ITC 2/54
Information Technology and Control
Vol. 54 / No. 2/ 2025
pp. 643-661
DOI 10.5755/j01.itc.54.2.36699

Hybrid Attention Approach for Source Code Comment Generation

Received 2024/03/18 | Accepted after revision 2025/01/15

**HOW TO CITE:** Meng, Y. (2025). Hybrid Attention Approach for Source Code Comment Generation. *Information Technology and Control*, 54(2), 643-661. https://doi.org/10.5755/j01.itc.54.2.36699

# Hybrid Attention Approach for Source Code Comment Generation

**Yao Meng**
North China University of Water Resources and Electric Power, Zhengzhou 450046, China

**Corresponding author:** mengyao@ncwu.edu.cn

Currently, developers are often obligated to enhance code quality. High-quality code is often accompanied with comprehensive summaries, including code documentation and function explanations, which are invaluable for maintenance and further development. Regrettably, few software projects provide sufficient code comments owing to the high costs associated with human labeling. Contemporary researchers in software engineering concentrate on the methods for automated comment generating. Initial algorithms depended on handwritten templates or information retrieval methods. With the advancement of machine learning, researchers construct automated models for machine translation instead. Nonetheless, the produced code comments remain inadequate owing to the significant disparity between code structure and normal language. This study introduces a unique deep learning model, At-ComGen, which utilizes hybrid attention for the automated creation of source code comments. Utilizing two separate LSTM encoders, our approach integrates essential tokens from source code functions with the code structure, represented by a corresponding Abstract Syntax Tree. In contrast to earlier data-driven models, our methodology utilizes code syntax and semantics in the generation of comments. The hybrid attention method, used for comment creation for the first time to our knowledge, enhances the quality of code comments. The tests demonstrate that At-ComGen is efficacious and surpasses other prevalent methodologies. Machine comments from Seq2Seq and CODE-NN disregard code structure underlying DeepCom and At-ComGen. At-ComGen has 59.3%, 36.4%, 43.3%, and 13.1% higher comment BLEU values than baseline models for a 5-line function. Even though model performance reduces with comment length, At-ComGen's comments often outperform others. 5–10-word machine comments work best. For reference length 10, At-ComGen has 38.2%, 23.7%, 9.3%, and 4.4% greater BLEU values than the other baseline models.

**KEYWORDS:** Code Comment Generation; Attention; Abstract Syntax Tree.

## 1. Introduction

In contemporary civilization, essential aspects of existence, including natural resources, healthcare, and public safety, rely on the operation of high-quality software. Nonetheless, software development has always been very expensive. Software developers strive to enhance code quality.

Software maintenance requires up to 90% of software engineering efforts, with a significant portion of time allocated to comprehending the existing code and associated documentation. Despite the plethora of coding standards established by software engineering authorities, few developers are inclined to fully comment their code for future maintenance. Novice developers also find it challenging to provide adequate comments for source code. Methods for annotating code snippets have been introduced throughout the previous decade. Early researchers produced code comments using manual templates or content selection methods. Sridhara et al. [26] produced textual comments using several templates, each formulated based on assertions in source code. Sridhara et al. [27] used a particular template derived from SWUM, an NPL approach for identifying short words concealed in code, to provide concise summaries. Nevertheless, these transformation rules are established by domain specialists, making them challenging to use in other programming languages [3].

In the age of deep learning, researchers are commencing the generation of source code comments using neural network models. The promising technology of neural machine translation (NMT), established within the NLP academic community, has lately garnered the interest of software programmers. A standard neural machine translation (NMT) system converts one natural language into another using sequence-to-sequence learning, such as translating an English phrase into a French sentence. In software engineering, individuals may use Neural Machine Translation (NMT) to produce comments: the lexicon, tokens, or structural information of source code constitute one sequence, while the intended natural language summary serves as the target language. Owing to their superior transformation accuracy and generalizability, deep learning methodologies rapidly supplanted conventional techniques for comment creation.

We provide a source code comment generation model named At-ComGen, which employs two distinct LSTM encoders integrated with a hybrid attention mechanism, preserving both lexical and structural information in code representation. The lexical encoder utilizes token information by extracting words and IDs from the source code, while the syntactical encoder derives structure information using a specialized traversal technique. The syntactical encoder first transforms a particular code snippet into an Abstract Syntax Tree (AST) and then divides it into a series of statement trees. Each subtree signifies a valid code statement. The encoder then inputted all the vectors, each representing a subtree, into an LSTM network sequentially. The integrated decoder in At-ComGen amalgamates the outputs from both encoders with a hybrid attention mechanism, producing the comment through a sequence-to-sequence learning framework. Through comparative studies, we ascertain that our model surpasses the majority of prevalent state-of-the-art methodologies.

Our contributions in this work are delineated as follows:

We provide a deep learning model for generating source code comments, using two separate LSTM encoders to process code tokens and structures. In contrast to other methodologies, the comments produced by our model preserve both lexical and structural characteristics of the code.

We provide an innovative statement tree traversal approach for extracting structural information from code, in contrast to conventional AST traversal strategies. Our technique divides the whole AST into sequential subtrees based on code statements, therefore reducing tree depth and complexity. The approach successively navigates each statement tree based on the naturalness of programming languages.

We are the pioneers in using the hybrid attention mechanism, first established in natural language processing, to the production of source code comments. The hybrid attention method establishes a strong correlation between code tokens, such as identifiers, and code structure, ultimately enhancing the precision of produced comments.

The remainder of the paper is structured as follows. Section 2 delineates the relevant literature in this domain. Section 3 delineates the notations and foundational concepts pertinent to automated comment production. Section 4 delineates the framework of our model. Section 5 presents the comparative experiments, whereas Section 6 analyzes the experimental results. In conclusion, we finalize the paper in Section 7.

### 1.1 Motivation and Contribution

The development of a hybrid attention approach for the purpose of writing source code comments has as its primary objective the elimination of the issues that are related with the methods that are now in use. When employing typical approaches, it may be difficult to capture the whole context of a code snippet, which may result in comments that are either erroneous or irrelevant to the code sample. Understanding the semantic connections that exist between the various pieces of code is essential for the generation of comments that are pertinent, but it is not a simple task to do. Most of the approaches that are now accessible may need a significant amount of processing power, particularly for big codebases. The proposed approach integrates many attention processes, enabling it to capture various elements of the code, such as syntax, semantic relationships, and context, among others. As a result, a hybrid attention strategy was devised to address these challenges. The proposed hybrid attention approach may enhance the understanding of a code snippet's context, leading to the generation of more accurate and relevant comments. The improved ability of the provided model to capture the semantic relationships across different portions of the code results in more coherent and meaningful comments. The hybrid attention strategy may be superior to current tactics, especially for large codebases.

## 2. Related Work

Recently, the source code summarization has drawn great attention in software engineering. Generally speaking, the related research can be classified as rule/template based approaches [26], statistical language model approaches [6] and deep learning approaches [18, 1].

### 2.1 Traditional Approaches

In the initial study, people manually crafted templates to produce comments from source code. Researchers [26, 3, 9] often generate code annotations using diverse templates. If a code snippet conforms to the contrived template, the associated summary will be generated automatically.

Information retrieval methods are also included in code summarizations. The vector space model and latent semantic indexing, prevalent in information retrieval, are used to generate code comments for classes and functions [10]. A number of researchers have tried to produce code comments via methods used in code cloning, since analogous code snippets may elicit comparable remarks [31]. Table 1 presents a comparison with state of art methods.

### 2.2 Deep Learning Approaches

Researchers first provide an attentional RNN for generating product remarks for SQL and C# [6]. Despite the popularity of the NMT approach in NLP, it has not been used to source code summarizing. The majority of code summarizing methods use the traditional attentional sequence-to-sequence structure [1]. The comparative trials demonstrate that their machine-generated remarks surpass those produced by traditional models.

Alongside the aforementioned token-extraction methods, individuals endeavor to extract the structures of source code while creating comments. Classical NLP techniques may overlook the concealed subtleties in the source code, which include both lexical and semantic information. Hu et al. [13] offer a model named DeepCom that generates comments by navigating the relevant Abstract Syntax Tree (AST) derived from the provided code sample. They encode the Abstract Syntax Tree using an attentional LSTM network and produce the comment using an LSTM decoder. They use an innovative method, SBT, which preserves all the information in the AST throughout traversal.

