Information and Software Technology 134 (2021) 106542

Contents lists available at ScienceDirect

Information and Software Technology

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

Self-Attention Networks for Code Search

Sen Fang [a], You-Shuai Tan [a], Tao Zhang [a,*], Yepang Liu [b]

[a] *Macau University of Science and Technology, Macau, China*
[b] *Southern University of Science and Technology, Shenzhen, China*

ARTICLE INFO

Keywords:
Code search
Self-attention mechanism
Joint embedding

ABSTRACT

Context: Developers tend to search and reuse code snippets from a large-scale codebase when they want to implement some functions that exist in the previous projects, which can enhance the efficiency of software development.
Objective: As the first deep learning-based code search model, DeepCS outperforms prior models such as Sourcere and CodeHow. However, it utilizes two separate LSTM to represent code snippets and natural language descriptions respectively, which ignores semantic relations between code snippets and their descriptions. Consequently, the performance of DeepCS falls into the bottleneck, and thus our objective is to break this bottleneck.
Method: We propose a self-attention joint representation learning model, named SAN-CS (**S**elf-**A**ttention **N**etwork for **C**ode **S**earch). Comparing with DeepCS, we directly utilize the self-attention network to construct our code search model. By a weighted average operation, self-attention networks can fully capture the contextual information of code snippets and their descriptions. We first utilize two individual self-attention networks to represent code snippets and their descriptions, respectively, and then we utilize the self-attention network to conduct an extra joint representation network for code snippets and their descriptions, which can build semantic relationships between code snippets and their descriptions. Therefore, SAN-CS can break the performance bottleneck of DeepCS.
Results: We evaluate SAN-CS on the dataset shared by *Gu et al.* and choose two baseline models, DeepCS and CARLCS-CNN. Experimental results demonstrate that SAN-CS achieves significantly better performance than DeepCS and CARLCS-CNN. In addition, SAN-CS has better execution efficiency than DeepCS at the training and testing phase.
Conclusion: This paper proposes a code search model, SAN-CS. It utilizes the self-attention network to perform the joint attention representations for code snippets and their descriptions, respectively. Experimental results verify the effectiveness and efficiency of SAN-CS.

1. Introduction

Code search is a frequent activity in software development, which can help developers to find suitable code snippets in their projects. It helps improve developers' productivity and shorten the product development cycle [1–3]. Therefore, a code search tool with high performance is essential for developers.

However, it is extremely challenging to design a practically useful code search tool. Over the past few years, lots of code search methods based on information retrieval (IR) have been proposed [4–7], all of which primarily focus on measuring the text similarity between a query and a code snippet. As a result, they ignored the semantic relationship between the high-level description expressed by natural language and low-level source code, which undoubtedly would affect the performance of code search [8,9]. For example, *Fei et al.* [7] proposed CodeHow, which utilized an expanded Boolean model to extract relations between descriptions and its relevant APIs. Rosf [10] can extract the feature relation of the descriptions and code snippets by combining the IR techniques and supervised learning. Unfortunately, these studies failed to build a semantic bridge between code snippets and descriptions. There are, of course, some methods that can take advantage of the semantic relationship between the code snippet and its description to some degree. For instance, FACoY [5] can capture semantic information by query alternation strategy and recommends code snippets that are similar to the inputting code snippet. *Sirres et al.* [11] proposed an approach that can augment user queries by the relevant but missing structural code entities. In our object, however, we

* Corresponding author.
E-mail addresses: tazhang@must.edu.mo (T. Zhang), liuyp1@sustech.edu.cn (Y. Liu).

https://doi.org/10.1016/j.infsof.2021.106542
Received 18 October 2020; Received in revised form 29 January 2021; Accepted 2 February 2021
Available online 10 February 2021

only want to utilize natural language query to perform code search. The user query of FACoY is a code snippet and *Sirres et al.* utilized relevant code entities to enrich the user query.

Different from IR-based methods, deep learning technologies can automatically learn feature representations and build mapping relationships between inputs and outputs [12,13]. In other words, deep learning is very helpful for bridging the semantics gap between code snippets and their descriptions. Under this consideration, *Gu et al.* [14] constructed a model entirely using deep learning techniques, named DeepCS[1] (**D**eep **C**ode **S**earch). Specifically, DeepCS exploits two separate LSTM (**L**ong **S**hort-**T**erm **M**emory) [15,16] to embed code snippets and their corresponding descriptions, mapping them into two different vector spaces, respectively, and then aligns these two vector spaces by a learned joint embedding model. After completing this step, the whole model returns the most similar code snippet for every query inputted by developers.

As the first code search model that uses deep learning, DeepCS outperforms many traditional models such as Sourcerer [6] and CodeHow [7]. However, researchers observe that there is still room for the improvement of DeepCS, both in the efficiency and effectiveness. There are two reasons: (1) the execution efficiency of LSTM is slow because of its special architecture [15], which is specifically described in Section 2.4. (2) Code and natural language are data from different modal. It is beneficial to embed them to the same vector space if we can build a semantic relation between these different modal data before performing the embedding operation. Therefore, *Shuai et al.* [17] proposed a code search approach based on convolution neural network (CNN) and build semantic relation of code-query via the co-attention mechanism. *Li et al.* [18] proposed CQIL, a CNN-based code search model that can build code-query correlation by hybrid representation. Observing these two approaches, they all first represent code snippets and their queries respectively, and then use an extra component to build the semantic relationship between code snippets and their queries. Therefore, there is an interesting question: as this extra component can build code-query correlation, can we also directly use it to represent code snippets and their queries?

Under the above consideration, we propose a self-attention joint representation learning model named SAN-CS[2] (**S**elf-**A**ttention **N**etworks for **C**ode **S**earch). Specifically, we directly use self-attention networks [19] to learn the contextual representation, separately for code snippets and their queries, due to the advantage that self-attention networks can capture global semantic relations (contextual information) and have a high execution efficiency, which is introduced in Sections 3 and 2.4. Besides, inspired by Transformer [19] which uses the encoder–decoder attention network to build semantic relations between source language sentences and target language sentences, we further utilize a self-attention network to construct an extra joint representation network, for building the semantic relationship between code snippets and their queries. By implementing this step, the code vector can have a more deep-level semantic relation with the query vector.

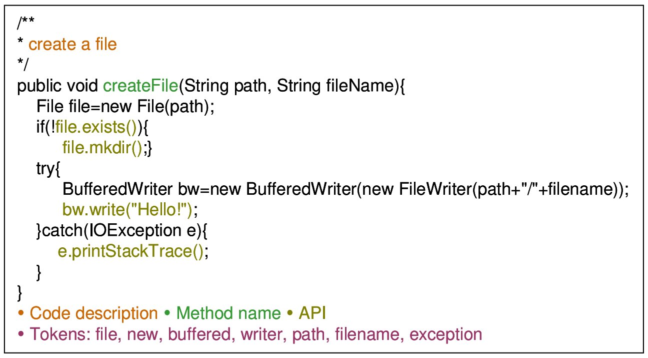
To evaluate the effectiveness of SAN-CS, we conduct a series of experiments on the public dataset [14] shared by *Gu et al.* and choose two baseline model, DeepCS and CARLCS-CNN. The experimental results indicate that SAN-CS outperforms DeepCS and CARLCS-CNN under the metric of MRR (0.908 vs. 0.568 and 0.535). In addition to it, SAN-CS can run faster in the training and testing process than DeepCS.

