5 show that employing ID-splitting significantly reduces the vocabulary size and reduces rare IDs to 13.3%. Furthermore, the model using the ID-splitting method attained higher metric scores for the overall LOC level. Therefore, the ID-splitting method actually improves comment generation.

RQ2: Does the vanishing problem be resolved by using CNN?

We raised this RQ to verify whether CNN could resolve the vanishing gradient problem and improve comment prediction for long snippets. As Fig. 6 shows, the M_{LSTM} model performed well in short snippets. However, the performance declined sharply when the LOC level increases. It clearly demonstrates the vanishing gradient problem when using the M_{LSTM} model. Conversely, $M_{CNN+LSTM}$ attained higher scores for BLEU3 and CIDEr than the M_{LSTM} model when LOC was above 10 lines, which indicated that the vanishing gradient problem is reduced when CNN is used.

RQ3: How accurate is ComCNN for comment generation when compared to the model architectures from other studies?

The RQ aims to verify whether the predicted comments can be more precise than those from other studies. Fig. 7 shows that ComCNN attains higher scores for all metrics than the comparative models. ComCNN attained higher BLUE3, CIDEr, and ROUGE-L scores than CODE-NN (+3.1%, +3.6%, and +5.8%), DeepCom (+4.3%, +4.7%, and +6.4%), and Hybrid-DeepCom

(+5.2%, +4.2%, and +4.3%). The results indicate that combining our model architecture and ID-splitting method led to considerable improvement in comment generation task.

RQ4: What are the time and GPU memory consumption of ComCNN and other approaches?

The evaluation of time cost and GPU memory consumption is presented in TABLE VIII. With the same batch-size value, ComCNN costs only 0.59× the GPU memory consumption of CODE-NN, 0.32× the consumption of DeepCom and 0.31× the consumption of Hybrid-DeepCom during the training process. In other words, ComCNN gained 41%, 68%, and 69% improvement on GPU memory consumption compared to CODE-NN, DeepCom and Hybrid-DeepCom, respectively. The main factor leading to reduced GPU memory cost was the dimensionality reduction of CNN. Dimensionality reduction reduces the memory cost in the later layers of ComCNN's neural networks. The heavy memory cost of DeepCom and Hybrid-DeepCom is caused by the long inputs from its data preprocessing approach. Hence, the DeepCom approach is only suitable for short snippets as opposed to long snippets.

In terms of training time, ComCNN was 2.7×, 3.9×, and 24.9× faster than CODE-NN, DeepCom, and Hybrid-DeepCom. The main cause of the shorter training time for ComCNN was the lower GPU memory consumption per batch. The lower GPU memory consumption per batch increases the maximum batch size that the GPU can afford, thus saving the training time. In short, the greater the GPU memory per batch they cost, the slower the training speeds they have.

V. CONCLUSION

Comment generation can automatically create comments describing the behavior of source code and can further resolve the comprehension issue of source code. Previous studies on comment generation [5][6][7] have demonstrated the strength of deep learning compared to manually designed approaches. To solve the challenges of the deep learning based approaches, we proposed a CNN based model architecture to solve the vanishing gradient problem. In addition, we applied the ID-splitting method to solve the OOV problem, and we decided to use the beam search algorithm to solve the defects of the greedy search. To remove the noisy data, we also demonstrated how to modify datasets from DeepCom [6] and finally released a new dataset consisting of 119,233 data units.

Experimental results reveal that ComCNN attains 3% ~ 7% higher prediction scores than the comparative models [5][6][7]. ComCNN gains 41%, 68%, and 69% improvement on GPU memory consumption compared to comparative models and has 1.7×, 2.9×, and 23.9× the training speed of comparative models. Our study represents progress in comment generation, and we hope it encourages further research on comment generation based on CNN-based architecture.

Finally, some open issues remain: (i) The ID-splitting method increases the length of input source code and may lead to the vanishing gradient problem; (ii) some noisy data still remain in the dataset, even after the elimination of some redundant data; (iii) although ComCNN demonstrates superb performance for Java code in this study, the generalizability of ComCNN to other programming languages still needs to be considered. In the future, we plan to work on solutions to the unresolved issues that have been outlined. In addition, in order to highlight the advantages of ComCNN, we also plan to conduct more comparative studies using other models.

REFERENCES

[1] S. C. B. de Souza, N. Anquetil, et al., "A Study of the Documentation Essential to Software Maintenance," *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, pp. 68-75, New York, NY, USA, Sep, 2005.

[2] Smartcomments, "SmartComments," 2014. [Online]. Available: http://smartcomments.github.io/#home. [Accessed: 30- Jun- 2020]

[3] E. Wong, J. Yang, et al., "AutoComment: Mining Question and Answer Sites for Automatic Comment Generation," *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 562-567, Silicon Valley, CA, USA, Nov, 2013.

[4] P. W. McBurney and C. McMillian, "Automatic Documentation Generation via Source Code Summarization of Method Context," *Proceedings of IEEE/ACM International Conference on Program Comprehension (ICPC)*, pp. 279-290, New York, NY, USA, Jun, 2014.

[5] S. Iyer, I. Konstas, et al., "Summarizing Source Code using a Neural Attention Model," *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, vol. 1, pp. 2073-2083, Berlin, Germany, Aug, 2016.

[6] X. Hu, G. Li, et al., "Deep Code Comment Generation," *Proceedings of IEEE/ACM International Conference on Program Comprehension (ICPC)*, pp. 200-210, New York, NY, USA, May, 2018.

[7] X. Hu, G. Li, et al., "Deep Code Comment Generation with Hybrid Lexical and Syntactical Information," *Empirical Software Engineering (EMSE)*, vol. 25, pp. 2179-2217, Jun, 2019.