## 3. Background

### 3.1 Language Model

The language model is a probabilistic model which produces sentences in a human language. It com-

putes the probability of occurrence of a number of words in a specific sentence. For a sentence $y$, where $y = (y_1, \ldots, y_T)$ is a sequence of words, the language model estimates the joint probability of its words $P_r(y_1, \ldots, y_T)$:

$$P_r(y_1,...,y_T) = \prod_{t=1}^{T} P_r(y_t \mid y_1,...,y_{t-1}) \quad (1)$$

It is equal to estimate the probability of each word in $y$ given its previous words.

As we know, it is difficult to calculate $P_r(y_t|y_1, \ldots, y_{t-1})$, in early days we use N-gram [14] to approximate it. The equation (1) is simplified as follows:

$$P_r(y_t \mid y_1,...,y_{t-1}) \approx P_r(y_t \mid y_{t-n+1},...,y_{t-1}) \quad (2)$$

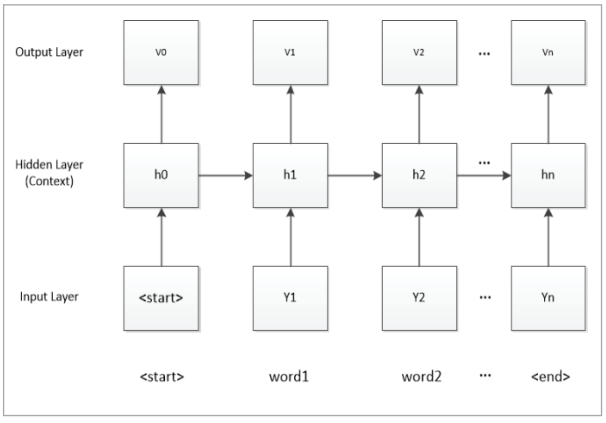
where an n-gram means n consecutive words. The approximation denotes the next word $y_t$ is conditioned on the previous $n-1$ words.

The original language model based on n-gram suffers obvious limitations [15, 16]. For instance, n-gram model probabilities cannot be generated from the frequency counts as the generated models might have serious problems when confronted with n-grams which have never been seen before.

Recent researchers have begun to apply deep learning to replace traditional approaches in every field of computing industry. Unlike the n-gram model that predicts a word according to a fixed number of predecessors, a neural language model predicts a target word with far away predecessors by a recurrent neural network (RNN). Figure 1. shows the RNN structure for sentence estimation, which includes three layers. The input layer transforms words to specific vectors. The hidden layer recurrently calculates and updates a hidden state after reading each word. The output layer computes the probability of the next word by the current hidden state.

**Figure 1**

RNN for sentence estimation.




In this section, we focus on the workflow of the neural language model. In order to realize equation (1), the RNN model loads the words sequentially and predicts the next word at each time step. At step $t$, it calculates the next word probability by three steps: (a) the current word $y_t$ is transformed to a vector according to the input embedding layer $e$. (b) it produces the hidden state $h_t$ at timestep $t$ by the previous hidden state $h_{t-1}$ and the current input $y_t$:

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

(c) the target word probability $p_r(y_{t+1}|y_1, \ldots, y_t)$ is predicted by the current hidden state $h_t$:

$$p_r(y_{t+1} \mid y_1,...,y_t) = g(h_t) \quad (4)$$

where $g$ is typically a softmax function that generates the output tokens.

### 3.2 BERT Pre-trained Model

BERT (Bidirectional Encoder Representations from Transformers) is a well-known pre-trained model proposed by Google's AI team in 2018 [5]. Specialized in the word-embedding task, the BERT model is the encoder part of a multi-layer bidirectional Transformer network. When people use the BERT model for encoding, they only need to fine tune the original pre-trained model, without the need to retrain the encoder model for specific tasks. Instead of a complete training from scratch for a given task, experts simply fine tune the encoder in BERT during word embedding.

Figure 2 depicts the basic structure of BERT model, where $T_1, T_2, \ldots T_N$ indicate input words in the model, $E_1$, $E_2, \ldots E_N$ indicate the model's output, i.e., the encoding vectors of input words. The encoding of input words in BERT is implemented by the multi-tiered bidirectional Transformer modules (Trm). Transformer is a complicated sequential model based on the multi-head attention mechanism. Different approaches of word embedding are provided by BERT developers

depending on the quality and arrangement of Trms.

BERT captures token context. Code and context knowledge is crucial for source code comment development. The pre-trained language model BERT as shown in Figure 2 has plenty of text data. This learns NLP word and phrase representations for code development. BERT's design readily interacts with attention techniques, which are needed to capture code connections and generate context-aware comments. BERT may pre-train a language model on a huge corpus of source code and comments to learn domain-specific representations for code creation. BERT might encode or decode code comments in a sequence-to-sequence paradigm. Extracting features from source code using BERT might feed a comment creation mode.

**Figure 2**
Structure of BERT.

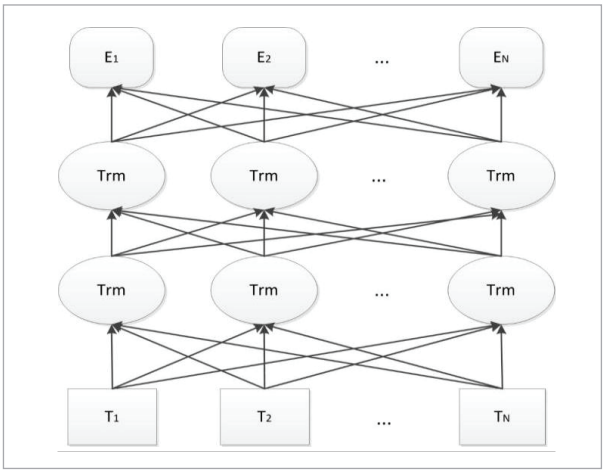



**Table 1**
Comparison with state of art methods.

| Author(s) | Paper Title | Key Contributions | Limitations |
|---|---|---|---|
| Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin (2020) [13] | Deep code comment generation with hybrid lexical and syntactical information | Comprehend programs and reduce additional time spent on reading and navigating source code | The comments are often mismatched, missing or outdated in software projects. Developers have to infer the functionality from the source code |
| Boao Li, Meng Yan, Xin Xia, Xing Hu and Ge Li (2020) [21] | DeepCommenter: A deep code comment generation tool with hybrid lexical and syntactical information | DeepCommenter formulates the comment generation task as a machine translation problem and exploits a deep neural network that combines the lexical and structural information of Java methods | Code comments are missing, mismatched or outdated due to tight development schedule or other reasons |
| Jiho Shin, Jaechang Nam (2021) [24] | A Survey of Automatic Code Generation from Natural Language | Surveys the approaches that generate source code automatically from a natural language description | The cost of learning different programming languages is high for novice developers |
| Frank F. Xu, Bogdan Vasilescu, and Graham Neubig (2022) [32] | In-IDE Code Generation from Natural Language: Promise and Challenges | Implements a hybrid of code generation and code retrieval functionality | Turning concept into code, especially when dealing with the APIs of unfamiliar libraries |
| Marija Kostić , Vuk Batanović, Boško Nikolić (2023) [19] | Monolingual, multilingual and cross-lingual code comment classification | Addresses the problem of code comment classification not only in the monolingual setting, but also in the multilingual and cross-lingual one | Dataset was manually annotated according to a newly proposed taxonomy of code comment categories |
| This work | A Hybrid Attention Approach for the Source Code Comment Generation | A novel deep learning model At-ComGen, which is based on the hybrid attention for the automatic generation of source code comment | The proposed model's efficiency depends on training data amount and quality. Data that is incomplete or distorted may not be reliable. The large codebases of hybrid attention models make them computationally expensive. Hybrid attention models may struggle to generalize to code from different domains or languages due to conventions and semantics. |

## 4. The Proposed Approach

In Section 4, we present our model At-ComGen, which is able to generate accurate comments for source code snippets. When generating the comments in natural language, the model extracts both tokens and structural information from the given code with hybrid attention mechanism. Figure 3 summarizes the overall architecture of At-ComGen.

### 4.1 Framework

As shown in Figure 3, At-ComGen is implemented with a sequence-to-sequence learning framework, which contains two parts: the encoder and the decoder. The encoder model on the left transforms code into high dimensional vectors with two independent LSTM networks. The decoder generates the comment from the output vectors from the two encoders with hybrid attention mechanism.