To sum up, the major contributions of our work are as follows:

- We propose a new code search model based on self-attention networks, *i.e.*, SAN-CS. This model can build semantic representation for both code snippets and their queries through self-attention networks and an joint representation network.

[1] deep-code-search.

[2] https://github.com/TomasAndersonFang/SANCS.



```
/**
* create a file
*/
public void createFile(String path, String fileName){
    File file=new File(path);
    if(!file.exists()){
        file.mkdir();}
    try{
        BufferedWriter bw=new BufferedWriter(new FileWriter(path+"/"+filename));
        bw.write("Hello!");
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

• Code description • Method name • API
• Tokens: file, new, buffered, writer, path, filename, exception

Fig. 1. An example of code snippet.

- We evaluate the effectiveness of SAN-CS on a large-scale public dataset, and the experimental results demonstrate that it performs better than DeepCS and CARLCS-CNN because we can effectively build the semantic relations between code snippets and their queries.

The remaining of this paper includes the following parts. Section 2 introduces the background of a deep learning-based code search and our motivation. Section 3 describes our proposed model in detail. Section 4 presents the experimental setup and results. Section 5 discusses why our approach performs better and Section 6 introduces the related works. Finally, we conclude the paper and describe the future research plans in Section 7.

2. Background and motivation

In this section, we introduce the background of code search approaches that use deep learning technologies, mainly including how to use neural networks to embed code snippets and their queries, as well as how to evaluate the validity of a code search model in the real-world scenario. In addition, we also introduce the motivation of our work.

2.1. Word embedding

Word embedding [20,21], also called distributed representation of words, is an important Natural Language Processing (NLP) technique that uses a fixed-length dense vector to represent each word at high dimensional space. Compared with one-hot representation, distributed representation is able to build semantic relations between different words by estimating their euclidean distance. Moreover, it is also the cornerstone of sentence embedding and document embedding [22–24] techniques.

A well-known word embedding tool is word2vec[3] proposed by Google, which uses the CBOW (**C**ontinuous **B**ag-of-**W**ords) or Skip-Gram model to embed words. Both models are built with a neural network and trained with a large-scale text corpus to capture [20] the semantic relationship between words based on distributed hypothesis [25]. In this paper, we use the CBOW model to embed words because it has a shorter training time [20].

2.2. Sequence embedding

A crucial step of code search is to transform the code snippets and queries to the code vectors and query vectors. By conducting this

[3] https://github.com/danielfrg/word2vec.

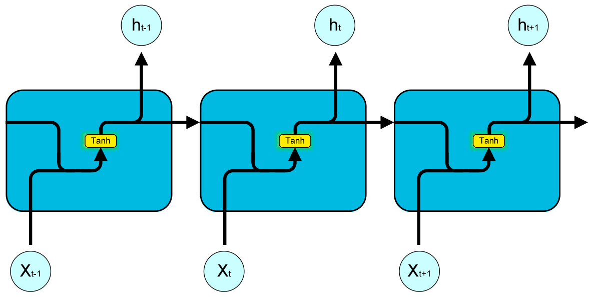


Fig. 2. The basic structure of RNN, X_t represents the word that is inputted into RNN, h_t represents the hidden state of RNN.

step, we can calculate their semantic similarity. The specific embedding processing is described as follows.

Code Sequence Embedding. Before conducting the code embedding, we execute a pre-processing technique for each code snippet. As shown in Fig. 1, every code snippet is divided into three parts: (1) method name sequence, a list of tokens split by camel case; (2) API sequence, a list of API words used in the code snippet; (3) tokens sequence, a list of words used in the code snippet. Then we encode tokens in those three components by the vocabulary with the top-n frequent words that appeared in those three components, respectively. After completing this step, we apply three individual word embedding layers for those three components and transform them into three vectors with the same embedding dimension. Finally, method name and API vectors are further individually embedded through a neural network based on the recurrent neural network (RNN) [26,27], the detailed structure of which is shown in Fig. 2. In DeepCS, it uses the LSTM model to embed method names and APIs vector because it can relieve the gradient disappearing in RNN [15]. At the same time, tokens vectors are further simply embedded via a common multi-layer perceptron (MLP), *e.g.*, a fully connected layer [28]. By completing this step, the whole embedding process can be represented as follows:

$$v_{name} = embed(s_{name}), \tag{1}$$

$$v_{api} = embed(s_{api}), \tag{2}$$

$$v_{token} = embed(s_{token}), \tag{3}$$

$$v_{code} = LSTM_1(v_{name}) + LSTM_2(v_{api}) + MLP(v_{token}), \tag{4}$$

where $embed(\cdot)$ represents word embedding, s_{name}, s_{api} and s_{token} represent method name sequence, API sequence and tokens sequence, respectively. Note that in CNN-based code search method, it uses CNN instead of LSTM.

Query Sequence Embedding. Similar to the code sequence embedding, for a query sequence shown, which consists of a list of English words as shown in Fig. 1, We first encode it using the vocabulary (different from the vocabularies appeared in Code Sequence Embedding). After completing this step, we apply the word embedding layer to the query sequence and transform it into a query vector (v_q) with the same embedding dimension with the code vector. Finally, the query vector is further embedded by the LSTM model. The whole process can be represented as follows:

$$v_q = embed(s_{query}), \tag{5}$$

$$v_{query} = LSTM_3(v_q), \tag{6}$$

where $embed(\cdot)$ represents the word embedding, s_{query} denotes the query sequence, and v_{query} denotes the final embedding output.

2.3. Joint embedding

When finishing the code and query representation, we conduct a joint embedding for them. Specifically, the joint embedding is achieved by measuring the cosine similarity between the code vector and the query vector.

$$cos(v_{code}, v_{query}) = \frac{v_{code} \cdot v_{query}}{\|v_{code}\| \cdot \|v_{query}\|}. \tag{7}$$

To get a good joint embedding model, for each given code vector v_{code}, we provide a positive query vector v^+_{query} and a negative query vector v^-_{query} randomly chosen from the query dataset except v^+_{query}. Our goal is to maximize the similarity of v_{code} and v^+_{query} and minimize the similarity of v_{code} and v^-_{query}, thus we need to minimize the rank loss function [29,30]:

$$L(\theta) = max(0, \xi - cos(v_{code}, v^+_{query}) + cos(v_{code}, v^-_{query})), \tag{8}$$

where θ is the learnable model parameters, ξ is a constant margin. In order to train the aforementioned model well, *Gu et al.* [14] processed about 18 million Java code methods from the open-source Java projects on Github. In detail, for each Java code method, they chose the first line of the documentation comment as the corresponding query. Moreover, they also provided about 10k queries with their corresponding code snippets to evaluate the performance of the model in the real-world condition.

2.4. Motivation

The literature [1] has shown that during a software development cycle, developers spend an average of 20% of their time in searching for reusable code. Therefore, an efficient search engine can effectively save developers' time to develop high-quality software.

