Summarizing source code with hierarchical code representation

Ziyi Zhou, Huiqun Yu [*], Guisheng Fan, Zijie Huang, Xingguang Yang

Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China

A R T I C L E I N F O

Keywords:
Code summarization
Program comprehension
Deep learning
Hierarchical attention

A B S T R A C T

Context: Code summarization aims to automatically generate natural language descriptions for code, and has become a rapidly expanding research area. Data-driven code summarization models based on neural networks have proliferated in recent few years.
Objective: Almost all of existing neural models are built upon the granularity of token or AST node. This has several drawbacks: a) Code summarization requires high-level knowledge of code while token representations are limited to provide a global view; b) Such approaches can hardly model the hierarchy of code; c) Long input codes challenge such models to handle long-range dependencies due to the large number of tokens and AST nodes.
Method: To address these issues, we propose a novel framework to utilize hierarchical representation of code to generate better summaries. We consider two levels of code hierarchy: token-level and statement-level. Our framework contains a pair of customized encoder-decoder models for tokens and AST of code respectively. Each of them has a hierarchical encoder that aims to extract both token and statement-level code features, and an attentional decoder with the ability to attend to those different levels of representation during decoding. They are then combined to predict summaries via ensemble learning.
Results: We conduct extensive experiments to evaluate our models on a large Java corpus. The experimental results show that our approach outperforms several state-of-the-art baselines by a substantial margin.
Conclusion: In conclusion, our approach could better learn global information of code and shift attention between important statements during summary generation. With the help of hierarchical attention, the models are able to locate keywords more accurately in a top-down way. Ensemble learning is also proved to be an effective way to benefit from multiple input sources.

1. Introduction

Research has shown that developers spend more than half of their time on program comprehension activities during software development and maintenance [1]. It is essential to provide high-level natural language descriptions of code for developers because such summaries allow them to understand a program more easily without digging into what it does or how it works. High-quality code summaries, in the forms of comments or documents etc., can not only help developers to comprehend programs [2], but also benefit important tasks like code search [3, 21, 22, 24] and code categorization [30]. With the increasing scale and complexity of software, code summaries play a crucial role in the life cycle of software development and maintenance. However, documenting or commenting source code is labor-intensive. In fact, code summaries are often missing, outmoded or even misleading due to the lack of time and energy of developers.

Code summarization aims to automatically generate natural language descriptions for code, and has become a rapidly expanding research area in the past decades. At first, code summarization models are mainly based on heuristics, templates and information retrieval (IR) [2–6]. Inspired by the advance of deep learning in natural language processing, data-driven models that based on neural networks have proliferated in recent few years [8–25]. These models mainly refer to the attentional encoder-decoder (also called Seq2Seq) framework of neural machine translation (NMT) [26–29], where the encoder works as a component for program comprehension and provides a continuous representation of code, which is then feed to the decoder for summary generation. Generally, a code snippet can be treated as a token sequence like a natural language sentence, then recurrent neural network (RNN) [22–24], or convolutional neural network (CNN) [9] could be applied directly.

While simply applying NMT has its limit because source code is

* Corresponding author.
E-mail addresses: zhouziyi@mail.ecust.edu.cn (Z. Zhou), yhq@ecust.edu.cn (H. Yu), yhq@ecust.edu.cn (X. Yang).

https://doi.org/10.1016/j.infsof.2021.106761
Received 21 March 2021; Received in revised form 10 August 2021; Accepted 26 October 2021
Available online 23 November 2021

highly structured and contains a lot of interacting components compared with natural language. Recent studies are beginning to recognize this issue, and thus construct code representation from different sources including abstract syntax tree (AST) [13-15, 17-20] and control flow graph (CFG) [19, 25]. Neural code summarization models vary in their ways to encode structural inputs, e.g., by using recursive neural network (Tree-RNN) [13, 17], graph neural network (GNN) [18–20], or applying Seq2Seq on the flattened node sequence [10, 11, 14].

The observation is that almost all of existing neural models are based on the granularity of token or AST node, or more specifically, obtain token or node feature vectors from the encoder and let the decoder to use these vectors for summary generation. However, this has several drawbacks: a) Intuitively, the task of code summarization requires high-level knowledge of code while token representations are limited to provide a global view; b) Such approaches can hardly model the hierarchy of code, such as token→statement→block→method ("→" denotes "forms"); c) When it comes to long code snippets, there will be a lot of tokens and AST nodes, causing larger size of input, which challenges the model to handle long-range dependencies especially for graph and tree-based models. These can directly influence the quality of generated summaries.

To address these issues, we propose a novel framework to leverage hierarchical representation of code to generate better summaries. In this work, we consider the hierarchy of token→statement→code. This follows the natural behavior of human participants to summarize a code snippet: scan the code firstly to locate key statements such as method header, API invocations and control flows, and then gather more specific information within these statements, e.g. identifiers and strings, to give a brief description. A branch of early studies has designed techniques to simulate this process [2, 4], which select core statements using heuristics and then translate them into natural language via templates. We further extend this idea into the language of deep learning. In our framework, token sequence and AST of input code are split by statement, and then fed into a pair of customized encoder-decoder models. Each encoder-decoder model is equipped with a hierarchical encoder that aims to extract both token (node) and statement-level code features, and an attentional decoder with the ability to attend to those different levels of representation during decoding. Our design of encoder refers to the Hierarchical Attention Network (HAN) for document classification [31], though with some modifications. Finally, these two models are combined to predict summaries via ensemble learning.

We evaluate our models on a filtered large Java corpus from previous studies [10, 16, 17]. The experimental results show that our approach significantly outperforms several state-of-the-art baselines on various metrics. Moreover, we discuss and explain why our approach to hierarchical code representation performs better than traditional token or AST-node-based models.

2. Background and related work

2.1. Neural code summarization

Early approaches to code summarization are mainly based on IR algorithms [3, 5], rules [2, 4, 6], statistical language models [7] and statistical machine translation [41]. While most of recent studies are data-driven and turn to deep learning techniques, utilizing neural networks to represent source codes. We broadly divide them into three categories: single-source models, multi-source models and models based on multi-task learning.

Single-source models utilize only one type of information to generate code summaries. Iyer et al. [8] use an RNN with attention to produce summaries directly from token embeddings. Allamanis et al. [9] design a convolutional attention network to extract features from token sequences to generate function name-like summaries. Structural information from AST is also be considered. Hu et al. [10] customize a traversal method called SBT to serialize ASTs and then apply a standard Seq2Seq model. Alon et al. [11] represent code as a set of AST paths and encode them via RNN, and use attention to select relevant paths while decoding. In [12], the authors propose a retrieval-based neural code summarization approach, where they enhance the neural model with the most similar code snippets retrieved from the training set.

Multi-source models utilize multiple types of information to improve code summarization. This can be done using multiple code encoders or combining different information into one input. Wan et al. [13] employ a Tree-RNN to model AST and an RNN to represent sequential content of code, and then integrate their context vectors to generate summaries. They also exploit reinforcement learning to cope with exposure bias. LeClair et al. [14] and Hu et al. [15] have the same intention: combining the contexts of SBT and token sequence of code to generate summaries. Hu et al. [16] leverage transfer knowledge of API by taking API sequence as an additional input. To better leverage both lexical and syntactical information, Zhou et al. [17] propose a CNN to extract vector representation of AST node and a mechanism to learn an adaptive weight vector for different code representations. Similar to [12], Wei et al. [42] also use retrieved samples to improve this task. Except for the retrieved similar code snippet, they further leverage its AST and paired comment (called exemplar). There are also increased research interests on the application of GNN. Fernandes et al. [18] construct a graph representation of code using AST and lexical information, and apply a GNN to encode it. They combine the sequential information into GNN by initializing a node in the graph with the corresponding output of an RNN. Liu et al. [19] take this idea a step further: combine diverse representations of the source code including AST, CFG and program dependency graph (PDG) into a joint code property graph to produce summaries. Similar to their previous work [14], LeClair et al. [20] use a GNN for the AST as a separate input for token sequence.