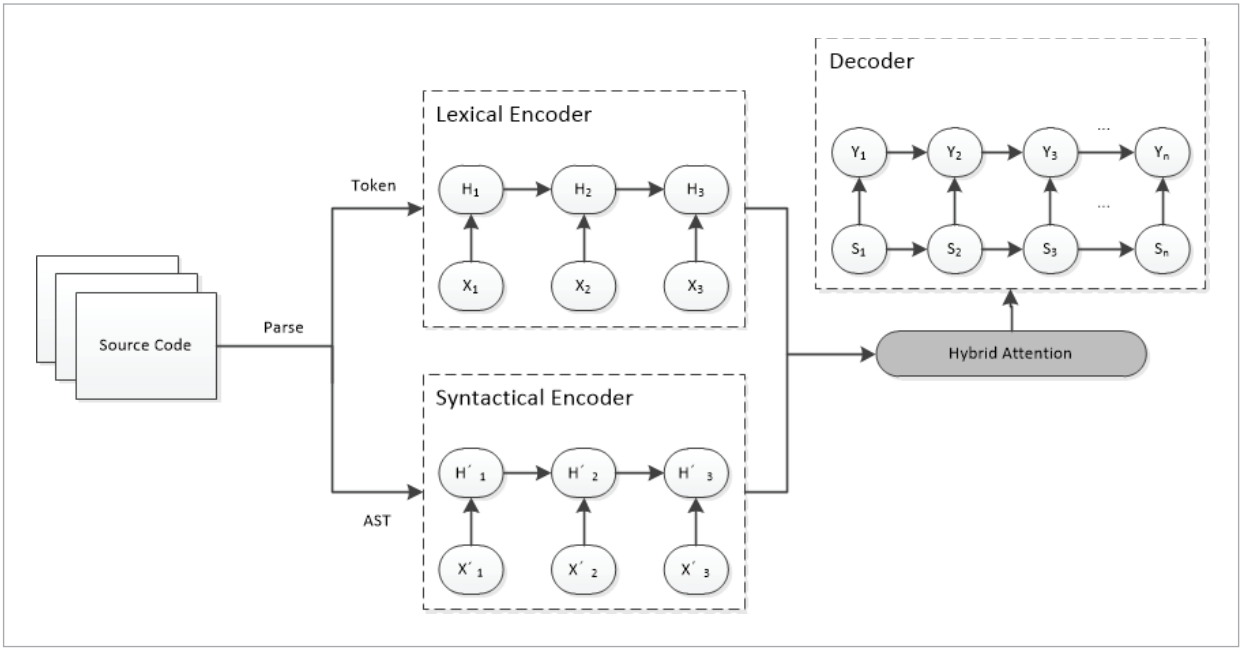
The first LSTM encoder is a lexical encoder which extracts the important token information from the source code with traditional NLP techniques. The second LSTM encoder is a syntactical encoder which extracts structural information with a specific AST traversal algorithm. The right part of the graph is a decoder, which is another LSTM network. It generates the code comments according to the output vectors of encoders with hybrid attention. As the encoder model retains both the lexical and structural information, code comments generated from our model describes code snippets exactly. The detailed components of At-ComGen are described in the following subsections.

### 4.2 Encoders

#### 4.2.1 Lexical Encoder

Many comments are extracted from code tokens, such as function names, variable names, identifiers and so on. It is a natural way to generate comments by combines the useful information hidden from these important words. Early approaches focus on the extraction of tokens from the source code snippets when generating code comments [18]. In our model, the lexical encoder sequentially input the embedding of core tokens into a unidirectional LSTM network. For a code snippet $X = x_1, x_2, \ldots, x_n$, the lexical encoder input a token $x_t$ of the sequence at each time step $t$, then updates the current hidden state by the following equation:

**Figure 3**
The Architecture of At-ComGen.



$$h_t = f(h_{t-1}, x_t) \tag{5}$$

where f is a LSTM unit which maps a word of source language $x_t$ into a hidden state $h_t$. After the recurrent computation, the hidden states of the encoded source code are $h = [h_1, h_2, \ldots, h_n]$.

In the process of token selections, our model filters individual words from the codes such as numbers, operators, the string values, etc. These unnecessary tokens will lower the accuracy of generated comments. The user-defined identifiers usually contain several language words. We split the identifiers by the camel indication or underscore in order to reduce the UNK indicators in the generated corpus. The process of token extraction is detailed later.

#### 4.2.2 Syntactical Encoder

The source codes are different from plain texts, as it contains complicated structural information which is difficult to express with common NLP methods. In our model, another LSTM network called syntactical encoder is employed to learn structural information from AST sequences. Our syntactical encoder sequentially input the embeddings of AST nodes into a unidirectional LSTM network. For an AST sequence $X' = x'_1, x'_2, \ldots, x'_n$, the syntactical encoder input an AST node $x'_t$ of the sequence at each time step $t$, then updates the current hidden state $h'_t$ by the following equation:

$$h'_t = f(h'_{t-1}, x'_t) \tag{6}$$

where f is another LSTM. Finally, the hidden states of the encoded AST are $h' = [h'_1, h'_2, \ldots, h'_n]$.

The sequence of input nodes is determined by AST-traversal algorithms, which will affect the accuracy of generated comments.

### 4.3 Decoder with Hybrid Attention

In our model, the decoder is another unidirectional LSTM, which produces the target sequence $y$ by predicting the probability of each word $y_i$, given context vector $c_i$ and all the previously predicted words $y_1, y_2, \ldots, y_{i-1}$ [14]:

$$p(y_i \mid y_1, y_2, \ldots, y_{i-1}) = g(h_{i-1}, c_t) \tag{7}$$

where g is used to calculate the probability of the word $y_t$.

With the attention mechanism, the translation model generates the target word according to the various contribution of each input token. At-ComGen is designed to extract information from both code tokens and AST sequences. The hybrid attention mechanism projects the hidden states of two independent encoders into a shared space and computes the distributions.

Our model defines the unified context vector $c_i$ in order to predict each target word $y_i$ as a weighted sum of all hidden states in two encoders. $c_i$ is calculated as follows:

$$c_i = \sum_{j=1}^{m} \alpha_{ij} h_j + \sum_{j=1}^{n} \alpha'_{ij} s'_{ij} \tag{8}$$

where $\alpha$ and $\alpha'$ are attentional distributions of code tokens and AST sequences respectively. The attention score $\alpha_{ij}$ of each hidden state $h_j$ is calculated as follows:

$$\alpha_{ij} = \frac{\exp(\varepsilon_{ij})}{\sum_{k=1}^{m} \exp(\varepsilon_{ik})} \tag{9}$$

where

$$e_{ij} = a(s_{i-1}, h_j) \tag{10}$$

In the same way, another attention score $\alpha'_{ij}$ is computed as

$$\alpha'_{ij} = \frac{\exp(e'_{ij})}{\sum_{k=1}^{m} \exp(e'_{ik})} \tag{11}$$

where

$$e'_{ij} = a(s_{i-1}, h'_j) \tag{12}$$

### 4.4 AST Traversal Algorithm

It is challenging to make a proper translation from a source code to a natural language. If we only generate the comments according to the sequence of tokens, i.e. we view the code as plain text, the lost syntactical information will cause serious inaccuracies.

In order to retain structural information, translation models have to apply AST traversal algorithms.

Many approaches have been taken to traverse the AST. One simple way is to use a classical pre-order or post-order traversal. However, these algorithms might cause information lost as the old ASTs cannot be reconstructed unambiguously. Moreover, the generated ASTs are too deep to traverse due to long-term dependency. Recent experts attempt to traverse the ASTs with a RvNN or CNN [30, 33, 34]. However, these approaches suffer from high computation costs, and the ASTs have to be transformed to a regular form, e.g. complete binary tree.

In this paper, we present a novel AST traversal algorithm in our model. Basically, the traversal process is divided into two steps: (1) In order to reduce the computational complexity, the whole AST is split into a sequence of subtrees by the granularity of the language statement. (2) Each statement tree is traversed by a classical pre-order algorithm. Compared to other recursive traversal algorithms, the statement trees do not need to be transformed, e.g. a complete binary tree.