In contrast to LSTM, self-attention networks can better capture global semantic information (contextual information) for each element in the sequence as long as having better execution efficiency. Diving into the internal of LSTM, we observe that the computation of the next time step depends on the output of the previous time step. Therefore, if the distance of two elements in a sequence is quite long (suppose the distance is m), LSTM needs to transmit through m times to build the semantic relation between these two elements. Moreover, because of such a sequential transmission mode, each element in a sequence only can build the semantic relation with its left elements, which means that LSTM only can extract limited contextual information for each element in a sequence. From the perspective of efficiency, this mode makes that LSTM cannot achieve parallel computing in GPUs (Graphics Processing Units). Although CNN can better achieve parallel computing, it is not suitable to encode sequence. The reason is that, for each element, CNN only extracts local semantic information for it because of the limited size of the convolution kernel. Therefore, if we want to extract the global semantic information by CNN, we need to stack CNN many times, which complicates our model. As for the self-attention network, it is designed based on the self-attention mechanism. In detail, when representing an element in a sequence, the self-attention mechanism enables this element to pay attention to all the elements in this sequence by a dot-product operation, and then represent this element by the weighted average operation [19]. It illustrates that the self-attention mechanism not needs to consider the distance factor. Therefore, the self-attention network can capture contextual information for each element in the sequence. Considering the execution efficiency, the sequential operation of the self-attention mechanism is $O(1)$, whereas the LSTM requires $O(n)$ sequential operations [19]. In terms of computation complexity, the self-attention layers are faster than the LSTM layers when the sequence length n is smaller than the representation dimensionality d (in our experiments, the max n is 86 and the d is 128) [19]. In addition, the self-attention network can achieve parallel computation in the GPUs because the core of self-attention networks is the dot-product operation. Therefore, Transformer model [19], as the first model that is based solely on self-attention networks, has achieved state-of-the-art performance in the machine translation task that requires a high degree of semantic information, which demonstrates that the self-attention mechanism can better capture the contextual information than LSTM and CNN.

Therefore, due to the capability of self-attention network in capturing the contextual information, we directly adopt self-attention networks to construct code search model that can both achieve better performance and faster execution efficiency.

3. Self-Attention Networks for Code Search

The models built with self-attention networks have achieved great success in many NLP tasks [19,31–35]. Therefore inspired by them, we try to apply self-attention networks to the code search task. In this section, we introduce the architecture of our proposed model SAN-CS in detail.

3.1. An overview of SAN-CS

Fig. 3 gives an overview of our proposed model SAN-CS. The input of SAN-CS contains two parts, one is code snippets composed with the method name, API, and tokens sequences, and another is the query sequence. Then, these four sequences are embedded individually into the corresponding code vector and query vector, where the code vector is obtained by merging method name vector, API vector, and tokens vector. Afterward, SAN-CS learns joint representations for the code vector and query vector.

3.2. Code embedding

Each code snippet is composed of three elements: method name sequence, API sequence, and tokens sequences. We utilize the following four steps to obtain the final code vector.

3.2.1. Embedding for method name

For a given method name sequence, such as “openFile”, we split it into a sequence of words following the widely-adopted camel-case naming convention.[4] Different from API and tokens sequence, the length of the method name sequence is very short (the max length of the method name sequence is about 6), but it is a functionality summarization for the whole code snippet, thus the model we used to embed method sequence should have strong ability to extract semantic information. In terms of the LSTM, we use the output of the last time step in it to represent the entire information of a sentence. In our view, we think it is not enough to represent a sentence by a fixed-length vector with representation dimensionality 128 [12], and via experiments, we also find that the LSTM tends to learn the feature of the end of the sentence, which is not beneficial for representing the entire sentence. Although method names are short, they concisely summarize the functionality of the code snippet, and thus capture the methods’ semantic information. Compared with LSTM, the self-attention network can pay attention to all the words in the sentence, and it can build a contextual relationship for each word, therefore it is proficient to extract semantic information and very suitable to embed the method name sequence. Under the above consideration, we use self-attention networks based on the self-attention mechanism rather than LSTM.

In detail, given a method name sequence $S_n = \{s_1, \ldots, s_i\}$ of length I, we suppose each word in the sequence has been embedded by the word embedding layer with the same representation dimensionality d. We first transform S_n into the query vector $Q_n \in \mathbb{R}^{I\times d}$, the key vector $K_n \in \mathbb{R}^{I\times d}$, and the value vector $V_n \in \mathbb{R}^{I\times d}$ with three individual weight metrics $W_Q \in \mathbb{R}^{d\times d}$, $W_K \in \mathbb{R}^{d\times d}$, and $W_V \in \mathbb{R}^{d\times d}$.

$$Q_n = S_n \cdot W_Q^T, \tag{9}$$

$$K_n = S_n \cdot W_K^T, \tag{10}$$

$$V_n = S_n \cdot W_V^T. \tag{11}$$

After completing the above step, we use the self-attention mechanism [24] to capture semantic information of the method name sequence. Specifically, it can be calculated as follows:

$$context_n = Att(Q_n, K_n) \cdot V_n, \tag{12}$$

where Att(·) is a scaled dot-product attention model, which can be defined by Eq. (13):

$$Att(Q_n, K_n) = \text{SoftMax}(attn), \tag{13}$$

$$attn = \frac{Q_n \cdot K_n^T}{\sqrt{d}}, \tag{14}$$

where $\sqrt{d}$ is a temperature factor that can avoid gradient disappearing of the model during the training phase, and it has the same representation dimensionality d. SoftMax(·) is a normalized function, which is used to get attention weight matrix.

When completing these steps, we can obtain the final output $v_{name} \in \mathbb{R}^{I\times d}$ by applying $context_n$ vector into a simple position-wise fully connected feed-forward network [19].

$$v_{name} = \text{ReLu}(context_n \cdot W_1 + b_1) \cdot W_2 + b_2, \tag{15}$$

where $W_1 \in \mathbb{R}^{d\times 4d}$, $W_2 \in \mathbb{R}^{4d\times d}$, $b_1 \in \mathbb{R}^{4d}$, and $b_2 \in \mathbb{R}^{d}$ are the learnable parameters in the model, RelU(·) is the activation function [36].

3.2.2. Embedding for API