Different code-related tasks could share their knowledge to improve them jointly. Chen et al. [21] propose a Bimodal Variational Auto Encoder to project natural language and code into a common semantic space, which could be used for both code retrieval and summarization tasks. In [22], the authors investigate a novel perspective of code annotation for code retrieval, where a code annotation model is trained to generate code summaries that can be leveraged by a code retrieval model to better distinguish relevant codes. Wei et al. [23] design a dual learning framework to train a code summarization and a code generation model simultaneously to exploit the duality of them. Ye et al. [24] also leverage code generation via dual learning, but improve both code summarization and retrieval.

We noticed that a very recent study [25] also attempt to leverage the hierarchical nature of code for this task, by applying vanilla HAN together with multi-inputs and reinforcement learning. However, there are differences between [25] and this work. First of all, we design mechanisms to enable the decoder to attend to the hierarchical representation of code, while their model does not have such dynamic, i.e. their decoder has no attention over inputs. Actually, we found their application of HAN fails to improve a standard Seq2Seq model, as shown in our Secion V (also can be observed from their experimental results). On the other hand, the authors of [25] do not specify how they split the AST for hierarchical input while we elaborate our data processing. Note that different forms of input could greatly affect the performance. Another major difference is that we explore the technique of ensemble learning to benefit from multiple inputs.

2.2. Attentional encoder-decoder framework

Neural code summarization models apply NMT techniques to translate source code to natural language. A basic NMT system follows the attentional encoder-decoder framework. The encoder maps a sequence of input vectors $x = (x_1, \ldots, x_n)$ to a continuous representation $h = (h_1, \ldots, h_n)$. Commonly, an RNN such as long short-term memory (LSTM) [32] and gated recurrent units (GRU) [33] can be used as the encoder:

$$h_t = f_{enc}(x_t, h_{t-1}) \tag{1}$$

where f_{enc} denotes an RNN function. In order to capture structural information of codes, Tree-RNNs or GNNs can also be applied [17–20]. The decoder is a unidirectional RNN which uses h and its previous outputs to predict a desired target sequence $y = (y_1, \dots, y_m)$ one element at a time. The conditional probability of generating next token y_t is

$$\begin{aligned} p(y_t|y_{<t}, x) &= \mathrm{softmax}(g(s_t, c_t)), \\ s_t &= f_{dec}(s_{t-1}, y_{t-1}, c_{t-1}) \end{aligned} \tag{2}$$

where g is a parametric function for probability estimation, f_{dec} is a decoder function, s_t is a decoding state and c_t is a context vector computed by the attention mechanism [28]. Attention aims to select important parts from input sequence by assigning different weights to elements in h at each decoding time step t.

NMT models as well as neural code summarization models are traditionally trained via maximum likelihood estimation (MLE). Namely, the goal is to maximize the objective

$$J = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log p(y_t^i | y_{<t}^i, x^i) \tag{3}$$

where N is the number of training sentence pairs, (x^i, y^i) is the i-th pair and T_i is the length of the i-th target sequence y^i.

3. Proposed approach

We provide details of our approach in this section, including the hierarchical code encoder, attention mechanism and data processing. Fig. 1 shows the overall workflow (during prediction) of our final model. It consists of two independent encoder-decoder models that receive token sequences and AST node sequences split by statement respectively, called summarizer (token) and summarizer (AST). Each of them has a hierarchical encoder to better model code structure and a decoder with the ability to reference the learned different levels of code representation. The final model is an ensemble of them, i.e., their predicted probability of target words at each time step are combined during inference.

3.1. Splitting code and its AST

We split the code and its AST by the granularity of statement and feed statement sequences into both of the summarizers.

It is conventional to write one statement per line while the styles of braces (Java for example) and complex expressions may vary. Another concern is that directly split the code by lines can lose some structural information such as nesting. Therefore, we first cancel the breaks in long statements, add braces to unbraced one-line conditional statements, and then format the code according to Allman style. These can be done using a code formatter. After that, we split the code by lines to obtain statement sequence. Each statement is then split into a token sequence. The identifiers and strings are further split into subtokens according to snack_case and camelCase. A single brace ("{" or "}") is treated as a special statement to indicate the beginning or end of a block in the statement sequence.

Syntactical information has been shown to be helpful to this task [11, 13-15, 17-20], so we also encode ASTs in a hierarchical manner. Splitting an AST on statement level is trickier than a raw code snippet. We traverse the AST in preorder to determine which statement a node should belong to. If a node has a child named *block* (e.g. *WhileStatement* and *ForStatement*) or *body* (e.g. *MethodDeclaration* and *ClassDeclaration*), it will lead a group of nested statements. It is also the root of a statement such as a method header, so its descendants excluding the ones in its *block* or *body* will first be collected as a statement subtree. The (direct) children of its *block* or *body* node are the roots of its following nested statements. For each of these children, if it also contains *block* or *body*, then it is treated like before. Otherwise, it will be included in a new subtree together with its all descendants. Sequence of statement subtrees can be obtained accordingly. Since we use preorder traversal, these subtrees appear in the same order as their corresponding statements in the code. We use preorder node sequence to represent a subtree. Note that a terminal AST node has a value field that is a token somewhere in the code, and could contain rich semantics. Similarly, we further split the token if it is an identifier or string, and then treat the subtokens as the children of this node. The order of these subtokens is kept during traversal. Fig. 2 shows a method and the node sequences of its statements. Algorithm 1 illustrates the pseudocode to get the input of summarizer (AST), where GetTokens will return a sequence of subtokens including a node's type and split value (if exists).

3.2. Hierarchical encoder

The hierarchical encoder consists of a token encoder, a statement encoder and a token-level intra attention layer, as illustrated in Fig. 3. The token sequence $x_i = (x_{i1}, x_{i2}, \dots, x_{il})$ of the i-th statement will first be mapped to a sequence of embedding vectors using a learnable embedding lookup matrix. Then a bi-directional LSTM (biLSTM), i.e., the token encoder, is applied to encode it into a continuous representation $h_i^{tok} = (h_{i1}^{tok}, h_{i2}^{tok}, \dots, h_{il}^{tok})$, where each of its element h_{ij}^{tok} is the concatenation of the forward LSTM state $\overrightarrow{h}_{ij}^{tok}$ and the backward one $\overleftarrow{h}_{ij}^{tok}$. Considering

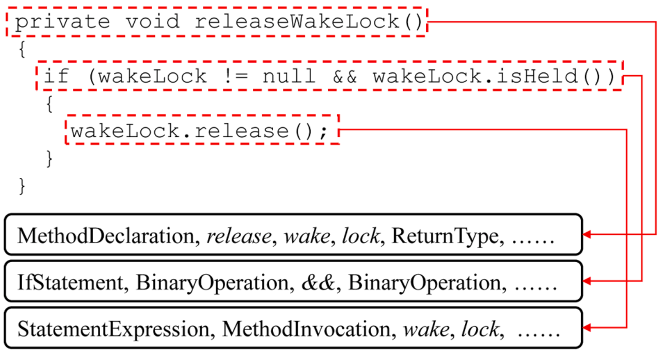

```
private void releaseWakeLock()
{
    if (wakeLock != null && wakeLock.isHeld())
    {
        wakeLock.release();
    }
}
```
MethodDeclaration, *release*, *wake*, *lock*, ReturnType, ……
IfStatement, BinaryOperation, *&&*, BinaryOperation, ……
StatementExpression, MethodInvocation, *wake*, *lock*, ……

Fig. 2. A formatted code snippet with its AST node sequences split by statement. Subtokens within node sequence are in italics.

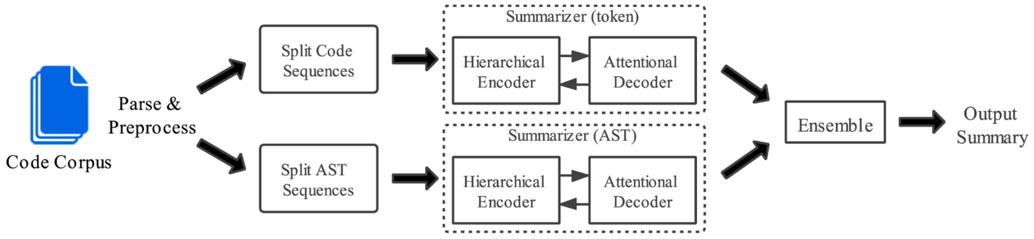


Fig. 1. Overall workflow of our approach.