#### 4.4.1 Decomposing the Whole AST

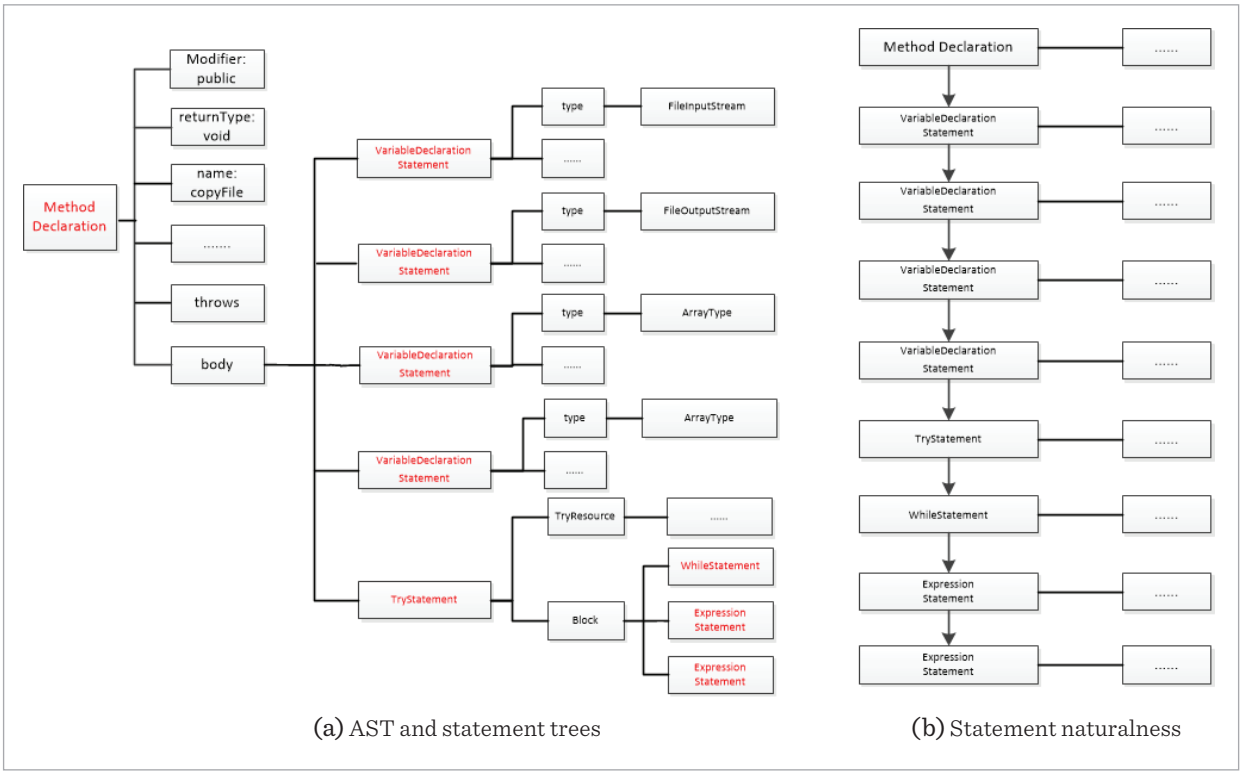
Many methods for the decomposition of ASTs into subtrees without overlapping exist. Inspired by the algorithm mentioned in [34], we split the AST by the granularity of natural statements. The splitting algorithm is detailed on [23], which we apply to the identification of code clone. Readers are advised to refer to it. Figure 4 shows the decomposition process from an AST to small statement trees. Figure 4(a) depicts top 5 layers of the Java AST for simplicity, and Figure 4(b) shows the actual execution order of statement trees.

#### 4.4.2 Encoding the Statement Trees

All the complex encoding algorithms mentioned before might change the structure of ASTs, or enlarge the tree size, which will affect the accuracy of generated comments. Therefore, we decide to encode the statement subtrees with a classical pre-order traversal algorithm.

Before encoding the subtrees, all the internal nodes have to be vectorized according to the generated vocabulary, which is also a challenge. In the field of NLP, people often limit vocabulary to common words in the

**Figure 4**
The decomposition of an AST.




(a) AST and statement trees (b) Statement naturalness

corpus, e.g. 50000 words. The unusual words are indicated by <UNK>. It takes effect to apply such a strategy as words out of vocabulary are rare in NLP. However, it is inappropriate to generate the vocabulary as the source codes are different from the natural languages [11]. The source code contains many user-defined identifiers, each of which might only appears once. If we take a regular vocabulary size for code, a mass of unknown tokens in the form of <UNK> will arise. If we attempt to reduce <UNK> tokens during transformation, the vocabulary size will increase heavily. In order to balance the vocabulary size and <UNK> appearances, we split the internal nodes of the AST into several tokens to generate the vocabulary. For example, FileInputStream -> File, Input, Stream. The total number of unique tokens in the corpus are reduced from more than 428000 to 42873.

## 4.5 Proposed Algorithm

**Input:** A series of source code tokens.

**Lexical Attention:** Determines attention weights using token lexical similarity.

**Structural Attention:** Calculates attention weights from code syntax.

**Functions:**

- lexical_attention (query, key, value): Weighs lexical attention.
- structural_attention (query, key, value): Weighs structural attention.

The lexical_attention and structural_attention functions may be implemented using several methodologies, including scaled dot-product attention, additive attention, or dot product attention.

- combine_attention (lexical_weights, structural_weights): Weighs lexical and structural attention.

The combine_attention function facilitates the integration of lexical and structural weights using various methods, such as weighted sum and concatenation.

- generate_comment (context_vector): Creates context vector-based comment.

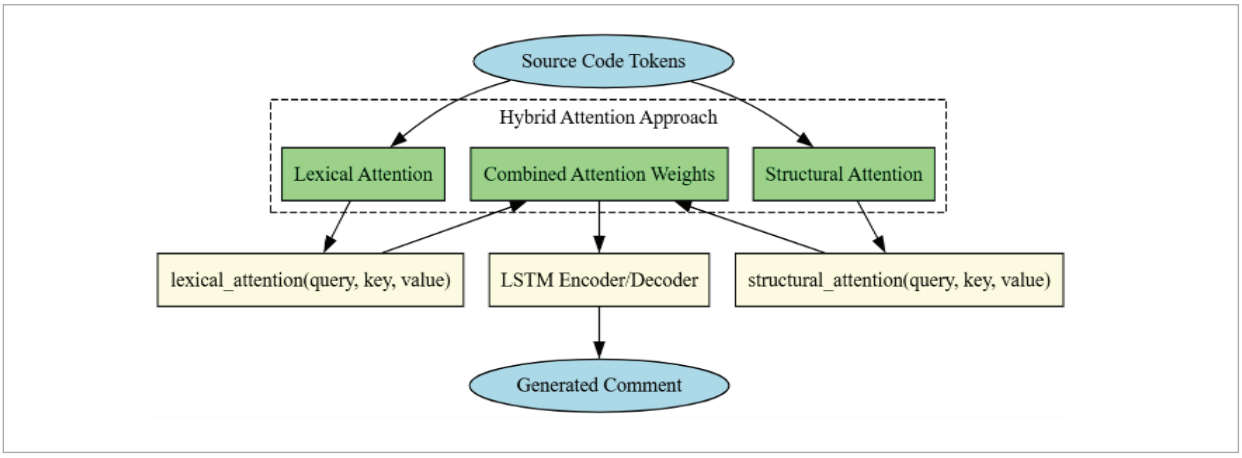
The generate_comment function may use a language model or an alternative generation method to produce the final remark.

**Algorithm steps:**

1. Get lexical and structural attention going.
2. Initialize the weights function for lexical and structural weights, respectively.
3. Compute the attention that token in the input sequence will get both lexically and structurally
4. Combine focus combined weights
5. Create a background vector
6. Create a remark by passing the context vector

The generate_comment function may use a language model or an alternative generation method to produce the final remark. Figure 5 presents the workflow diagram for the proposed algorithm.

**Figure 5**
Workflow diagram for the proposed algorithm.



The proposed method is quite good at handling complex control flow topologies. Conventional methods might find difficult to grasp sophisticated control flow systems, which results in erroneous claims. The hybrid attention approach efficiently captures the connections across many code blocks and contexts thereby guaranteeing that the produced comments fairly represent the logic of the code. At-ComGen may precisely provide a comment clarifying the purpose of every loop and the circumstances under which it runs. This remarks about a nested loop with many criteria.

The proposed approach helps one to understand semantic relationships among variables. When using indirect or complicated expression, it might be challenging to determine the semantic links among the variables in a code fragment.

Using the proposed method in large-scale codes might provide useful annotations. Creating useful comments for big and complicated codebases might be a tedious and prone to mistakes effort. While efficiently controlling big codebases, the hybrid attention approach produces annotations that faithfully represent the intended functionality of the code. At-ComGen can examine a large collection of hundreds of functions and provide succinct but clear comments for each one. This helps developers to understand and apply the library.

### 4.6 Loss Function

In our model, the loss function is defined as the minimized cross-entropy, which is described as follows [14]:

$$H(y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{i}\log p(y_j^{(i)}) \tag{13}$$

where N is the number of training samples, and l is the length of each target sequence. $y_j^{(i)}$ indicates the j-th word in the i-th instance. The equation describes how much the predicted probability diverges from the ground truth. We optimize Equation (11) by gradient descent algorithm.