[8] Y. R. Zeng, "Improving Automatic Generation of Source Code Comments using Convolutional Neural Networks and ID-splitting Method," M.S. thesis, Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, Aug, 2020.

[9] T. Luong, I. Sutskever, et al., "Addressing the Rare Word Problem in Neural Machine Translation," *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, vol. 1, pp. 11-19, Beijing, China, July, 2015.

[10] C. C. Huang, H. C Yen, et al., "Using Sublexical Translations to Handle the OOV Problem in Machine Translation," *ACM Transactions on Asian Language Information Processing (TALIP)*, vol. 10, iss. 3, no. 16, Sep, 2011.

[11] S. Hochreiter and J. Schmidhuber, "Long Short-Term

Memory," *Neural Computation*, vol. 9, iss. 8, pp. 1735-1780, Nov, 1997.

[12] N. Kalchbrenner, E. Grefenstette, et al., "A Convolutional Neural Network for Modelling Sentences," *Proceedings of the Association for Computational Linguistics (ACL)*, vol. 1, pp. 655-665, Baltimore, Maryland, Jun, 2014.

[13] A. Graves and J. Schmidhuber, "Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures," *Neural Network*, vol. 18, iss. 5-6, pp. 602-610, July, 2005.

[14] M. Schuster and K. K. Paliwal, "Bidirectional Recurrent Neural Network," *IEEE Transactions on Signal Processing*, vol. 45, iss. 11, pp. 2673-2681, Nov, 1997.

[15] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," *Doklady Physics*, vol. 10, no. 8, pp.707-710, Feb, 1966.

[16] J. Zheng, X. Wang, et al., "Retrieval-based Neural Source Code Summarization," *Proceedings of 42nd International Conference on Software Engineering (ICSE)*, Seoul, South Korea, July, 2020.

[17] C. Tillmann and H. Ney, "Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation," *Computational Linguistics*, vol. 29, no. 1, pp. 97-133, Mar, 2003.

[18] S. Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 2, pp. 107-116, Apr, 1998.

[19] c2nes, "Javalang, Pure Python Java Parser and Tools," 2012. [Online]. Available: https://github.com/c2nes/javalang. [Accessed: 19- Mar- 2020]

[20] H. A. Basit, S. J. Puglisi, et al., "Efficient Token Based Clone Detection with Flexible Tokenization," *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, pp. 513-516, New York, NY, USA, Sep, 2007.

[21] D. Binkley, M. Davis, et al., "To Camelcase or Under_score," *Proceedings of IEEE International Conference on Program Comprehension (ICPC)*, pp. 158-167, Vancouver, Canada, May, 2009.

[22] B. Sharif and J. I. Maletic, "An Eye Tracking Study on camelCase and under_score Identifier Styles," *Proceedings of IEEE International Conference on Program Comprehension (ICPC)*, pp. 196-205, Braga, Minho, Portugal, July, 2010.

[23] I. Sutskever, O. Vinyals, et al., "Sequence to Sequence Learning with Neural Networks," *Advances in Neural Information Processing Systems*, vol. 4, Sep, 2014.

[24] G. Lample and F. Charton, "Deep Learning for Symbolic Mathematics," *arXiv:1912.01412*, Dec, 2019.

[25] D. Adiwardana, M. T. Luong, et al., "Towards a Human-like Open-Domain Chatbot," *arXiv:2001.09977*, Jan, 2020.

[26] E. Grefenstette, P. Blunsom, et al., "A Deep Architecture for Semantic Parsing," *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pp. 22-27, Baltimore, MD, USA, Jun, 2014.

[27] Y. Shen, X. He, et al., "Learning Semantic Representations Using Convolutional Neural Networks for Web Search," *Proceedings of the International Conference on World Wide Web*, pp. 373-374, New York, NY, USA, Apr, 2014.

[28] S. Mittal, "A Survey of FPGA-based Accelerators for Convolutional Neural Networks," *Neural Computing and Applications*, vol. 32, pp. 1109-1139, Oct, 2018.

[29] A. Giusti, D. C. Cireşan, et al., "Fast Image Scanning with Deep Max-pooling Convolutional Neural Networks," *Proceedings of IEEE International Conference on Image Processing (ICIP)*, pp. 4034-4038, Melbourne, VIC, Australia, Sep, 2013.

[30] D. Bahdanau, K. Cho, et al., "Neural Machine Translation by Jointly Learning to Align and Translate," *arXiv:1409.0473v7*, May, 2016.

[31] A. M. Rush, S. Chopra, et al., "A Neural Attention Model for Abstractive Sentence Summarization," *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 379-389, Lisbon, Portugal, Sep, 2015.

[32] xing-hu, "Dataset for Comment Generation from DeepCom," Available: https://github.com/xing-hu/DeepCom, 2018 [Accessed: 25- Mar- 2020]

[33] Google, "Google C++ Style Guide," Available: https://google.github.io/styleguide/cppguide.html#Write_Short_Functions, [Accessed: 25- Mar- 2020]

[34] M. Lippert and S. Roock, *Refactoring in Large Software Projects*, Wiley, May, May, 2006.

[35] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *Proceedings of International Conference for Learning Representations (ICLR)*, San Diego, CA, USA, May, 2015.

[36] K. Papineni, S. Roukos, et al., "BLEU: a Method for Automatic Evaluation of Machine Translation," *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 311-318, Philadelphia, PA, USA, July, 2002.

[37] R. Vedantam, C. L. Zitnick, et al., "CIDEr: Consensus-based Image Description Evaluation," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4566-4575, Boston, MA, USA, Jun, 2015.

[38] C. Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, pp. 74-81, Barcelona, Spain, July, 2004.