Algorithm 1.
Constructing input for summarizer (AST).

```
Input: AST with root $R$
Output: Sequence of statement subtrees
1: Initialize subtreeSeq
2: def Traversal(node, subtreeSeq):
3:   Initialize nodeSeq
4:   If node has block or body:
5:     Take block or body apart from its children
6:   For n in PreOrder(node):
7:     nodeSeq.extend(GetTokens(n))
8:   subtreeSeq.append(nodeSeq)
9:   If node has block or body:
10:    For child in its block or body:
11:      Traversal(child, subtreeSeq)
12: Traversal(R, subtreeSeq)
```

that not all tokens contribute equally to the representation of a statement, we use an attention mechanism similar to [31] to extract important parts and aggregate the low-level representation to a statement vector k_i. Since the decoder also has its attention mechanism, we call this attention layer "intra attention" to distinguish them. Specifically,

$$\begin{aligned} u_{ij} &= \tanh\left(W_u h_{ij}^{tok} + b_u\right) \\ \alpha_{ij} &= \frac{\exp\left(u_{ij}^T u_x\right)}{\sum_{j=1}^{l} \exp\left(u_{ij}^T u_x\right)} \\ k_i &= \sum_{j=1}^{l} \alpha_{ij} h_{ij}^{tok} \end{aligned} \tag{4}$$

where W_u and b_u are parameter matrix and bias, α_{ij} is the normalized importance weight of x_{ij}, and u_x is a jointly learned context vector which can be seen as a high-level representation used for token selection. We randomly initialize u_x following [31].

The statement vector k_i is a summary of tokens x_i, but without information from context around the i-th statement, i.e., other statements in the code. So we use another biLSTM, called statement encoder, to encode statement sequence $k = (k_1, k_2, \ldots, k_L)$ into $h^{stat} = (h_1^{stat}, h_2^{stat}, \ldots, h_L^{stat})$. Different from [31], no intra attention is applied to h^{stat} because in our case, it is up to the decoder to extract information from different granularity of code representation for summary generation.

3.3. Attentional decoder

The decoder is a unidirectional LSTM. Its initial state is the backward final state of the statement encoder, i.e., $s_0 = \overleftarrow{h_L}^{stat}$. Actually, $\overleftarrow{h_L}^{stat}$ could only roughly represent the input code. So we make the decoder to attend to both token-level and statement-level representations of each statement, i.e., k_i and h_i^{stat}, at each decoding time step t. Specifically, we use their concatenation $p_i = [k_i; h_i^{stat}]$ as both key and value element of the attention mechanism:

$$\begin{aligned} p_i &= \left[k_i; h_i^{stat}\right] \\ \beta_i &= \frac{\exp\left(s_t^T W_a p_j\right)}{\sum_{j=1}^{L} \exp\left(s_t^T W_a p_j\right)} \\ c_t &= \sum_{i=1}^{L} \beta_i p_i \end{aligned} \tag{5}$$

where W_a is the parameter matrix of the multiplicative attention [28] to improve the interaction between two vectors, c_t is the attentional context vector at step t. Then we use c_t to compute the conditional probability of generating a target token y_t:

$$\begin{aligned} p(y_t | y_{<t}, x) &= \mathrm{softmax}(W_s \tilde{c}_t) \\ \tilde{c}_t &= \tanh(W_c[c_t; s_t] + b_c) \end{aligned} \tag{6}$$

where W_s, W_c and b_c are trainable parameters. The above process is also shown in Fig. 3, and we call (5) decoder attention.

Motivation behind aggregating p_i. Besides the code encoder, the most important part of our task is summary generation, so let the decoder to perform fine-grained information selection from encoding results is intuitive. With the cooperation of intra attention and decoder attention, both levels of encoding result h^{tok} and h^{stat} can be efficiently evaluated by the decoder:

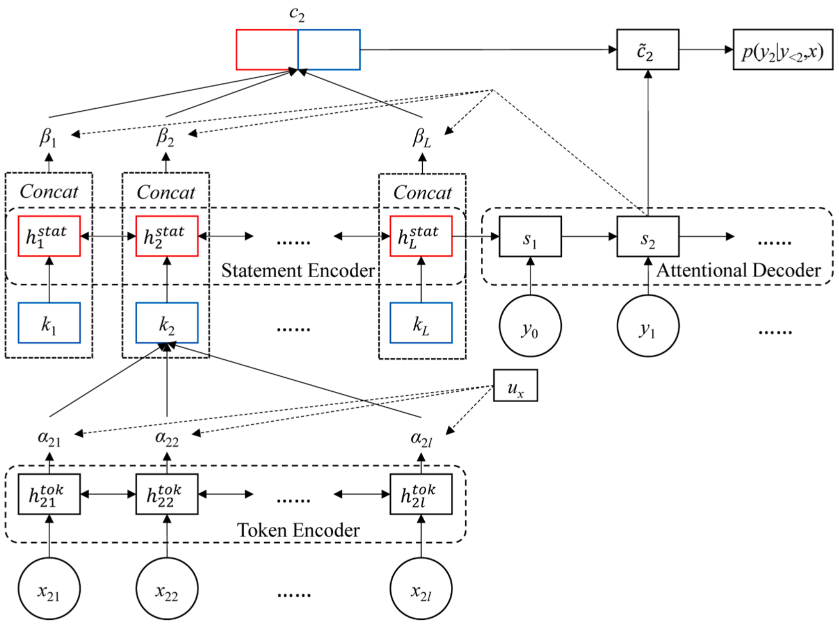


Fig. 3. Hierarchical encoder-decoder architecture.

$$\begin{aligned} c_t &= \sum_{i=1}^{L} \beta_i p_i = \sum_{i=1}^{L} \left[\beta_i k_i ; \beta_i h_i^{stat}\right] \\ &= \left[\sum_{i=1}^{L} \sum_{j=1}^{l} \alpha_{ij} \beta_i h_{ij}^{tok} ; \sum_{i=1}^{L} \beta_i h_i^{stat}\right] \end{aligned} \tag{7}$$

where $\alpha_{ij}\beta_i$ can be considered as a refined normalized weight over all tokens in the code. Despite that the values of $\alpha_{ij}\beta_i$ can be biased due to the variable lengths of the statements, they enable the model to refer to the token representation of code indirectly while also utilizing the higher-level representation through h^{stat}.

Ensemble Decoding. The summarizers share the same topology and are trained separately. In order to combine them to benefit from both input types, we simply average their output probability vectors at each step, and feed the currently chosen target word back to both of them as inputs at next step.

4. Experimental setup

4.1. Dataset

The dataset we use for model evaluation is originally provided by Hu et al. [10, 16] and processed by Zhou et al. [17]. It contains over 513K pairs of Java method and comment collected from GitHub. Zhou et al. have removed the duplicate pairs and carefully filtered the dataset for code summarization task. In particular, they removed a lot of redundant content that are hard to generated by models from the comments such as HTML labels, “TODO: …” and URLs, and also split the identifiers in the comments into subtokens. For better memory efficiency and evaluation, we limit the length of comments to 3 to 30 subtokens and the length of codes to less than 400 (longer inputs are truncated). We use the same data partition as [17] for model training, validating and testing. Javalang[1] is applied to tokenize the codes and parse them to ASTs.

4.2. Training and inference settings

The hidden size of the LSTMs in our model is set to 256. We use 128-dimensional (sub)token embeddings for both encoders and decoders. To efficiently guide the token and node-level encoders, we pretrain them using a pair of vanilla Seq2Seq models. We find this improves both the results and convergence speed of the hierarchical model. Other parameters are initialized using Glorot initialization [34]. We limit the vocabulary sizes of codes, ASTs and comments to 30K, 31K and 25K, respectively.