### 4.7 Evaluation Metrics

Generated comments should accurately and briefly reflect the meanings of the source codes. In this paper, we apply both automatic and manual approaches to evaluate comment quality with previous models.

#### 4.7.1 Automatic Evaluation

In our model, we use BLEU [17] score and METEOR [20] to measure the accuracy of machines comments.

BLEU measures the average n-gram precision on a set of reference sentences, which is popular in machine translation [2, 4, 28].

BLEU is computed as follows:

$$BLEU = BP \bullet \exp(\sum_{n=1}^{N} \omega_n \log p_n) \tag{14}$$

where $p_n$ is the precision of n-grams, that is, the ratio of length $n$ subsequences in the candidate that are also in the reference. We set $n = 4$, which is a common practice for code comment generation [14, 29, 35]. BP is a brevity penalty, which is computed as follows:

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-\frac{r}{c})} & if\ c \le r \end{cases} \tag{15}$$

where $c$ is the length of the candidate translation and $r$ is the effective reference sequence length. We regard the generated comment as a candidate and the Javadoc comment as a reference.

METEOR score is recall-oriented and wildly used for machine translation and code summarization. It measures how well the model captures content from the references by calculating sentence-level similarity scores. The METEOR is computed as follows:

$$METEOR = (1 - Pen)F_{mean} \tag{16}$$

#### 4.7.2 Manual Evaluation

As the automatic test does not always agree with experimental result [22], especially for the code comments described in natural languages, we decide to perform a manual verification for the similarity of generated comments from our model and the ground truth. Inspired by the methods [35], we invite 3 participants to evaluate generated comments respectively. They are doctors in computer science with 3+ years Java experience. Their rating scores are given according to the similarity criteria between artificial comments and machine ones, which are shown in Table 1.

**Table 1**
Criterion of similarity.

| Score | Explanation |
|---|---|
| 0 | No similarity between two comments or the generated comments are meaningless. |
| 1 | Two comments share some similar tokens, but they are not semantically similar. |
| 2 | Two comments have some similar information, but each of them contains some information which is not involved by the other. |
| 3 | The two comments are very similar in semantic, but their means are not the same. |
| 4 | The two comments are identical in meaning, or the ground truth are more difficult to understand. |

Participants are asked to score the comments between 0 to 4 according to the criteria listed in Table 1. During the rating process, the two comments for the same code snippet are shuffled and listed randomly.

#### 4.7.3 Time Complexity Analysis

To calculate time complexity, we use source code length (N) for input sequence, dot product or scaled dot product for attention mechanism, Recurrent neural network design for LSTM encode and sequence to sequence based model for comment generation.

**Time complexity of Attention Mechanism:**

The Dot-product or scaled dot-product attention requires $O(N^2)$ matrix multiplication.

**Time complexity of LSTM Encoder:**

Time complexity of each LSTM cell is $O(N)$. The complexity of an N-length sequence is $O(N^2)$.

**Time complexity of Comment Generation:**

If the encoder and decoder share an architecture, the time complexity is also $O(N^2)$.

The hybrid attention technique for source code comment creation has a temporal complexity of $O(N^2)$ when combining these components.

A hybrid attention technique for source code comment creation has quadratic time complexity relative to input sequence length ($N$). This indicates that computing time grows proportionately with source code length.

## 5. Experiments

We compare our model At-ComGen with several baseline methods on a dataset composed of GitHub data [7, 8].

### 5.1 Baselines

At-ComGen is compared with different state-of-the-art models, including CODE-NN [18], a generic Seq2Seq model, and DeepCom [14], all of which are state-of-art code summarization models.

The famous generative model CODE-NN exploits a common LSTM with an attention algorithm to produce comments by integrating token embeddings instead of making language models.

A second baseline is a classical encoder-decoder model based on Seq2Seq model. We implement a code summarization model built on a Seq2Seq framework, which takes a sequence of tokens from the given code snippet as input and output a comment described in English.

A third baseline is DeepCom, which generates code comments from the given code structure. Based on a LSTM encoder-decoder framework, DeepCom extracts the code semantic by traversing the corresponding AST. The authors proposed a novel traversal algorithm SBT, with which no information are lost in the process of AST traversing.

### 5.2 Dataset Description

Due to the challenge of obtaining source code with exemplary comments, researchers often compile their own datasets by aggregating data from several source code repositories, including GitHub and Stack Overflow, for comparative studies. We gathered an extensive corpus of Java methods from GitHub projects established between 2017 and 2018. We download repositories with above 10 stars to exclude unqualified code. Our dataset undergoes preprocessing with the following processes.

**Table 2**
Statistics of Java dataset.

| Methods | Words | Uniq Words | Training Set | Validation Set | Testing Set |
|---|---|---|---|---|---|
| 588108 | 44378497 | 13779297 | 468108 | 60000 | 60000 |

**Table 3**
Statistics of code lengths.

| Average | <100 | <150 | <200 |
|---|---|---|---|
| 99.94 | 68.63% | 82.06% | 89.00% |

**Table 4**
Statistics of comments lengths.

| Average | <20 | <30 | <50 |
|---|---|---|---|
| 8.86 | 75.50% | 86.79% | 95.45% |

We extract Java functions accompanied by Javadocs from 11,034 projects on GitHub. The first statement in each Javadoc serves as the definitive remark, often outlining the method's functionality in accordance with Java guidelines [25]. Java methods without comments or containing comments of less than three words are excluded.

We eliminate Java methods containing comments not written in English, since our model cannot generate comments in other languages. We further omit auto-generated codes, including setter, getter, and test methods.

We have successfully acquired 588,108 pairs. Table 2 illustrates that our Java dataset comprises around 580,000 functions with annotated data labels (code comments), categorized into training set, validation set, and testing set. Table 3 delineates the specifics of method lengths, whilst Table 4 elucidates the particulars of comment length. The data indicate that over 95% of comments include less than 50 tokens, whereas more than 89% of programs consist of fewer than 200 tokens. Consistent with prior trials, 80% of pairings are randomly designated for training, 10% for validation, and 10% for testing.

## 5.3 Experimental Setting

We use javalang to produce ASTs from given code snippets. According to the statistics described in Table 3 and Table 4, the maximum code length and AST token sequence are set to 250 and 500 respectively. Longer token sequences are reduced to meet the maximum length. For word-embedding, we transform the words into vectors with word2vec tools and set the embedding size to be 128.

Based on an attentional Seq2Seq framework, At-ComGen implements both lexical and syntactical encoders by two independent LSTMs with 128 dimensions of the hidden states. The decoder is also a single LSTM following a hybrid attention layer. In the training process, the model is updated with SGD. We use Adam optimizer with a learning rate 0.5 during training and use dropout with 0.5 to prevent over-fitting. It takes more than 105 hours to train the model with an epoch number 50.

**Table 5**
Evaluation results on Java methods.

| Models | BLEU | METEOR | ROUGE-L |
|---|---|---|---|
| CODE-NN | 27.89% | 13.10% | 34.83% |
| Seq2Seq | 34.36% | 20.98% | 48.21% |
| DeepCom | 38.17% | 22.19% | 47.48% |
| At-ComGen | 40.19% | 23.47% | 51.23% |

## 5.4 Automatic Evaluation

For automatic tests, BLEU, METEOR and ROUGE-L are used to evaluate the quality of generated comments respectively. Table 5 shows the comparison of At-ComGen and CODE-NN, attentional Seq2Seq, DeepCom on our dataset.

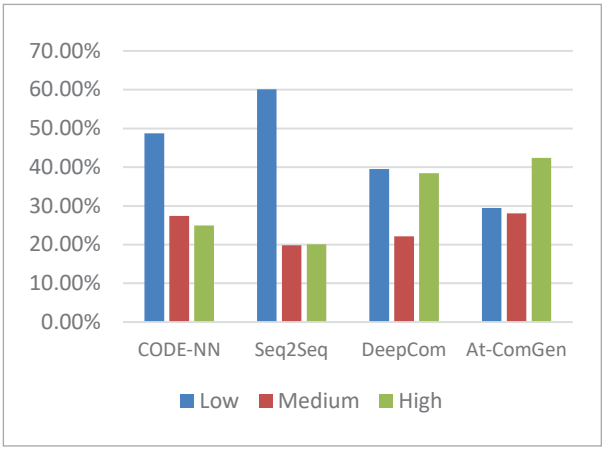
We see that all the metrics of CODE-NN are relatively poor. It is because CODE-NN cannot learn code semantics when it produces comments from code tokens directly. The performance of Seq2Seq is not ideal, as it ignores both the code structure and contributions of key words. DeepCom improves machine comments by the introduction of AST traversing algorithm. In the process of code encoding, At-ComGen employs two independent LSTM to extract both lexical and syntactical information. In the process of code decoding, At-ComGen introduces BERT technology to promote the expression of generated comments. At-ComGen has become the new baseline model due to its performance.