Different from the method name sequence, most of API sequences are much longer. Therefore, for the LSTM model, it is difficult to capture long-distance dependency well [12,37], which means it cannot fully capture semantic information from API sequences. As a result, it will hurt the performance of the code search model. As described in Section 3.2.1, self-attention networks can resolve this problem. Besides, because of the limitation of the structure in LSTM, it cannot achieve parallel computing, therefore we need more time to train LSTM model. As for the self-attention network, it can achieve parallel computing like CNN [38,39], thus we can fully use GPUs to train our model.

Similar to the method name sequence embedding, given an API sequence $S_a = \{s_1, \ldots, s_n\}$ of length N, we suppose each word in the sequence has been embedded by the word embedding layer with the same representation dimensionality d. We first transform S_a into the query vector $Q_a \in \mathbb{R}^{N\times d}$, the key vector $K_a \in \mathbb{R}^{N\times d}$, and the value vector $V_a \in \mathbb{R}^{N\times d}$, and then we utilize the self-attention mechanism to capture semantic information of the API sequence. Finally, we obtain the final output $v_{api} \in \mathbb{R}^{N\times d}$ by applying the output of the self-attention network vector into a simple position-wise fully connected feed-forward network.

$$context_a = \text{SoftMax}\left(\frac{Q_a \cdot K_a^T}{\sqrt{d}}\right) \cdot V_a, \tag{16}$$

$$v_{api} = \text{ReLu}(context_a \cdot W_1 + b_1) \cdot W_2 + b_2. \tag{17}$$

3.2.3. Embedding for tokens

Tokens sequence, extracted from the method body, is composed of a list of words. Through data pre-processing phase, we find that the tokens sequence only contains the informative keywords of the code snippets, which means locative relations between tokens are not strong [14]. A simple example shows that although we exchange the position of two tokens (such as variable name), the function of the code snippet is changeless. Although these tokens have weak location relations, they have enriched semantic information and relations. Because

[4] Camelcase, https://en.wikipedia.org/wiki/Camel_case.

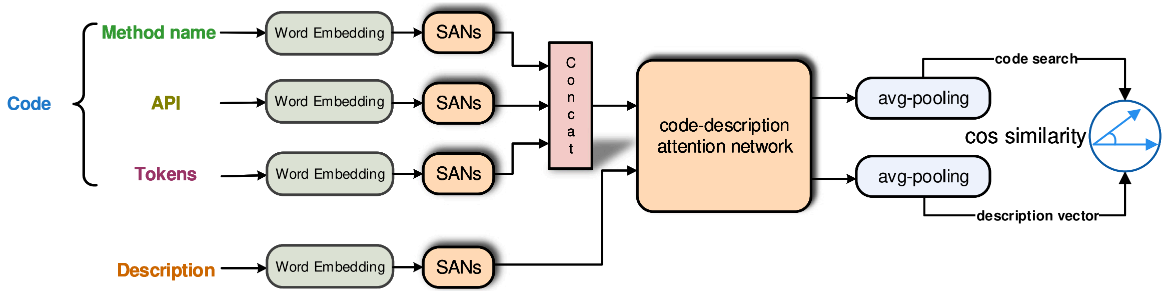


Fig. 3. An overview of SAN-CS.

of the above considerations, we think that just using a simple multi-layer perceptron (MLP) is unable to build semantic relation for each word [40], thus we choose the self-attention networks [19] instead of the MLP to embed tokens sequence. The reason for this choice is that the self-attention network can pay attention to all words in the sentence without considering their locations through the weighted average operation, thus it ignores the relative relation of words (for other sequences embedding, we add an extra position encoding to obtain positional information).

Specifically, given a tokens sequence $S_t = \{s_1, \dots, s_m\}$ of length M, we suppose each word in the sequence has been embedded by the word embedding layer with the same representation dimensionality d. We first transform S_t into the query vector $Q_t \in \mathbb{R}^{M\times d}$, the key vector $K_t \in \mathbb{R}^{M\times d}$, and the value vector $V_t \in \mathbb{R}^{M\times d}$, and then we use the self-attention mechanism to capture semantic information of the API sequence. Finally, we obtain the final output $v_{token} \in \mathbb{R}^{M\times d}$ by applying the output of the self-attention network vector into a simple position-wise fully connected feed-forward network.

$$context_t = \text{SoftMax}(\frac{Q_t \cdot K_t^T}{\sqrt{d}}) \cdot V_t, \tag{18}$$

$$v_{api} = \text{ReLu}(context_t \cdot W_1 + b_1) \cdot W_2 + b_2. \tag{19}$$

3.2.4. Embedding merging

When completing embedding for the method name sequence, API sequence, and tokens sequence, we can gain code embedding $V_{code} \in \mathbb{R}^{(I+N+M)\times d}$ by simply concatenating v_{name}, v_{api}, and v_{token}.

$$V_{code} = concat(v_{name}, v_{api}, v_{token}). \tag{20}$$

3.3. Description embedding

Code descriptions contain the semantic information that can directly reflect the query purposes of the developers. Therefore, similar to the code embedding, we also adopt the self-attention network to embed the code description sequence. Given a code description sequence $S_d = \{s_1, \dots, s_j\}$ of length J, we suppose each word in the sequence has been embedded by the word embedding layer with the same representation dimensionality d. We first transform S_d into the query vector $Q_{d1} \in \mathbb{R}^{J\times d}$, the key vector $K_{d1} \in \mathbb{R}^{J\times d}$, and the value vector $V_{d1} \in \mathbb{R}^{J\times d}$, and then we use the self-attention mechanism to capture semantic information of the API sequence. Finally, we obtain the final output $V_{desc} \in \mathbb{R}^{J\times d}$ by applying the output of the self-attention network vector into a simple position-wise fully connected feed-forward network.

$$context_t = \text{SoftMax}(\frac{Q_{d1} \cdot K_{d1}^T}{\sqrt{d}}) \cdot V_{d1}, \tag{21}$$

$$V_{desc} = \text{ReLu}(context_d \cdot W_1 + b_1) \cdot W_2 + b_2. \tag{22}$$

3.4. Code-description attention

When completing code embedding and its paired code description embedding, we can collect $< V_{code} \in \mathbb{R}^{H\times d}, V_{desc} \in \mathbb{R}^{J\times d} >$ pairs. Here H equals to $I + N + M$. In order to map the $\langle V_{code}, V_{desc}\rangle$ pairs into the same vector space better, we adopt an extra joint embedding for them by the code-description attention network (a variant of the self-attention network) before mapping them to the same vector space. In detail, we first transform V_{code} into the query vector $Q_c \in \mathbb{R}^{H\times d}$, and the value vector $V_c \in \mathbb{R}^{J\times d}$ and transform V_{desc} into the keys $K_d \in \mathbb{R}^{J\times d}$, and the values $V_d \in \mathbb{R}^{J\times d}$ with three corresponding weight metrics.

$$Q_c = V_{code} \cdot W_{qc}^T, \tag{23}$$

$$V_c = V_{code} \cdot W_{vc}^T, \tag{24}$$

$$K_d = V_{desc} \cdot W_{kd}^T, \tag{25}$$

$$V_d = V_{desc} \cdot W_{vd}^T. \tag{26}$$

Then we compute code-description attention matrix A $\in \mathbb{R}^{H\times J}$ as follows:

$$A = SoftMax(\frac{Q_c \cdot K_d^T}{\sqrt{d}}). \tag{27}$$

The code-description attention matrix allows V_{code} and V_{desc} to pay attention to each other during the joint embedding phrase. And we can represent the outputs of code-description attention network as follows,

$$\boldsymbol{c} = A \cdot V_d, \tag{28}$$

$$\boldsymbol{d} = A^T \cdot V_c, \tag{29}$$

where $\boldsymbol{c}$ is V_c joint embedding with V_d, and $\boldsymbol{d}$ is V_d joint embedding with V_c.

Next, we conduct an average-pooling operation to $\boldsymbol{c}$ and $\boldsymbol{d}$ separately to obtain semantic vectors $\boldsymbol{C} \in \mathbb{R}^d$ and $\boldsymbol{D} \in \mathbb{R}^d$. Moreover, in the experiment, we also find that max-pooling makes our model invalid. One reasonable interpretation is that the self-attention network builds a contextual vector for each word vector, and max-pooling only captures the feature with the highest value, which significantly hurts the semantic relationship between each word. The concrete definition is as follows:

$$\boldsymbol{C} = avgpooling([c_1, \dots, c_h]), \tag{30}$$

$$\boldsymbol{D} = avgpooling([d_1, \dots, d_j]). \tag{31}$$

3.5. Training