We optimize the training objective (3) using Adam [35] with initial learning rate of 0.001. The batch size is set to 45 for both of the summarizers. We clip gradient norm by 5, and apply dropout [36] of 0.3 on embedding vectors and the recurrent units. We use early stop to prevent the models from overfitting. Specifically, we stop training when the perplexity[2] recorded on the validation set does not improve within 4 epochs and select the model with best validating BLEU score [37]. Beam search [29] is used during validating and testing, with the beam size and length penalty set to 4 and 0.4. The code for the whole workflow of our model is available on GitHub[3].

4.3. Baselines

We compare our approach with five deep learning based code summarization models from previous work, which take various types of information as their inputs:

DeepCom [10] uses a special traversal method named SBT to linearize AST into node sequence, and then a standard Seq2Seq model is applied to translate the node sequence into natural language summary. Since we also use traversal to derive node sequences, DeepCom is chosen to be a baseline.

code2seq [11] represents code as a set of AST paths and encode them via biLSTM. Then a decoder with attention to select the relevant paths is used for sequence generation. AST paths can be seen as another split method of AST compared to subtrees, so we examine this model in our experiment.

CNN-TreeLSTM is another recent AST-based model described in [17]. It uses a CNN with different widths of kernels to extract information from each AST node, including its type and subtokens. The obtained node feature vectors are then fed to a Tree-LSTM equipped with self-attention to form a tree-level representation. The motivation of this model is close to ours: design different modules to learn different granularity of code features, i.e. lexical and structural.

ast-attendgru [14] is a typical model with multiple inputs, which has one GRU encoder for token sequence of code and another one for its flattened AST, i.e. SBT sequence. During decoding, the attentional contexts of both inputs are fused to predict a target word. Similar combination strategies can be found in [13, 15, 20].

HybridHAN+RL [25] is the most relevant baseline to this work. It applies three standard HAN encoder for code, AST and CFG respectively, and then take the combined context vector as the decoder’s initial state. Furthermore, it uses reinforcement learning for comment generation to alleviate the exposure bias issue.

For comparison, all baselines use training and inference settings similar to ours. In detail, they use the same embedding size and RNN hidden size as ours, and benefit from beam search during prediction. We also enhance DeepCom with a biLSTM encoder. Subtoken splitting is applied to the inputs of baselines (except for SBT sequences because it has its own logic to handle tokens). For HybridHAN+RL, we perform 10 epoches of reinforcement training after it early stopped on MLE objective (3). To be fair, it also takes the well-organized inputs described in Section 3.1. Furthermore, we also consider their approach to the hierarchical attention as variants of our model, and will be compared in ablation study.

4.4. Evaluation metrics

We evaluate the quality of generated summaries using three automatic metrics: BLEU [37], ROUGE-L [38], and METEOR [40]. They are widely-used in NMT and code summarization since they are close to human assessment. All models are evaluated using the same script, and these scores are reported in percentage.

BLEU. Given a generated sentence, BLEU calculates the weighted geometric mean of n-gram precision on reference sentence for different n, and brings brevity penalty to short predictions. It is calculated as

$$\begin{aligned} BP &= \begin{cases} 1 & \text{if} c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if} c \le r \end{cases} \\ BLEU &= BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \end{aligned} \tag{8}$$

where p_n is the co-occurrence rate of length n subsequences between candidate and reference, c is the length of the candidate, r is the effective reference sentence length, and BP refers to brevity penalty. We set $w_n =$ 0.25 and compute BLEU for $N = 1$, 2, 3, and 4. Considering that higher order n-grams may not overlap, we evaluate both BLEU-4 with and without NIST smoothing [39]. The smoothed BLEU-4 is denoted as BLEU-4(s).

ROUGE-L. ROUGE-L computes the similarity between generated summary and reference based on the length of longest common subsequence. Giving a reference summary X of length r and a candidate

1 https://github.com/c2nes/javalang
2 https://en.wikipedia.org/wiki/Perplexity
3 https://github.com/zy-zhou/HACS-release

summary Y of length c, it is computed as

$$R_{lcs} = \frac{\mathrm{LCS}(X,Y)}{r}$$
$$P_{lcs} = \frac{\mathrm{LCS}(X,Y)}{c} \tag{9}$$
$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs}+\beta^2 P_{lcs}}$$

where LCS calculates the length of the longest common subsequence of two sequences. The parameter β is set to 1. One advantage of using LCS is that it does not require consecutive matches but in-sequence matches that reflect sentence level word order as *n*-grams.

METEOR. METEOR is an improvement of BLEU, which takes recall into account and applies synonym matching. Based on the number of mapped unigrams m found between the two sequences, it first calculates unigram precision $P = m / c$ and unigram recall $R = m / r$, and then computes a parameterized harmonic mean of P and R

$$F_{mean} = \frac{P\cdot R}{\alpha\cdot P + (1-\alpha)\cdot R} \tag{10}$$

The sequence of matched unigrams between the two strings is divided into the fewest possible number of chunks. The number of chunks ch and the number of matches m are then used to calculate a penalty for the final score

$$Pen = \gamma\cdot\left(\frac{ch}{m}\right)^{\beta}$$
$$score = (1-Pen)\cdot F_{mean} \tag{11}$$

We use the optimized parameter settings for English target described in [40]: $\alpha = 0.85$, $\beta = 0.2$ and $\gamma = 0.6$. The implementations of BLEU and ROUGE are from NLTK[4].

5. Results and analysis

We present the experimental results and analysis by investigating the following research questions:

RQ1. How does our proposed approach perform compared to the baselines?
RQ2. To what extent can our models handle long-range dependencies within the code?
RQ3. What are the effects of different configurations of our models, i.e. different variants of attention over code representation?
RQ4. How does the proposed approach perform in practice?

5.1. RQ1: overall performance against baselines

Table 1 shows the overall results achieved by our models and the baselines. We call our approach HACS (Hierarchical Attentional Code Summarizer), where HACS-AST, HACS-token and HACS-ensemble denote the AST-based summarizer, token-based summarizer and the ensemble model mentioned in Section 3.

We can see from the table that on most metrics, ast-attendgru is a competitive model among the baselines. This is mainly because it uses both information from token sequence and AST. Handling tokens is crucial for code summarization since many keywords in the summary could also appear in the source code. Feeding token sequence makes it possible to directly copy tokens via attention. We will illustrate how this works in our case in Section 6. Since HybridHAN+RL optimizes BLEU-4 during reinforcement training, it achieves higher BLEU scores. However, it failed to obtain consistent improvements on other metrics against ast-attendgru, despite that extra information from CFG is utilized. In effect, such application of vanilla HAN has drawbacks in decoding process. We will discuss this in Section 5.3. It is obvious that the combination of multiple inputs can improve the quality of generated results, which is why we introduce HACS-ensemble. CNN-TreeLSTM is a baseline with great potential. Although it only takes AST as input, it achieves similar BLEU-1 and BLEU-4 scores compared to ast-attendgru. This suggests that there also contains rich lexical semantics within the syntax tree, so it is reasonable to find effective ways to encode both information. code2seq performs relatively worse than the above baselines. We assume the reason is that code2seq only considers the relationship between paired AST leaves (by encoding the paths) but less global knowledge, and its encoding result is not as rich as the other two. The authors of DeepCom concatenates the type and value of an AST leaf as a token within SBT sequence, so it mainly focuses on structural information of AST. Since it does not build subtoken embeddings, it obtains less lexical features than others and thus gets lower results.

It is clear that either of our models outperforms all baselines on all metrics by a certain gap. The individual summarizers within the ensemble model already show their great capability. More specifically, HACS-AST outperforms the best AST-based baseline, namely CNN-TreeLSTM by 7.5% BLEU-4(s), 3.7% ROUGE-L and 5.2% METEOR. It also gains about 1.6 BLEU-4 points compared to ast-attendgru. This indicates that our approach could better utilize both lexical and structural features in AST. It is not surprising that HACS-token shows even better scores than HACS-AST, because lexical knowledge could guide the generation process more directly, as analyzed before. In essence, hierarchical inputs explicitly reflect the structure within the token sequence, and our model has the ability to extract these different levels of features during decoding. The best scores are achieved by HACS-ensemble, showing that our combining method makes sense, and significantly boost the performance of the individuals. HACS-ensemble outperforms ast-attendgru by 17.2% BLEU-4(s), 6.8% ROUGE-L and 9.3% METEOR, which is a considerable improvement under the similar 2-way input.