### 5.5 Manual Evaluation

We randomly pick up 100 pairs of code snippets with comments from the testing set. Each model generates independent code comments from these 100 samples. The ground-truth and generated comments are shuffled before the tests.

Three experts evaluate the comment pairs independently and the final scores are calculated by average. Figure 6. shows the results of our manual survey. We regard a score [0,1.5] as low quality, a score (1.5,2.5] as medium quality, and a score (2.5,4] as high quality. From the picture we conclude that the proportion of high-quality comments generated by At-ComGen outperforms the others again. The performance of CODE-NN and Attentional Seq2Seq are similar due to the extraction of code tokens. The quality of comments generated by DeepCom is significantly superior to CODE-NN and Attentional Seq2Seq as it employs structural information hidden in the source code. Results of manual evaluation are close to those of automated evaluation, although they are slightly different.

**Figure 6**
Results of manual evaluations.




## 6. Discussion

We take a discussion on the performance of At-ComGen and other state-of-the-art models. We attempt to analyze components which affect the quality of generated comments via further experiments.

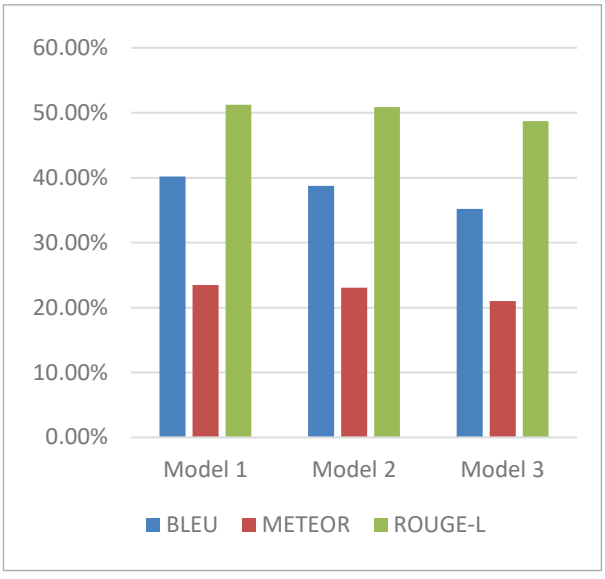
### 6.1 Investigating the Impact of Word Embedding

Around 2018, Transformer and BERT technology have made breakthroughs in NLP tasks such as text processing, text reading and sentiment classification. Experts attempt to use BERT in the process of word embedding. We are inspired to encode words with BERT technology in the process of building At-ComGen.

On the basis of At-ComGen, three models for experiments are built according to different word embedding technologies. Model 1 is the basic version of At-ComGen, which extracts lexical information from source code with Word2vec and generates code comments with "BERT-Base, Uncased". Model 2 uses Word2vec to encode words for source code words, AST nodes and code comments, respectively. Different corpus is used for source code and comments. In Model 3, "BERT-Base, Uncased" is used in the process of word encoding for source code words, AST nodes and code comments. The experimental results for three models on the same Java test set are shown in Figure 7.

The quality of code comments generated by model 1 outperforms model 2 and model 3. The word vectors cannot be well expressed with relatively outdated technology Word2vec, in model 2. Surprisingly,

**Figure 7**
Results of comment quality on different word embedding technologies.



Model 3, in which BERT Pre-trained technology is fully utilized during encoding and decoding, falls behind the other models. We believe that it is due to rare words, especially AST internal nodes, such as InfixExpression, SimpleName. These compound words are very uncommon in human languages, which cannot be embedded with BERT pre-trained model. While the corpus in model 2 is made according to all the words generated before word vectorization. Rare words are easily encoded by the corpus above generated by Word2vec. Most of the tokens appeared in code comments are natural language words, which are easily encoded with BERT pre-trained model.

Currently, it is difficult for At-ComGen to train a BERT model with its own corpus due to hardware limitations. We hope that in the future a specialized BERT model can be trained for source code embedding, which may greatly improve the machine comments.

### 6.2 Investigating the Impact of the AST Splitting

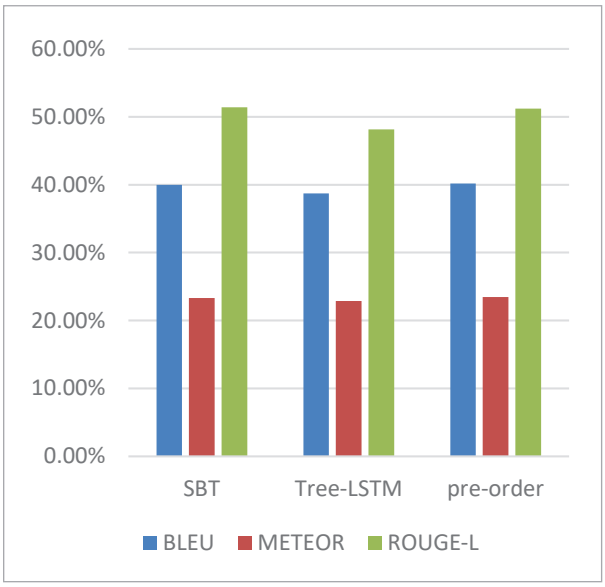
The traversing of the generated AST plays a significant role in the process of code summarization. To compare the results, the original AST is decomposed with three strategies by the splitting granularity. **Model 1:** AST is treated as a special statement tree, which is encoded directly via a classical pre-order algorithm without any splitting.

**Model 2:** At-ComGen is used to generate comments.

**Model 3:** AST is decomposed according to blocks (compound statements including multiple statements within the same brace pairs). For example, trees rooted by TryStatement are not divided into subtrees in Model 3. After decomposition, the following encoding processes are the same as those in At-ComGen. Figure 8. shows the performance of generated comments.

It is shown that complicated traversing algorithms cannot improve the comment quality significantly. The structural encoder based on Tree-LSTM actually reduces comment quality slightly. We believe that the depth of the complex AST declines model 2. In order to update Tree-LSTM parameters successfully, encoder has to reshape the AST to a common binary tree, which might increase the tree depth. As the performance of model 3 is similar to model 1, the pre-order algorithm is applied in At-ComGen for simplicity.

**Figure 8**
Results of comment quality on different AST splitting.




### 6.3 Investigating the Impact of Code Length and Comment Length

In this chapter, we investigate the performance of generated java comments according to source code lengths and comments lengths.

In java, "statement" is the minimum division to keep code function. In this chapter, we study the impact of statements on generated comments. Test data are grouped carefully according to code lengths. Automated tests are applied to evaluate the performance of code comments generated from different models. Figure 9 presents the BLUE-4 scores of 3 baseline models and At-ComGen according to source code lengths*, and Figure 10 presents the METEOR scores of 4 models under the same conditions.

We conclude from Figures 9-10 that both BLUE and METEOR Curves fluctuate as the code lengths vary, while the code length does not have a significant impact on the generated comment performance in a certain range. In most cases, both BLEU and METEOR values from At-ComGen outperform the other baseline models. The machine comments generated from Seq2Seq and CODE-NN, both of which ignore the code structure, fall behind DeepCom and At-ComGen obviously. For example, in the case of a function of 5 lines, the comment BLEU value of