Now we present the training details of SAN-CS model. To make SAN-CS learn joint representation for code snippets and descriptions, we hope that our model can make code and description vectors that have similar semantics as close as possible in the vector space. In addition, for a random code vector $\boldsymbol{c}$, we also hope that our model can give a high similarity to its positive description, and give a low similarity to other negative descriptions. Based on the above consideration, we achieve our purpose by minimizing the rank loss during the training time.

$$L(\theta) = \sum_{\langle c,d^+,d^-\rangle \in T} max(0, \xi - cos(\boldsymbol{c}, \boldsymbol{d}^+) + cos(\boldsymbol{c}, \boldsymbol{d}^-)), \tag{32}$$

where θ is the model parameters, T is the dataset to train SAN-CS model, and $cos(\cdot)$ is used to compute the similarity. From the rank loss function, we note that for each code vector $\boldsymbol{c}$, there is a positive description vector $\boldsymbol{d}^+$ and a negative description vector $\boldsymbol{d}^-$, and it encourages $\boldsymbol{c}$ and $\boldsymbol{d}^+$ to have a high similarity, and $\boldsymbol{c}$ and $\boldsymbol{d}^-$ to have a low similarity. ξ is a margin constant that avoids gradient disappearing.

At the training time, we utilize the Adam optimizer [41] to optimize our rank loss function. SAN-CS model gets the basic representation of code snippets and descriptions, and then those two representations pass through a code-description attention network to generate a joint representation for code snippets and descriptions, respectively. In the back-propagation phase, the updated model parameters θ guide the code-description attention network to generate joint representations of code snippets and descriptions that can minimize the rank loss function.

3.6. Prediction

After completing the training phase, SAN-CS model has embedded all code snippets and descriptions in the training dataset to the same vector space. For a query from the developer, for instance, “close a writer object”, SAN-CS model first transforms the query into a query vector $\boldsymbol{q}$, and then the model calculates the semantic similarity between $\boldsymbol{q}$ and each code vector $\boldsymbol{c}$ in the vector space. Ultimately, the model obtains the top-k code snippets related to the query and recommends them to the developer.

4. Evaluation

4.1. Experimental setup

4.1.1. Research questions

Our work mainly focus on the following four research questions (RQ):

RQ1:

Can our proposed model SAN-CS work well?

In RQ1, we mainly want to investigate whether SAN-CS model is more effective comparing to the code search model DeepCS [14] and CARLCS-CNN [17]. If our experimental results can support SAN-CS, this indicates that the self-attention network is beneficial for code search.

RQ2:

Can SAN-CS get good performance in term of efficiency?

RQ2 aims at exploring the model efficiency. DeepCS can provide good search results, but it needs a huge amount of time in training and has a long response time. Thus if SAN-CS has good performances in terms of both effectiveness and efficiency, it will have more value in practice.

RQ3:

Can self-attention networks capture contextual information better than LSTM and CNN?

As described in Sections 3.2 and 3.3, we exploit self-attention networks, instead of LSTM or CNN, to learn the representation for code sequences and description sequences, respectively. In Section 3.4, we use a self-attention network to built a joint representation network, for adopting an extra joint representation to the code and description vectors. The goal of making these choices is to capture the semantic relations between the code and description vectors better. Therefore, in this RQ, we want to explore whether those two self-attention networks can make the model capture semantic information better.

Table 1
Statistics of the dataset.

Dateset	Size	Language	Time
Training	18.23M	Java	2008.8-2016.6
Testing	10000	Java	2008.8-2016.6

RQ4:

How do different parameter settings of the self-attention network, such as representation dimensionality d and the number of self-attention layers, affect the model effectiveness??

In SAN-CS, the size of representation dimensionality d and the number of the self-attention layers play an important role in the code search effectiveness. To study what parameter settings are most suitable for SAN-CS, we perform two groups of controlled experiments by using the different representation dimensionality d and different number of the self-attention layer.

4.1.2. Dataset and baseline model

We conduct our experiments on the dataset released by *Gu et al.* [14]. They had evaluated their code search model, DeepCS, on this dataset. As shown in Table 1, the dataset for training is composed of over 18 million public Java code methods collected from GitHub repositories with at least one star, covering Aug. 2008 to Jun. 2016. The dataset for testing contains 10K code-query pairs collected from GitHub, and it can effectively alleviate the bias from the manual evaluation. Some queries and code snippet examples are shown in Table 2 and Fig. 4. By the way, to guarantee the fairness of subsequent comparison experiments, we do not make any changes to this dataset.

Baseline: DeepCS [14]. It is the first and state-of-the-art model that applies deep learning to the code search. To get the best performance of DeepCS, we directly reuse the source code and the trained model, both of which are shared by *Gu et al.* [14]. This can minimize the uncertainty when evaluating.

Baseline: CARLCS-CNN [17]. A CNN-based code search model that uses the co-attention to build the semantic relationship between code snippets and their queries. To get the best performance of CARLCS-CNN, we also directly reuse the source code opened by *Shuai et al.* [17]. In addition, we denote CARLCS-CNN as CARLCS for convenience.

Baseline: SAN-CS. Our proposed model is solely based on the self-attention network, as detailed in Section 3. In our experiments, we utilize the self-attention network with a single self-attention layer to learn to represent the code snippets and their descriptions, and we also set the size of parameter d and word embedding to 128, the learning rate as 10^{-4}.

4.1.3. Evaluation metrics

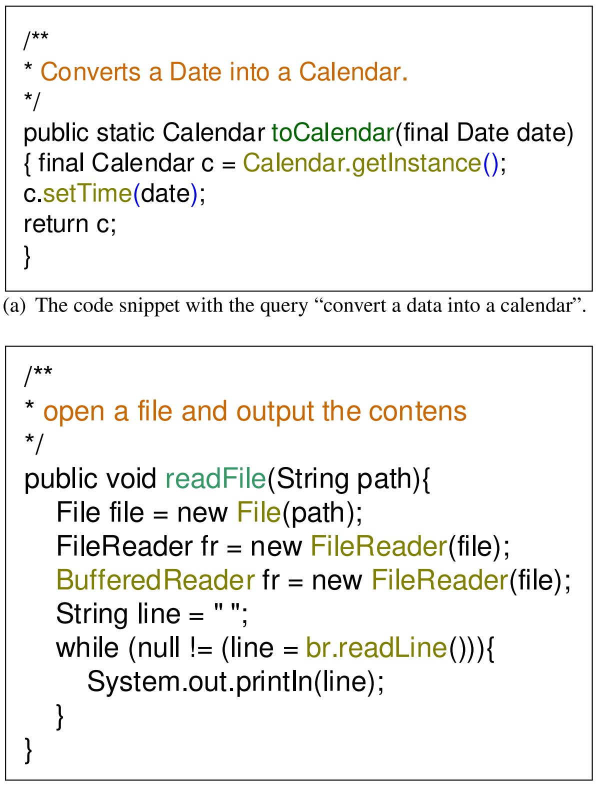
We evaluate the effectiveness of SAN-CS using three different metrics: Recall, NDCG (**N**ormalized **D**iscounted **C**umulative **G**ain), and MRR (**M**ean **R**eciprocal **R**ank). Those metrics are widely used to estimate the code search and some relative tasks [7,42–45].

Recall@k. This metric aims at calculating the percent of queries that are related to code methods and can be indexed in the top-k list.

$$Recall@k = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \epsilon(Q_i <= k), \tag{33}$$

Table 2
Some query examples in the testing dataset.

No.	Description	No.	Description
1	Convert an asn	11	Close the proxy
2	Create a new file	12	Log the details of a file
3	Handle remote methods	13	Test if the file exist
4	Contain the data	14	Load the activities
5	Private constructor	15	Print out the classifier
6	Read the next line	16	Close a writer object
7	Return a string representation of this set	17	Convert a string to a char
8	Create an instance of jaxb element	18	Delete a folder on the hdfs
9	Get int property for activity	19	Converts a date into a calendar
10	Sort an int array into ascending order	20	Run an event on the thread queue