5.2. RQ2: effect of input lengths

The quality of generated summaries can vary when altering the lengths of source code, because larger input size challenges the model to capture long-range dependencies. To answer RQ2, we examine the average testing ROUGE-L scores for different models with respect to various code lengths, as shown in Fig. 4.

We can observe that the peak performance of all models achieves at code lengths within 30 to 70 subtokens, and there is a natural downward trend of the scores for longer codes. In particular, the AST-based baselines get very low score at code lengths of 10. A possible reason is that such short codes contain only a few AST nodes or paths so these models have difficulty to obtain useful features from input, while lexical information could be more helpful in this case. At almost all code lengths, our ensemble model keeps leading the baselines by a large gap, and also obviously improves both HACS-AST and HACS-token. This indicates that our approach is more robust for both short and long codes. It is interesting to compare the curves of ast-attendgru and HACS-AST. When code length is within 230, they show close performance, but HACS-AST scores higher when the code becomes longer, namely 240 to 350 subtokens. Thus, we can conclude that the higher overall score of HACS-AST (Table 1) is due to its better performance on the long codes. HACS-token outperforms HACS-AST for most lengths of code, indicating that it also benefits from hierarchical representation. HACS decomposes input code into short snippets and relieves the burden of the model from memorizing long sequences, so that the long-range dependencies within the code could be better modeled through encoders of different levels.

[4] http://www.nltk.org/

Table 1
Overall performance compared to baselines.

Model	BLEU-1/2/3/4				BLEU-4(s)[a]	ROUGE-L	METEOR
DeepCom	28.62	19.99	15.00	11.32	18.18	36.92	17.93
code2seq	32.88	22.72	16.68	12.69	20.29	42.77	20.85
CNN-TreeLSTM	34.95	25.06	18.59	13.82	22.01	45.66	22.11
ast-attendgru	34.94	25.56	18.89	13.79	22.24	46.70	22.51
HybridHAN+RL	34.60	25.22	19.39	14.91	22.71	44.69	21.74
HACS-AST	36.96	27.05	20.36	15.42	23.66	47.33	23.25
HACS-token	37.94	28.24	21.63	16.50	24.80	48.40	23.87
HACS-ensemble	**39.06**	**29.61**	**22.96**	**17.67**	**26.07**	**49.87**	**24.60**

BLEU-4(s) denotes the smoothed BLEU-4.

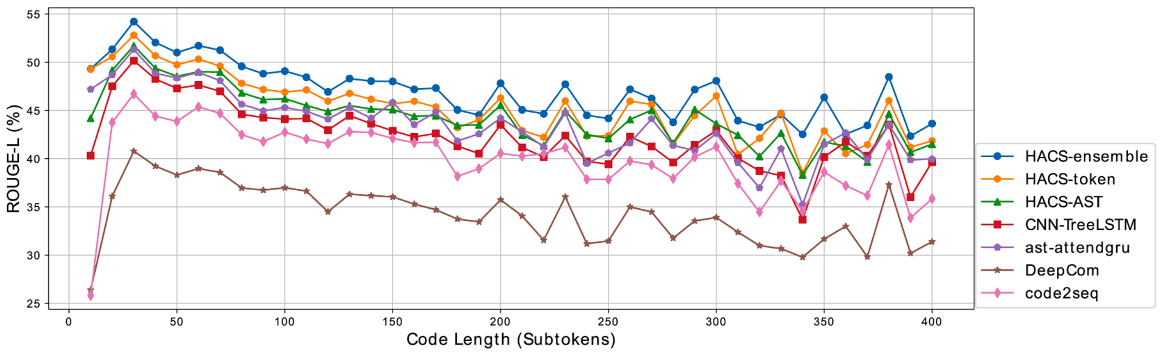


Fig. 4. ROUGE-L scores for different code lengths.

5.3. RQ3: ablation study

We conduct an ablation study to check the effectiveness of our proposed mechanisms. The following variants of our model are examined:

biLSTM-AST & **biLSTM-token** are standard attentional Seq2Seq models, which encode token sequence and AST node sequence respectively. They use biLSTM encoders and attentional LSTM decoders. Their inputs are similar to HACS but are not split by statement.

HAN-AST & **HAN-token** are derived models from HybridHAN+RL, using vanilla HAN (based on biLSTM) as their encoders but without attention for their decoders. To be specific, the top output vector of HAN is used to initialize the decoder. Similarly, they use the inputs described in Section 3.1.

HACS-AST* & **HACS-token*** let the decoders to attend to the higher-level encoding results h^{stat} instead of both token and statement-level, i.e., replacing p_i in (4) using h_i^{stat}. This is a simpler design of the decoder attention.

Table 2 shows the results of these configurations. It is clear that HAN-AST and HAN-token show the worst scores, and seriously degrade the standard Seq2Seq baselines. Specifically, HAN-AST loses 11.1% BLEU-4, 9.2% ROUGE-L and 10.4% METEOR from biLSTM-AST, while HAN-token loses 8.1%, 7.5% and 8.5% of those from biLSTM-token. The reason is that for each of these models, only a single encoding vector is utilized by the decoder, and its ability of information selection and token copy is lost. In other words, the attention weights calculated by HAN are fixed throughout the generation process. As described in Section 1, HAN is originally designed for text classification and does not aim to guide a text generation task, while code summarization relies more on the details and lexical resources in the input than classification. Therefore, simple application of vanilla HAN as in [25] is not practical for its larger model size and worse performance. HACS-AST* and HACS-token* score better than the above two but are still not as competitive as standard Seq2Seqs, despite that they have already equipped with the decoder attention. As seen, HACS-AST* shows slightly lower results than biLSTM-AST on all metrics, and HACS-token* only achieves better BLEU scores. By contrast, our models consistently outperform the others on all metrics, which proves the effectiveness of our approach. This again suggests that utilizing only high-level code representations is not enough for the models to produce good summaries, while the reference to the tokens is preferred.

5.4. RQ4: human evaluation

Although above automatic metrics could evaluate the generated summaries objectively and quickly, they do not always agree with the actual quality of the results. Therefore, we perform human studies to evaluate the results from HACS and baselines. Following [8, 15], we consider two aspects of the generated summaries:

Naturalness. Grammaticality and fluency of the generated comments. It is on a scale between 1 and 5. Specifically, 1 means the result is totally not readable, 3 means the result contains a few grammatical errors or repeated phrases but does not affect understanding, and 5 means there is no grammatical error and the result is as smooth as from human.

Informativeness. The amount of key content carried over from the input code to the generated comments, ignoring fluency of the text). It is also range from 1 to 5. Specifically, 1 means the result could not reflect any functionality of the code snippet and is totally misleading, 3 means the result could cover the main functionality but loses the details, and 5 means the result is able to summarize the code precisely and no worse than the reference.

We invite 5 volunteers with rich Java development experience and excellent English level to rate the generated summaries according to the above two aspects. They are either developers or Ph.D. students of computer science (CS). Their background information is shown in Table 3. We randomly choose 150 code snippets from the test set, and then equally divide them among the volunteers. We make a

Table 2
Results of different variations of our model.

Model	BLEU-1/2/3/4				BLEU-4(s)	ROUGE-L	METEOR
biLSTM-AST	35.32	25.92	19.31	14.36	22.70	46.86	22.94
HAN-AST	32.10	22.42	16.54	12.39	20.18	42.53	20.56
HACS-AST*	35.15	25.32	18.84	14.08	22.34	46.06	22.39
HACS-AST	**36.96**	**27.05**	**20.36**	**15.42**	**23.66**	**47.33**	**23.25**
biLSTM-token	35.85	26.39	19.79	14.73	23.07	47.33	23.16
HAN-token	33.28	23.59	17.57	13.25	21.21	43.79	21.18
HACS-token*	36.15	26.45	19.97	15.08	23.44	47.31	22.97
HACS-token	**37.94**	**28.24**	**21.63**	**16.50**	**24.80**	**48.40**	**23.87**

Table 3
Backgrounds of the participants.