**Figure 9**
Comment BLUE values based on code lengths (*Code length is calculated by counting the number of tokens)

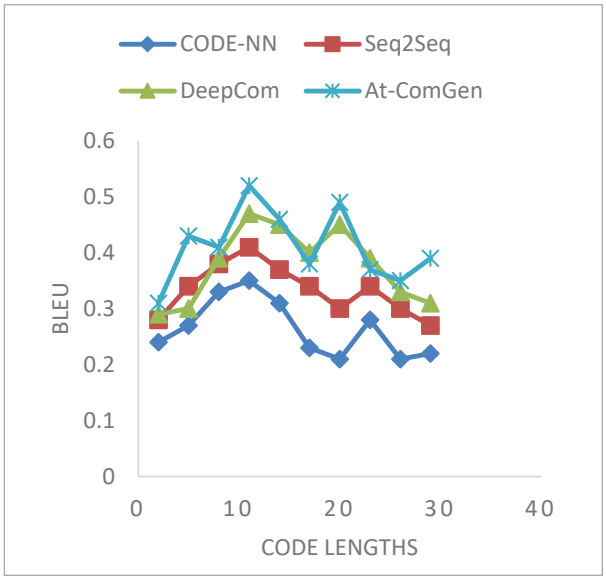



**Figure 10**
Comment METEOR values based on code lengths (Code length is calculated by counting the number of tokens)

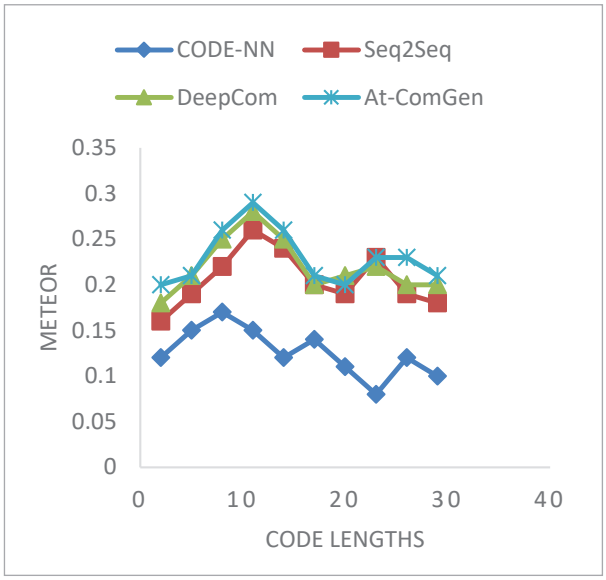



**Figure 11**
Comment BLEU values based on Reference lengths.

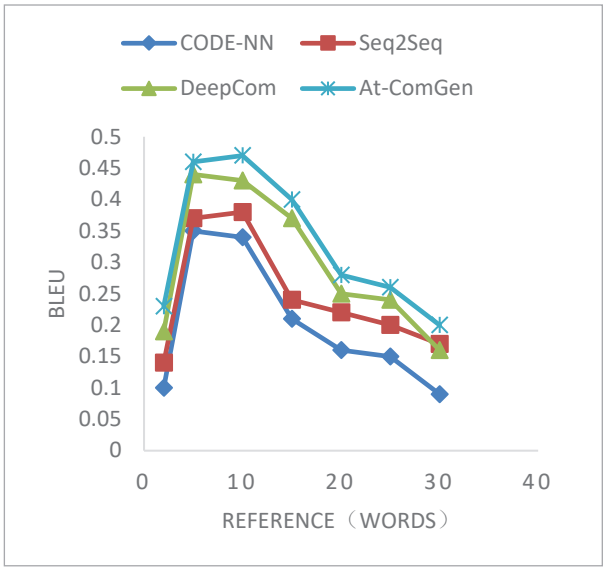
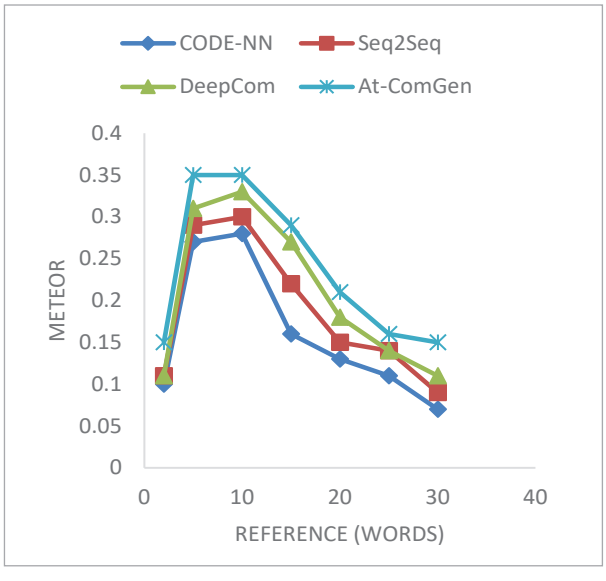



**Figure 12**
Comment METEOR values based on Reference lengths.




At-ComGen is 59.3%, 36.4%, 43.3%, and 13.1% higher than the other baseline models, respectively.

Generally, the generated comment length and the reference length have a positive correlation. We investigate the impact of reference lengths to the performance of machine comments by ablation tests. Automatic evaluations are applied to the generated comments from 4 models according to reference lengths. Figure 11 indicates the trend of BLEU values according to the reference lengths. Figure 12 shows the trend of METEOR values under the same conditions.

We conclude from Figures 11-12. that the reference length has a significant impact on generated comments. On the whole, both the BLUE and METEOR curves of all the models decrease when the lengths of reference comments increase. We guess it is because long comments detail on code functions due to the rich expressive power in natural languages. Hence, some human evaluations are held as a supplement in ablation experiments.

Although the model performance decreases with the increasing of comment length, the comment quality of At-ComGen outperforms the other models in most cases. The quality of machine comments reaches its peak when the reference length is between 5 and 10 words. For example, when the reference length is 10, the BLEU values of At-ComGen are 38.2%, 23.7%, 9.3%, and 4.4% higher than the other baseline models, respectively.

## 6.4 Cases Investigation

In this chapter, some typical cases are introduced to investigate the code comments generated by At-ComGen, CODE-NN, and DeepCom respectively according to the same input functions.

The Java function in Case 1 converts the time from String format into Date format (Figure 13). The machine comment generated from At-ComGen is identical to the reference. However, machine comments generated by CODE-NN and DeepCom are ambiguous, which are heavily influenced by non-key words such as simpleDateFormat and Exception during encoding.

**Figure 13**

Case 1: To convert the time from String format into Date format via CODE-NN, DeepCom and At-ComGen.

```
public Date stringToDate(String time)
{
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
        Date dateTime = null;
        try {
                dateTime = simpleDateFormat.parse(time);
                return dateTime;
        } catch (ParseException e)
                {
                      e.printStackTrace();
                      return null;
                }
}
```

| | |
|---|---|
| Reference: | Convert String format to Date. |
| CODE-NN: | Construct simple date format to date time. |
| DeepCom: | Parse string to date with parse exception. |
| At-ComGen: | Convert string format to date. |

The Java function in Case 2 is designed to extract the information from an HTML page (Figure 14). The machine comment generated by At-ComGen contains more information than the reference. The comment generated by CODE-NN has repetitive words, and the machine comment from DeepCom is also ambiguous. Hidden features in the source code are carefully extracted by the specialized encoder and decoder in At-ComGen.

**Figure 14**

Case 2: To extract the information from an HTML page via CODE-NN, DeepCom and At-ComGen.

```
public String extract(String html, CETR.Parameters parameters)
{
        html = clearText(html);
        List<String> rows = extractRows(html);
        List<Integer> selectedRowIds = selectRows(rows, parameters);
        StringBuilder sb = new StringBuilder(html.length());

        for(Integer rowId : selectedRowIds) {
            String row = rows.get(rowId);
            row = StringCleaner.removeExtraSpaces(HTMLParser.extractText(row));
            if(row.isEmpty()) {
                continue;
            }
            sb.append(row).append(" ");
        }
        return sb.toString().trim();
}
```

| | |
|---|---|
| Reference: | Extracts the main content for an HTML page. |
| CODE-NN: | Extract rows rows html. |
| DeepCom: | Extract html to string builder after clear text. |
| At-ComGen: | Extract the string content from html page after clear text. |

It is shown in Case 3 that the expressive diversity of natural languages might lead to low matching between generated comments and reference comments (Figure 15). The Java function in Case 3 is to check the existence of a given item. Compared to the

**Figure 15**

Case 3: To check the expressive diversity of natural languages via CODE-NN, DeepCom and At-ComGen.

```
public boolean contain(int key, List<String> keyList)
{
       if (keyList.contains(key+""))
             return true;
       else return false;
}
```

| | |
|---|---|
| Reference: | Is the key in the keylist? |
| CODE-NN: | Return true if. |
| DeepCom: | Return true if key list contains key. |
| At-ComGen: | Checks whether the given key is contained within the string list. |

reference, the generated comments from DeepCom and At-ComGen are more expressive and accurate. However, the automatic evaluation fails to give an accurate similarity due to the ground truth. It is the reason why we involve manual inspection in the evaluation phase.

## 7. Conclusion and Future Work

We have presented a hybrid attentional deep learning model for the generation of source code comments. In order to retain both the lexical and structural information, our model employs two independent LSTM encoders in the process of code representation. The lexical encoder extracts the tokens, words and vectorizes them into a unified high dimensional space, while the structural encoder maps the generated AST to a vector with a specific traversal algorithm. Compared to other baseline models, At-ComGen has following advantages: (1) The hybrid attention has intensified the key words and statements during code encoding. (2) The special encoding of statement nodes in AST has greatly reduced the size of generated corpus and the occurrences of "UNK". (3) The introduction of BERT in the encoder obviously improves the expressiveness of the output comments.