```
/**
* Converts a Date into a Calendar.
*/
public static Calendar toCalendar(final Date date)
{ final Calendar c = Calendar.getInstance();
c.setTime(date);
return c;
}
```

(a) The code snippet with the query "convert a data into a calendar".

```
/**
* open a file and output the contens
*/
public void readFile(String path){
    File file = new File(path);
    FileReader fr = new FileReader(file);
    BufferedReader fr = new FileReader(file);
    String line = " ";
    while (null != (line = br.readLine())){
        System.out.println(line);
    }
}
```

(b) The code snippet with the query "open a file and output the contens".

Fig. 4. Two code snippets with queries in testing dataset.

where Q denotes the 10K queries in our testing dataset, σ is a detecting function that returns 1 if Q_i is in the top-k list, otherwise it returns 0. In order to evaluate our proposed model comprehensively, we calculate Recall@1, Recall@5, and Recall@10.

NDCG [46,47] hopes that search results in the top-k list are as relevant as possible to the query, and also wants the more relevant result to be displayed in the front of the top-k list. We can calculate it as follows,

$$NDCG = \frac{1}{|Q|}\sum_{j=1}^{k}\frac{2^{r_j}-1}{\log_2(1+j)}, \tag{34}$$

where Q denotes the 10K queries in our testing dataset, r_j is relevant search results with position j in the top-k search results, and k denotes the maximum value that NDCG can give the query. A code search model with a high NDCG score means that it not only has a high overall search quality but also ranks the results that users need in the front position.

Table 3
Performance comparison of DeepCS, CARLCS, and SAN-CS. We use R@1/5/10 to express Recall@1/5/10, and the same goes for the rest.

Model	R@1	R@5	R@10	NDCG	MRR
DeepCS	0.585	0.750	0.816	0.626	0.568
CARLCS	0.549	0.713	0.782	0.592	0.535
SAN-CS	**0.931**	**0.956**	**0.962**	**0.921**	**0.908**

MRR primarily finds the index of first relevant result. The MRR is computed as follow,

$$MRR = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{Index_{Q_j}}, \tag{35}$$

where Q denotes 10K queries in the testing dataset; $Index_{Q_j}$ is the index of the first search result related to the ith query (Q_j) in the top-k rank list. If the top-k rank list has no search results related to Q_j, we set the $1/Index_{Q_j}$ equal to 0. The higher the MRR score, the shorter code developers need to inspect to find their expected results, thus we let the value of k equal to 10.

4.2. Results

4.2.1. RQ1: Model effectiveness

Table 3 shows the performance of the state-of-the-art model DeepCS and our proposed SAN-CS under the different metrics. For the metric of Recall@k, SAN-CS outperforms DeepCS and CARLCS by the average of 34.8% and 42.2%, respectively. For the metric of NDCG, SAN-CS model advances DeepCS and CARLCS by 47.1% and 55.6%, respectively. As for the metric of MRR, the scores of DeepCS and CARLCS are 0.568 and 0.535. SAN-CS obtains 59.9% and 69.7% improvement in terms of MRR metric. All these results demonstrate that our proposed model SAN-CS is more effective that DeepCS and CARLCS. Besides, we can also observe that, although CNN has faster execution efficiency than LSTM, LSTM is more suitable than CNN to encode the sequence.

Result1:

SAN-CS outperforms DeepCS and CARLCS in terms of Recall@k, NDCG, and MRR. This indicates that SAN-CS is more effective for code search tasks.

4.2.2. RQ2: Model efficiency

In Table 4, we compare the total parameters, training time, and searching time of two models on the dataset described in Section 4.1.2. To compare model efficiency fairly, we train these two models under the same experimental environment that a server with one Tesla V100-SXM2 GPU with 32 GB video memory. The experimental results demonstrate that, with the condition that SAN-CS has more parameters, DeepCS takes about 1.0 ms to train every sample and 0.6 s to search for each query, while SAN-CS only spends about 0.3 ms on training a single sample and 0.1 s on responding to each code query. This indicates

Table 4
Comparison of the efficiency of DeepCS and SAN-CS (The parameters denote the total model parameters).

Model	Parameters	Training	Searching
DeepCS	5.81M	1.0ms/sample	0.6s/query
SAN-CS	6.64M	0.3ms/sample	0.1s/query

Table 5
Comparison of the effects of the LSTM, CAELCS and SANs.

Model	R@1	R@5	R@10	NDCG	MRR
DeepCS	0.585	0.750	0.816	0.626	0.568
CARLCS	0.549	0.713	0.782	0.592	0.535
SAN-CS⁻	0.595	0.751	0.814	0.632	0.577
SAN-CS	**0.931**	**0.956**	**0.962**	**0.921**	**0.908**

that comparing with DeepCS, SAN-CS has a significant improvement at execution efficiency, and also means SAN-CS is more suitable than DeepCS in practice.

Result2:

Comparing with DeepCS, SAN-CS has a dramatic enhancement in efficiency.

4.2.3. RQ3: The effect of self-attention network