ID	Programming Experience	Java Experience	Occupation
1	9 years	2 years	CS Ph.D. student
2	10 years	5 years	Development Engineer
3	5 years	2 years	Development Engineer
4	6 years	3 years	CS Ph.D. student
5	6 years	5 years	CS Ph.D. student

questionnaire for each participant. In detail, the comments generated by different approaches (the models in Table 1) are evaluated together for each snippet. The ground truth is also presented to the participants for reference, but they do not know which model a certain result is generated from.

The evaluation results are shown in Table 4. As seen from the table, all models are able to generate fluent summaries, and achieve close scores on naturalness. This is mainly due to their same beam search settings. However, the difference in their scores of informativeness is obvious. HACS surpasses all baselines in terms of informativeness, which means that it is able to generate more relevant and accurate summaries than others. Through the feedback from volunteers, we find that when the references contain information outside the given code snippet, the models always fail to generate such contents, e.g., the variable or method names appear in the context of class. We will try to eliminate this limitation by including higher levels of code hierarchy in future work.

6. Discussion

6.1. Case study and visualization

In this section, we show output examples and visualizations of our models to discuss the effectiveness of attending to the hierarchical code representation during decoding. We choose to compare with the outputs of biLSTM-AST / token and HAN-AST / token because the comparison against these models could provide a clearer view of the contribution of our approach. The example codes and corresponding outputs are shown in Table 5. We omit the outputs from HACS-ensemble since they match the ones from HACS-token in these examples.

Although the method header of Example 1 implies its functionality, but it requires the models to extract useful information from its body to determine "what" contains the given value. Checking the results, we can see that none of the baselines is able to predict the target word "hashtable" while they can capture the word "value". By contrast, HACS-token and HACS-AST could produce matched summaries. In order to find out why HACS generates better results, we visualize the (decoder) attention weights of both HACS-AST and biLSTM-AST during decoding, as shown in Fig. 5 (a) and Fig. 6 (a) respectively. In these figures, the lighter the pixel, the bigger the weight at that position, and the special token "</s>" denotes the end-of-sentence tag. We can see from Fig. 6 (a) that when predicting "value", biLSTM-AST is exactly attending to the "value" token in the code, indicating that it is able to perform token copy with the help of attention. However, in its generated sequence, "map" is less accurate than the target "hashtable". When predicting this word, it focuses mostly on the root node *MethodDeclaration*. In this row, a token "tab" is also assigned little attention, but not enough to guide the model to make good prediction. Fig. 5 (a) is the visualization of β_i in (4), i.e., how HACS-AST locates important statement subtrees. When it predicts "hashtable", it is attending to three statements: two *for* statements and the nested *if* statements. From the source code we can find that the user-defined variables "tab" and "e" within these statements is associated with "table". Taking a closer look at the subtrees of two *for* statements, we find that the intra-attention weights α_{ij} for the "tab" tokens are up to 0.18 and 0.14 respectively, which are much larger than other tokens. Since we force the decoder to attend to both token-level and statement-level encoding results, the signal from the token "tab" should be detected by the decoder, and then it could predict the most likely next word is "hashtable" via the learned embedding matrix. It is interesting that a closer term "table" also appears in an assignment statement (line 5), but the model does not focus on it. Note that this does not mean the model cannot see the assignment because each output from biLSTM provides the memory of surrounding contexts. This indicates that at this step, HACS-AST decides the control flows are more important to summarize the code, which is exactly a right decision because they implement the core functionality of the given code snippet. Comparing Fig. 5 (a) and Fig. 6 (a), we can see that HACS-AST is able to attend to various important statements during generation while the attention of biLSTM-AST is always stuck to a small area. This means the former could better capture the global structural information of AST, with the fine-grained token representations retained.

Example 2 shows a longer code snippet. The main difference between the models is whether they can predict the target word "cookie". As seen, only HAN-token and our models could do this. Since the token "cookie" appears in the code, we choose to visualize the attention of biLSTM-token and HACS-token this time, as shown in Fig. 5 (b) and Fig. 6 (b). From Fig. 6 (b), we can find that biLSTM-token is good at copy tokens. Namely, it pays much attention to the method header through the decoding process and directly uses the tokens "json", "object" and "string" within that line. However, it does not consider "cookie" is an important source token which is far from the location it focuses on. We attribute this to its limited capability to handle long input. On contrary, the statement attention of HACS-token covers more critical parts of the input. To be specific, when generating the word "cookie" that missed by biLSTM-token, our model is focusing on line 13 and 15, which is exactly where "cookie" appears. Going deeper, we find that the token "cookie" is assigned the largest intra-attention weight in both of these two statements, of 0.31 and 0.23 respectively. Thus, the model knows this token

Table 4
Results of human evaluation.

Model	Naturalness	Informativeness
DeepCom	4.25	2.33
Code2Seq	4.21	2.59
TreeLSTM	4.23	2.63
ast-attendgru	**4.31**	2.81
HybridHAN+RL	4.29	2.72
HACS-ensemble	**4.31**	**3.16**

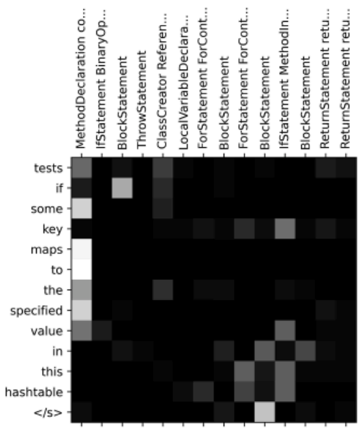
Table 5
Output examples of different models.

Source Code	Example 1	Example 2
	`public synchronized boolean contains(Object value) { if (value == null) { throw new NullPointerException(); } Entry[] tab=table; for (int i=tab.length; i– > 0;) { for (Entry e=tab[i]; e != null; e=e.next) { if (e.value.equals(value)) { return true; } } } return false; }`	`public static String toString(JSONObject jo) throws JSONException { boolean b=false; Iterator keys=jo.keys(); String string; StringBuffer sb=new StringBuffer(); while (keys.hasNext()) { string=keys.next().toString(); if (!jo.isNull(string)) { if (b) { sb.append(';'); } sb.append(Cookie.escape(string)); sb.append("="); sb.append(Cookie.escape(jo.getString(string))); b=true; } } return sb.toString(); }`
Reference	tests if some key maps into the specified value in this hashtable	convert a json object into a cookie list
biLSTM-token	returns true if this map contains the specified value	convert json object to string
biLSTM-AST	returns true if this map maps one or more keys to the specified value	returns a string representation of the json object
HAN-token	returns true if this map maps one or more keys to the specified value	convert a json object into a cookie string
HAN-AST	tests if the specified value is present	convert a json string to a json string
HACS-token	tests if some key maps to the specified value in this hashtable	convert a json object to a cookie string
HACS-AST	tests if some key maps to the specified value in this hashtable	convert a json object into a cookie string
Source Code	Example 3	Example 4
	`private static void innerListFiles(Collection<File> files, File directory, IOFileFilter filter,boolean includeSubDirectories) { File[] found=directory.listFiles((FileFilter) filter); if (found != null) { for (File file: found) { if (file.isDirectory()) { if (includeSubDirectories) { files.add(file); } innerListFiles(files,file,filter, includeSubDirectories); } else { files.add(file); } } } }`	`public void action() { synchronized (myAgent) { NGramDocumentComparatorAgent a= (NGramDocumentComparatorAgent)myAgent; ACLMessage msg=new ACLMessage(ACLMessage.INFORM); msg.setSender(a.getAID()); msg.addReceiver(a.ResultConsumer); try { msg.setContentObject(Result); } catch (IOException ex) { System.err.println("Cannot add result to message. Sending empty message."); ex.printStackTrace(System.err); } a.send(msg); } }`
Reference	finds files within a given directory (and optionally its subdirectories)	actually sends the result to the dispatching agent
biLSTM-token	recursively walks the files in the given directory filter	performs the action
biLSTM-AST	add files to files	this method is called when the user selects an action
HAN-token	lists the files in the given directory	performs the action
HAN-AST	scans the given files recursively	this method sends a message to the server
HACS-token	finds files in the given directory	sends the message to the server
HACS-AST	adds the given files to the given directory	sends a message to the user

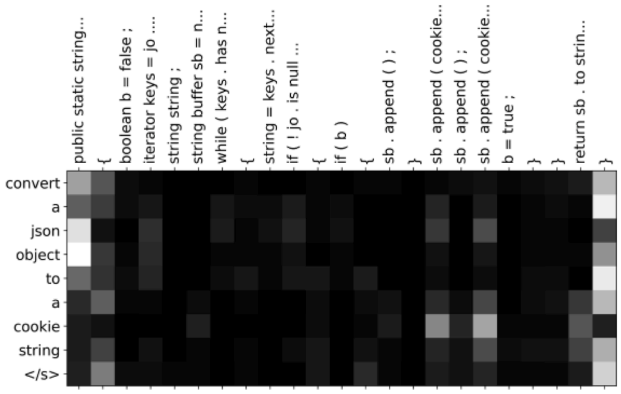
is currently important. Unlike biLSTM-token, HACS-token does not simply refer to the method header when predicting the return value. Instead, it turns to line 13, 15 and the *return* statement to generate "cookie string". In fact, these are important statements that directly associated with the return value "sb".