There are many promising directions for further study such as intelligent code search, code clone and other code translations. We plan to extend our model to solve these problems. The future version of our model plans to generate new corpus from both source code and syntax tree, and improve the performance of generated comments with self-trained BERT models.

The proposed model's efficiency depends on training data amount and quality. Data that is incomplete or distorted may not be reliable. The large codebases of hybrid attention models make them computationally expensive. Hybrid attention models may struggle to generalize to code from different domains or languages due to conventions and semantics.

We want to explore potential methods for enhancing the precision and relevance of comment creation via the use of data from vast code repositories in the near future. We can enhance code and comments using semantic analysis and word embeddings. Developing a way to automate the reorganization of code comments would enhance their uniformity and precision. We must prioritize code comments and organization, since they will facilitate our ability to anticipate necessary revisions. Users of the change comments section may choose from three unique kinds of remarks: succinct, informative, and detailed. The intended use of the code for both internal and external purposes contextualizes comments, altering their style based on the situation. Optimization strategies may enhance the computational efficiency of hybrid attention, particularly in large codebases.

### Data Availability Statement

The dataset used in this paper has been gathered from https://github.com/xing-hu/EMSE-DeepCom and https://github.com/xjha2/ByteGen

### Declaration of competing interest

The authors declare no conflict of interest.

### Declaration of generative AI in scientific writing

The author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

## References

1. Allamanis, M., Peng, H., Sutton, C. A Convolutional Attention Network for Extreme Summarization of Source Code. Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA, 2016, 2091-2100.
2. Bahdanau, D., Cho, K., Bengio, Y. Neural Machine Translation by Jointly Learning to Align and Translate. ArXiv, 2014, vol. 1409.
3. Chen, Q., Xia, X., Hu, H., Lo, D., Li, S. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. ACM Transactions on Software Engineering and Methodology, 2021, 30(2), Article 25. https://doi.org/10.1145/3434280
4. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. Learning Phrase Representations Using RNN Encoder-Decoder

for Statistical Machine Translation. Computer Science, 2014. https://doi.org/10.3115/v1/D14-1179

5. Devlin, J., Chang, M. W., Lee, K., Toutanova, K. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint, arXiv:1810.04805, 2018. Available: http://arxiv.org/abs/1810.04805
6. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2020. https://doi.org/10.18653/v1/2020.findings-emnlp.139
7. GitHub Repository: ByteGen. https://github.com/xjha2/ByteGen
8. GitHub Repository: DeepCom. https://github.com/xing-hu/EMSE-DeepCom
9. Gros, D., Sezhiyan, H., Devanbu, P., Yu, Z. Code to Comment "Translation": Data, Metrics, Baselining and Evaluation. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, 746-757. https://doi.org/10.1145/3324884.3416546
10. Haque, S., LeClair, A., Wu, L., McMillan, C. Improved Automatic Summarization of Subroutines via Attention to File Context. Proceedings of the 17th International Conference on Mining Software Repositories, 2020, 300-310. https://doi.org/10.1145/3379597.3387449
11. Hellendoorn, V.J., Devanbu, P. Are Deep Neural Networks the Best Choice for Modeling Source Code? 2017 11th Joint Meeting on Foundations of Software Engineering, 2017. https://doi.org/10.1145/3106237.3106290
12. Hu, X., Gao, Z., Xia, X., Lo, D., Yang, X. Automating User Notice Generation for Smart Contract Functions. Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, 5-17. https://doi.org/10.1109/ASE51524.2021.9678552
13. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z. Deep Code Comment Generation. Proceedings of the 26th Conference on Program Comprehension (ICPC), Association for Computing Machinery, New York, NY, USA, 2018, 200-210. https://doi.org/10.1145/3196321.3196334
14. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. Empirical Software Engineering, 2020, 25, 2179-2217. https://doi.org/10.1007/s10664-019-09730-9
15. Huang, Y., Hu, X., Jia, N., Chen, X., Luo, X., Zheng, Z. CommtPst: Deep Learning Source Code for Commenting Positions Prediction. Journal of Systems and Software, 2020, 170, 110754. https://doi.org/10.1016/j.jss.2020.110754
16. Huang, Y., Hu, X., Jia, N., Chen, X., Xiong, Y., Zheng, Z. Learning Code Context Information to Predict Comment Locations. IEEE Transactions on Reliability, 2020, 69(1), 88-105. https://doi.org/10.1109/TR.2019.2931725
17. Huang, Y., Huang, S., Chen, H., Chen, X., Zheng, Z., Luo, X., Jia, N., Hu, X., Zhou, X. Towards Automatically Generating Block Comments for Code Snippets. Information and Software Technology, 2020, 127, 106373. https://doi.org/10.1016/j.infsof.2020.106373
18. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L. Summarizing Source Code Using a Neural Attention Model. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, 2073-2083. https://doi.org/10.18653/v1/P16-1195
19. Kostić, M., Batanović, V., Nikolić, B. Monolingual, Multilingual and Cross-Lingual Code Comment Classification. Engineering Applications of Artificial Intelligence, 2023, 124, Article 106485. https://doi.org/10.1016/j.engappai.2023.106485
20. LeClair, A., Haque, S., Wu, L., McMillan, C. Improved Code Summarization via a Graph Neural Network. Proceedings of the 28th International Conference on Program Comprehension (ICPC'20), ACM, 2020, 184-195. https://doi.org/10.1145/3387904.3389268
21. Li, B., Yan, M., Xia, X., Hu, X., Li, G., Lo, D. Deep Commenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual, 2020, 1571-1575. https://doi.org/10.1145/3368089.3417926
22. Li, Z., Wu, Y., Peng, B., Chen, X., Sun, Z., Liu, Y., Yu, D. SeCNN: A Semantic CNN Parser for Code Comment Generation. Journal of Systems and Software, 2021, 181, 111036. https://doi.org/10.1016/j.jss.2021.111036
23. Meng, Y., Liu, L. A Deep Learning Approach for a Source Code Detection Model Using Self-Attention. Complexity, 2020, 2020, 1-15. https://doi.org/10.1155/2020/5027198
24. Shin, J., Nam, J. A Survey of Automatic Code Generation from Natural Language. Journal of Information Processing Systems, 2021, 17(3), 537-555.

25. Song, X., Haque, S., Wang, X., Yan, J. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. IEEE Access, 2019, 7, 111411-111428. https://doi.org/10.1109/ACCESS.2019.2931579

26. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. L., Vijay-Shanker, K. Towards Automatically Generating Summary Comments for Java Methods. Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), Antwerp, Belgium, 2010. https://doi.org/10.1145/1858996.1859006

27. Sridhara, G., Pollock, L., Vijay-Shanker, K. Automatically Detecting and Describing High-Level Actions Within Methods. Proceedings of the International Conference on Software Engineering (ICSE), 2011. https://doi.org/10.1145/1985793.1985808

28. Sutskever, I., Vinyals, O., Le, Q. V. Sequence to Sequence Learning with Neural Networks. Advances in Neural Information Processing Systems, 2014. Available: http://www.researchgate.net/publication/319770465_Sequence_to_Sequence_Learning_with_Neural_Networks

29. Wan, Y., Zhao, Z., Xu, H., Ying, S., Zhao, J., Wu, J. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. https://doi.org/10.1145/3238147.3238206

30. Wei, H., Ming, L. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, 2017. https://doi.org/10.24963/ijcai.2017/423

31. Wong, E., Liu, T., Tan, L. CloCom: Mining Existing Source Code for Automatic Comment Generation. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, 380-389. https://doi.org/10.1109/SANER.2015.7081848

32. Xu, F. F., Vasilescu, B., Neubig, G. In-IDE Code Generation from Natural Language: Promise and Challenges. ACM Transactions on Software Engineering and Methodology, 2022, 31, Article 29. https://doi.org/10.1145/3487569

33. Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q. Neural Detection of Semantic Code Clones via Tree-Based Convolution. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, IEEE, 2019, 70-80. https://doi.org/10.1109/ICPC.2019.00021

34. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. International Conference on Software Engineering (ICSE), 2019. https://doi.org/10.1109/ICSE.2019.00086

35. Zheng, W., Zhou, H., Li, M., Wu, J. CodeAttention: Translating Source Code to Comments by Exploiting the Code Constructs. Frontiers of Computer Science, 2018. https://doi.org/10.1007/s11704-018-7457-6