In SAN-CS, self-attention networks and joint representation network play a crucial role, which is described in Section 3. For the code and description embedding, we directly exploit self-attention networks to represent code snippets and their queries. To study whether each self-attention network can improve representation effectiveness, we design a variant of SAN-CS without using the joint representation network (SAN-CS$^-$) and compare it with DeepCS, CARLCS, and SAN-CS. Therefore, if the self-attention network has a stronger representation learning ability, SAN-CS and SAN-CS$^-$ should perform better than DeepCS and CARLCS.

The results in Table 5 demonstrate that SAN-CS$^-$ obtains Recall@1/5/10 scores of 0.595/0.751/0.814, an NDCG score of 0.632, and an MRR score of 0.577, from which we observe that SAN-CS$^-$ only has slight increases compared with DeepCS and CAELCS. These experimental results not only support our idea that embedding the code snippets and descriptions separately may cause the performance bottleneck of the code search model but also illustrate that CNN is not suitable to sequence modeling. When we supplement the joint representation network to SAN-CS$^-$, the results show that this network has an extremely enhancement for SAN-CS$^-$ in terms of all evaluation metrics, which shows that the joint representation learning is beneficial to build semantic relation between the code vector and description vector.

Result3:

Self-attention networks can capture contextual information better than LSTM and CNN.

4.2.4. RQ4: The impact of parameters settings

As mentioned in Section 3, the efficiency of SAN-CS is affected by the size of representation dimensionality d and the number of the self-attention layer. In Table 6, we perform three groups of experiments with the different d, and the experimental results show that when we set the representation dimensionality d to 128, SAN-CS can get the best performance. As for Table 7, we can observe that a single self-attention layer is most suitable for SAN-CS if we take the total model parameters into consideration, which is distinct from the Transformer [19] that requires six self-attention layers for reaching the state-of-the-art results.

Table 6
Comparison of the effect of SAN-CS with different parameter d.

d	R@1	R@5	R@10	NDCG	MRR
64	0.921	0.954	0.958	0.912	0.901
128	**0.931**	**0.956**	**0.962**	**0.921**	**0.908**
256	0.919	0.952	0.960	0.910	0.894

Table 7
Comparison of the effect of SAN-CS with different number of the self-attention layers.

Layers	R@1	R@5	R@10	NDCG	MRR
1	**0.931**	0.954	0.958	**0.912**	**0.908**
2	0.921	**0.955**	**0.961**	**0.912**	0.896
3	0.642	0.800	0.866	0.679	0.622

Result4:

For SAN-CS, it is a good choice to set the representation dimensionality d to 128 when we choose a single self-attention layer.

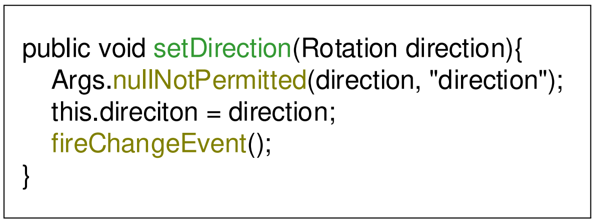
5. Discussion

5.1. Why does SAN-CS work well?

Although LSTM can capture long-range information in the sequence via the gating mechanism [15,37,48], it is actually unable to fully capture the semantic information of the sentences [12,49]. As for CNN, it only extracts local semantic information for it because of the limited size of the convolution kernel. Compared with LSTM and CNN, self-attention networks can model global information with the weighted averaging operation, which enables it to take into account all the elements in an input sentence without considering their distances. The code vector embedded by SAN-CS not only has abundant contextual information, but also semantic relations between the modules (method name sequence, API sequence, and token sequence), which is highly conducive to establish the semantic connection with the description vector and to better make joint embedding. This is why our proposed model SAN-CS can perform better than DeepCS. Fig. 5 shows the first search result of DeepCS and our proposed model for the query “set the attribute value". Form Fig. 5(a) we can observe that DeepCS returns an unrelated code snippet because its method name contains keyword “set" in the query, which means DeepCS cannot capture full semantic information of the code snippet. In contrast, SAN-CS can return the prospective code snippet, as shown in Fig. 5(b). An important reason is that SAN-CS can embed code snippets into a vector containing abundant semantic information, which enables code snippets to match suitable queries at the semantic level.

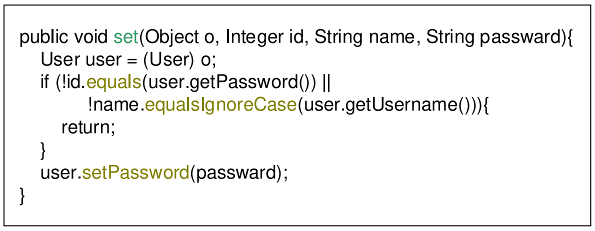
5.2. Why does SAN-CS perform faster?

In the recurrent neural network and its variants [37,48], the computation of the next time step depends on the output of the previous time step, which makes it impossible to conduct parallel computation in the GPUs. In DeepCS, it mainly uses the LSTM model to embed sequences, which makes it spend much more time on training and searching. As introduced in Section 5.1, self-attention network considers all elements in the sentence through the weighted average operation, which means it can make parallel computation well in the GPUs.