According to the visualizations, one of the advantages of HACS is that it holds a global view of the code and is able to shift attention between different statements in the generation process. Since the token-based baselines focus mostly on local information and lack high-level knowledge, they might have difficulty to locate keywords in long codes. On the other hand, HACS also keeps the ability of traditional attention, i.e. to copy specific tokens, despite that the process is indirect (as described in Section 3.3). Since HACS locates important tokens in a hierarchical manner, the result can be more accurate. Example 3 and 4 are the cases that HACS copies more accurately than the baselines. In example 3, only HACS-token could predict the correct action word "find" which does not appear in the method signature. This suggests that it further locates the token "found" in the following statements. The method name of example 4 is quite ambiguous, which totally disturbs biLSTMs and HAN-token. By contrast, our models perform much better: they are able to produce the summary of "send message".

As seen in Table 5, HAN-AST and HAN-token could sometimes produce good summary, but many of their outputs are flawed. This suggests that the representation from HAN could provide some useful features,

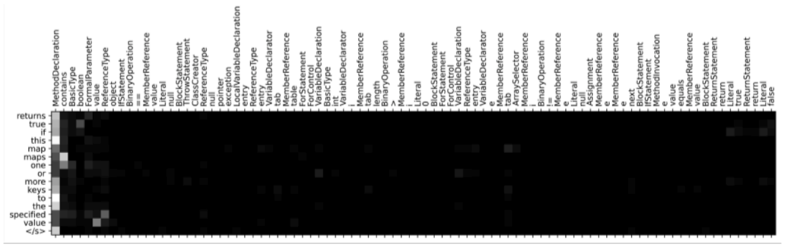


(a) HACS-AST (Example 1)

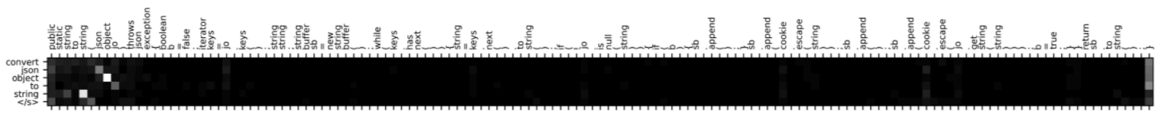


(b)HACS-token (Example 2)

Fig. 5. Visualizations of attention in HACS-AST and HACS-token.



(a) biLSTM-AST (Example 1)



(b) biLSTM-token (Example 2)

Fig. 6. Visualizations of attention in biLSTM-AST and biLSTM-token.

but the final model cannot make full use of them as HACS does. The main reason is that the top-level feature vector of HAN is less informative, and its paired decoder is not able to perform fine-grained feature selection from input.

6.2. *Time cost*

To examine the time cost of different models, we record their speed of transforming the test data under the same settings (batch size of 80, enabling beam search), as shown in Table 6. We do not compare with code2seq and HybridHAN+RL because they are implemented on different platform to others. Although DeepCom and biLSTM-AST share the same model architecture and both take flattened AST as input, we can see that biLSTM-AST is 13% slower. This is due to the node sequence input to biLSTM-AST contains many subtokens separated from the leaves, which leads to longer encoding time. For the same reason, it is also slower than CNN-TreeLSTM. biLSTM-token is the fastest baseline for its simplest design and input. Since HACS includes extra code encoder and more complex attention mechanisms, it seems like it will be more time-consuming than standard Seq2Seq model. However, we can see from Table 6 that HACS-AST / token are as fast as biLSTM-AST / token. The reasons are as follows. First, despite that HACS has two levels of encoder, each of them takes much shorter sequence: For any code snippet, the input length for the token encoder is the number of tokens (AST nodes) within the statement, and the input length for the statement encoder is the number of statements within the code. In addition, different statements can be encoded simultaneously by the token encoder. Thus, the efficiency of HACS is still high. However, we can see that HACS-ensemble is much slower than its dual-input counterpart, i.e. ast-attendgru. One of the main reasons is that the GRU of ast-attendgru is much faster than the biLSTM. On the other hand, since only a single GPU is used for our experiments, we need to calculate the outputs of HACS-AST and HACS-token sequentially to obtain the ensemble result. Fortunately, the training and inference of the individuals within HACS-ensemble is totally parallelizable, so the time cost can be significantly reduced if assign them to multiple GPUs. Overall, our approach is applicable.

Table 6
Inference time costs of different models.

Model	Time (sec. / batch)
DeepCom	0.68
CNN-TreeLSTM	0.70
ast-attendgru	0.75
biLSTM-AST	0.77
biLSTM-token	0.66
HACS-AST	0.74
HACS-token	0.68
HACS-ensemble	1.01

6.3. *Threats to validity*

The major threat to validity of this work is the dataset we use. First of all, the original datasets from [10, 16] are not classified by project and there is no extra tag on each method, so we are not able to split our filtered dataset by projects. This may cause information leakage from training data to validating and testing data. Zhou et al. [17] alleviate this threat by randomly shuffle the dataset to make sure that the data distribution is close to uniform. In addition, only Java language is used to evaluated our model. It is apparent that different programing languages have different characteristics, which may affect the performance of models. It is necessary to collect multi-language dataset with both quality and size, and also interesting to check whether existing models can be well generalized from one programming language to another.

For code2seq, we use the official implementation of the authors[5]. During data preprocessing, we found that a small proportion (around 8%) of our data are dropped by its customized JavaExtractor. We assume that it may fail to parse these codes or cannot extract enough AST paths for the model to use. This leads to another threat to validity, which could slightly affect the scores we calculated for code2seq.

7. Conclusion and future work

In this work, we propose a new encoder-decoder architecture for code summarization, where the encoder extracts hierarchical representations of code and the decoder makes good use of these features for better summary generation. We utilize both inputs of token sequence and AST via ensemble decoding. Detailed data processing including the construction of inputs is provided. In conclusion, attending to hierarchical representation of code could significantly improve the performance of neural models. Different from token or AST-node-based models, our approach could better learn global information of code and shift attention between important statements during summary generation. With the cooperation of intra attention and decoder attention, they are able to locate keywords more accurately in a top-down way. Also, ensemble learning is proved to be a powerful way to benefit from multiple input sources, and can be considered as an alternative to existing combination strategies. For future work, more complex code hierarchies need to be investigated. We will also try to extend our approach to advanced input forms and neural networks such as GNNs and Tree-RNNs.

CRediT authorship contribution statement

Ziyi Zhou: Conceptualization, Methodology, Software, Investigation, Visualization, Writing – original draft. **Huiqun Yu:** Writing – review & editing, Supervision, Funding acquisition. **Guisheng Fan:** Writing – review & editing. **Zijie Huang:** Investigation, Software. **Xingguang Yang:** Investigation, Validation.

Declaration of Competing Interest

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

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61772200), Shanghai Natural Science Foundation (No. 21ZR1416300).

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2021.106761.

[5] https://github.com/tech-srl/code2seq

References

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A.E. Hassan, S. Li, Measuring program comprehension: A large-scale field study with professionals, IEEE Trans. Softw. Eng. 44 (10) (2018) 951–976.
[2] September 20-24, 2010 G. Sridhara, E. Hill, D. Muppaneni, L.L. Pollock, K. Vijay Shanker, Towards automatically generating summary comments for java methods, in: C. Pecheur, J. Andrews, E.D. Nitto (Eds.), ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, ACM, 2010, pp. 43–52. September 20-24, 2010.
[3] S. Haiduc, J. Aponte, L. Moreno, A. Marcus, On the use of automated text summarization techniques for summarizing source code, in: 17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA, G.Antoniol, M.Pinzger, and E. J.Chikofsky, IEEE Computer Society, 2010, pp. 35–44.
[4] G. Sridhara, L.L. Pollock, K. Vijay-Shanker, Automatically detecting and describing high level actions within methods, in: R.N. Taylor, H.C. Gall, N. Medvidovic (Eds.), Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011, ACM, 2011, pp. 101–110.
[5] S. Haiduc, J. Aponte, A. Marcus, Supporting program comprehension with source code summarization, in: J. Kramer, J. Bishop, P.T. Devanbu, S. Uchitel (Eds.), Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, ACM, 2010, pp. 223–226.
[6] P.W. McBurney, C. McMillan, Automatic source code summarization of context for java methods, IEEE Trans. Softw. Eng. 42 (2) (2016) 103–119.
[7] D. Movshovitz-Attias, W.W. Cohen, Natural language models for predicting programming comments, in: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Short Papers. The Association for Computer Linguistics, 2013, pp. 35–40. Volume 2.
[8] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer, Summarizing source code using a neural attention model, in: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Long Papers. The Association for Computer Linguistics, 2016. Volume 1.
[9] M. Allamanis, H. Peng, C. Sutton, A convolutional attention network for extreme summarization of source code, in: M. Balcan, K.Q. Weinberger (Eds.), Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, ser. JMLR Workshop and Conference Proceedings 48, JMLR.org, 2016, pp. 2091–2100.
[10] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: F. Khomh, C. K. Roy, J. Siegmund (Eds.), Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018, ACM, 2018, pp. 200–210.
[11] U. Alon, S. Brody, O. Levy, E. Yahav, code2seq: Generating sequences from structured representations of code, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019. OpenReview.net.
[12] J. Zhang, X. Wang, H. Zhang, H. Sun, X. Liu, Retrieval-based neural source code summarization, in: G. Rothermel, D. Bae (Eds.), ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, ACM, 2020, pp. 1385–1397.
[13] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, P.S. Yu, Improving automatic source code summarization via deep reinforcement learning, in: M. Huchard, C. Kastner, G. Fraser (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM, 2018, pp. 397–407.
[14] A. LeClair, S. Jiang, C. McMillan, A neural model for generating natural language summaries of program subroutines, in: J.M. Atlee, T. Bultan, J. Whittle (Eds.), Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE /ACM, 2019, pp. 795–806.
[15] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation with hybrid lexical and syntactical information, Empir. Softw. Eng. 25 (3) (2020) 2179–2217.
[16] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, Z. Jin, Summarizing source code with transferred API knowledge, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, 2018, pp. 2269–2275. J. Lang, Ed. ijcai.org.
[17] Z. Zhou, H. Yu, G. Fan, Effective approaches to combining lexical and syntactical information for code summarization, Softw. Pract. Exp. 50 (12) (2020) 2313–2336.
[18] P. Fernandes, M. Allamanis, M. Brockschmidt, Structured neural summarization, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019. OpenReview.net.
[19] S. Liu, Y. Chen, X. Xie, J.K. Siow, Y. Liu, Automatic code summarization via multi-dimensional semantic fusing in GNN, in: CoRR, 2020 vol. abs/2006.05405.
[20] A. LeClair, S. Haque, L. Wu, C. McMillan, Improved code summarization via a graph neural network, in: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020, ACM, 2020, pp. 184–195.
[21] Q. Chen, M. Zhou, A neural framework for retrieval and summarization of source code, in: M. Huchard, C. Kastner, G. Fraser (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM, 2018, pp. 826–831.
[22] Z. Yao, J.R. Peddamail, H. Sun, Coacor: Code annotation for code retrieval with reinforcement learning, in: L. Liu, R.W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, L. Zia (Eds.), The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019, ACM, 2019, pp. 2203–2214.

[23] B. Wei, G. Li, X. Xia, Z. Fu, Z. Jin, Code generation as a dual task of code summarization, in: H.M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alche-Buc, E. B. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS2019, December 8-14, 2019, Vancouver, BC, Canada, 2019, pp. 6559–6569.

[24] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, S. Zhang, Leveraging code generation to improve code retrieval and summarization via dual learning, in: Y. Huang, I. King, T. Liu, M. van Steen (Eds.), WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020, ACM, 2020, pp. 2309–2319. /IW3C2.

[25] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, G. Xu, Reinforcement-learning-guided source code summarization via hierarchical attention, IEEE Trans. Softw. Eng. (2020), 1–1.

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

[27] A meeting of SIGDAT, a Special Interest Group of the ACL K. Cho, B. van Merrienboer, C¸. Gulc¸ehre, D. Bahdanau, F. Bougares,¨ H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, in: A. Moschitti, B. Pang, W. Daelemans (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, ACL, 2014, pp. 1724–1734. A meeting of SIGDAT, a Special Interest Group of the ACL.

[28] T. Luong, H. Pham, C.D. Manning, Effective approaches to attention-based neural machine translation, in: L. Marquez,` C. Callison-Burch, J. Su, D. Pighin, Y. Marton (Eds.), Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015, The Association for Computational Linguistics, 2015, pp. 1412–1421.

[29] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, et al., Google's neural machine translation system: Bridging the gap between human and machine translation, in: CoRR, 2016 vol. abs/1609.08144.

[30] A.T. Nguyen, T.N. Nguyen, Automatic categorization with deep neural network for open-source java projects, in: S. Uchitel, A. Orso, M.P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, IEEE Computer Society, 2017, pp. 164–166.

[31] Z. Yang, D. Yang, C. Dyer, X. He, A.J. Smola, E.H. Hovy, Hierarchical attention networks for document classification, in: K. Knight, A. Nenkova, O. Rambow (Eds.), NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016, The Association for Computational Linguistics, 2016, pp. 1480–1489.

[32] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural. Comput. 9 (8) (1997) 1735–1780.

[33] K. Cho, B. van Merrienboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: Encoder-decoder approaches, in: D. Wu, M. Carpuat, X. Carreras, E.M. Vecchi (Eds.), Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014, Association for Computational Linguistics, 2014, pp. 103–111.

[34] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010, ser. JMLR Proceedings, Y. W.Tehand D. M. Titterington, Eds., vol. 9. JMLR.org, 2010, pp. 249–256.

[35] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, Y.Bengioand Y.LeCun, Eds, 2015.

[36] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.

[37] K. Papineni, S. Roukos, T. Ward, W. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA, ACL, 2002, pp. 311–318.

[38] C.-Y. Lin, Rouge: a package for automatic evaluation of summaries, in: Workshop on Text Summarization Branches Out, Post-Conference Workshop of ACL 2004, Barcelona, Spain, July 2004.

[39] B. Chen, C. Cherry, A systematic comparison of smoothing techniques for sentence-level BLEU, in: Proceedings of the Ninth Workshop on Statistical Machine Translation, WMT@ACL 2014The Association for Computer Linguistics, June 26-27, 2014, Baltimore, Maryland, USA, 2014, pp. 362–367.

[40] M. Denkowski, A. Lavie, Meteor universal: language specific translation evaluation for any target language, in: Proceedings of the Ninth Workshop on Statistical Machine Translation, Maryland, USA, June 2014.

[41] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, S. Nakamura, Learning to generate pseudo-code from source code using statistical machine translation, in: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, NE, USA, November 9-13, IEEE Computer Society, 2015, pp. 574–584.

[42] B. Wei, Y. Li, G. Li, X. Xia, Z. Jin, Retrieve and refine: exemplar-based neural comment generation, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, Australia, September 21-25, IEEE Computer Society, 2020, pp. 349–360.