```
public void setDirection(Rotation direction){
   Args.nullNotPermitted(direction, "direction");
   this.direciton = direction;
   fireChangeEvent();
}
```

(a) The first research result of DeepCS for the query “set the attribute value”



```
public void set(Object o, Integer id, String name, String passward){
   User user = (User) o;
   if (!id.equals(user.getPassword()) ||
          !name.equalsIgnoreCase(user.getUsername())){
      return;
   }
   user.setPassword(passward);
}
```

(b) The first search result of SAN-CS for the query “set the attribute value”

Fig. 5. The first search result of DeepCS and SAN-CS for the query “set the attribute value".

5.3. *Why did not we use topic model?*

Topic models, such as latent dirichlet allocation (LDA) [50] and latent semantic indexing (LSI) [51], can represent each word as a word vector. However, topic models use co-occurrence relationships of words in documents to cluster words by topic, to represent words with topic information (high-level) [52], such coarse-grained word representation is not suitable for the code search task because we need to recommend appropriate code snippets for each inputted query rather than a topic. In other words, we need fine-grained word representation (at both syntactic and semantic levels) for the code search task. Apart from it, the choice of topics in topic models entirely depends on the researchers, and thus there are some subjective opinions on their choice.

As for language model [53,54], for example, the self-attention network we use in SAN-CS, it first uses word2vec technique [20] to represent each word as a word vector at the semantic and syntactic levels and then uses the self-attention mechanism to further represent each word vector as the word vector with contextual information, finally, it can represent a sentence as a sequence vector by an average pooling operation to word vectors of this sentence and build semantic relationship between the code snippet and its corresponding description. Therefore, SAN-CS is able to recommend appropriate code snippets for each inputted query. Although there are some topically driven language models [55,56] that can achieve the same function, the execution speed of these models is dramatically slow and these models mainly focus on explaining the topics via the sentences generated by language models.

5.4. *Threats to validity*

5.4.1. *Internal threats*

We believe that the internal threats to our proposed model focus on two aspects. The first one is the baseline re-implementation. The experimental environment we use is different from DeepCS and CARLCS-CNN’s, thus we implement and evaluate these two baseline models on our server. However, if we replicate baseline models by ourselves, its performance may have a certain degree of deviation. To mitigate this threat, we re-ran DeepCS and CARLCS-CNN with the source code and dataset shared by the authors. The second threat is the model parameters. If a model has much more parameters than another model, it is much possible to get better performance. We address this threat by keeping our model parameters the same as DeepCS.

5.4.2. *External threats*

The external threat to this work is primarily in the generalization of the proposed approach. While SAN-CS performs well in the test set, this does not necessarily mean that it will perform equally well in enterprise projects. We plan to build a more large-scale dataset and use it to evaluate SAN-CS’s generalization ability.

6. Related work

6.1. *Code search*

In recent years, many works study on how to capture semantic information of the code snippets and queries [4,57–62]. *McMillan et al.* [63] tried to retrieve and visualize relevant functions and their usages for each given query. *Lv et al.* [7] proposed CodeHow, which can utilize a Boolean model to deeply understand the relationship between the APIs and query. *Li et al.*. [42] proposed a relationship-aware model named RACS, a JavaScript code search tool that can use MCR graphs to capture the relationship of the features among the called API methods. Although those traditional models tried to capture semantic information of the code snippets or queries, they ignored that the code snippets in a programming language and their corresponding queries in natural language have semantic relationships. They only let their model to understand code snippets or query, thus they failed to build a bridge to break the semantic gap between the code snippets in programming and query in natural language.

To break this semantic gap, *Gu et al.* [14] proposed the first deep learning-based code search model called DeepCS. However, the operation that DeepCS uses two independent LSTM models to represent the code snippets and queries individually isolates the semantic relationship between the code snippets and queries. Except for DeepCS, *Cambronero et al.* [64] proposed UNIF, in which they added an attention mechanism [12] to LSTM to enforce the representation ability of their model. *Wan et al.* [65] proposed MMAN that is based on tree-LSTM and gated graph neural network. This model can extra consider abstract syntax trees and control-flow graphs of the source code. *Nguyen et al.* [66] proposed FuzzyCatch to recommend code for handling exception. *Li et al.* [18] proposed a CNN-based code search model that can build a semantic correlation between the code snippet and its query. *Yan et al.* [67] performed an empirical study on code search and found that deep learning-based methods have good performance on code search via natural language queries. Besides, they also public a dataset for code search task, named CosBench. *Li et al.* [18] proposed CQIL, a CNN-based code search model that first utilizes CNN to represent code snippets and their queries respectively, and then builds code-query correlation by hybrid representation. *Ling et al.* proposed AdaCS [68], which breaks down the learning process into the domain-specific words and matching general syntactic patterns, therefore it has excellent generalization capabilities. CodeMatcher [69] proposed by *Liu et al.*, which mainly aims to reduce the complexity and time-consuming of DeepCS by combining IR techniques with features in DeepCS. There are some research studies similar to the code search task. For example, *Lin et al.* [70] implemented a tool named CCDemon. This tool can boost the developers’ efficiency by recommending where and how to modify their pasted code. MICoDe, an Eclipse plugin constructed by *Lin et al.* [71], allows developers to make customization for generating new code by using suitable design templates in this tool. The inspiration of these two works comes from the observation in process of development, the insights from software engineering domain (worth utilizing in future study). Although SAN-CS is a deep learning-based code search model, it can deeply extract the semantic relationship between code snippets and their queries, which can help improve the performance of code search.

Different from the above models that add attention mechanism to LSTM or CNN, we directly utilize a kind of attention network, the self-attention network, to construct our code search model, SANCS. To build the semantic relation between code snippets and their queries, we extra construct a joint representation network by using the self-attention network. Therefore, SANCS first utilizes the self-attention networks to represent code snippets and queries separately, and then further adopts an extra joint representation for the code snippets and queries by the joint representation network. For the self-attention networks, it can capture semantics information better than LSTM and CNN; for the joint representation network, it can build the internal relationship in the code snippets and establish semantic relation between the code snippets and descriptions.

6.2. Attention mechanism

Attention mechanism, a technology that simulates the process of human reading and listening, can allow models learning to pay attention to crucial parts in data [12,33,72–74]. It brings a sea of energy to computer vision and NLP. In the field of NLP, attention mechanism is first used in the neural machine translation (NMT) by *Bahdanau et al.* [12], which makes the NMT achieve great improvement. *Wu et al.* [75] also utilized attention networks to construct the Google Neural Machine Translation system. *Rush et al.* [76] proposed a neural attention model to break the limitation of the text extraction-based summarization model, which can generate words of the summary conditioned on the input sentence. However, those models only apply attention mechanisms into the RNN and its variants, they improve the effectiveness of RNN models but cannot solve the limitation of structure in RNN.

To address the above issues, *Vaswani et al.* [19] proposed the self-attention mechanism, and the Transformermodel that is solely based on self-attention networks, which obtained significant successes in all kinds of NLP tasks, thus he led to the tendency of using self-attention in the field of artificial intelligence. *Zhang et al.* [35] proposed a self-attention generative adversarial network, which can generate details by using cues from all feature locations. *Yu et al.* [34] proposed QANet for reading comprehension, whose encoder uses local convolution and self-attention to encode input sentences. *Lee et al.* [77] used self-attention networks to generate a pre-trained biomedical language representation model named BioBERT, which can understand complex biomedical texts.

In our work, we fully apply the self-attention mechanism to the code search field to alleviate the semantic gap between data from the different modal. And our experimental results suggest that this attempt is valuable for the code search.

7. Conclusion

In this paper, we proposed a self-attention network-based code search model, named SAN-CS. Instead of using LSTM or CNN, we first directly utilize self-attention networks to represent code snippets and their queries, and then utilize a joint representation network to conduct an extra joint representation for the code and query vectors. By completing these steps, SAN-CS can learn a joint representation both for code snippets and their queries. Our experimental results demonstrate that SAN-CS outperforms DeepCS and CARLCS-CNN in MRR metric. Moreover, SAN-CS has faster execution efficiency than DeepCS. Therefore, self-attention networks and joint representation learning are suitable for the deep learning-based code search methods.

Our future works mainly focus on two aspects. On the one hand, we will try to explore the structure information of the code snippet, aiming at combining it with the semantic information of the code snippet and further improve the performance of our proposed model. On the other hand, we plan to apply the self-attention network to other software engineering tasks such as code summary, which also may benefit from this more effective feature extraction network.

CRediT authorship contribution statement

Sen Fang: Writing - review & editing, Methodology, Software, Data curation. **You-Shuai Tan:** Software, Conceptualization, Visualization. **Tao Zhang:** Conceptualization, Visualization, Writing - review & editing, Supervision. **Yepang Liu:** Writing - review & editing.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2021.106542.

Acknowledgment

Many thanks for all reviewers' valuable suggestions and comments. This work is supported by the Science and Technology Development Fund of Macau, Macau SAR (No. 0047/2020/A1), the China Postdoctoral Science Foundation, China (No. 2017M621247), and Faculty Research Grant Projects of MUST, Macau SAR (No. FRG-20-008-FI).

References

[1] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, S.R. Klemmer, Two studies of opportunistic programming: interleaving web foraging, learning, and writing code, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2009, pp. 1589–1598.
[2] K. Kevic, T. Fritz, Automatic search term identification for change tasks, in: Companion Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 468–471.
[3] M.P. Robillard, What makes APIs hard to learn? Answers from developers, IEEE Softw. 26 (6) (2009) 27–34.
[4] J. Brandt, M. Dontcheva, M. Weskamp, S.R. Klemmer, Example-centric programming: integrating web search into the development environment, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2010, pp. 513–522.
[5] K. Kim, D. Kim, T.F. Bissyandé, E. Choi, L. Li, J. Klein, Y.L. Traon, FaCoY: a code-to-code search engine, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 946–957.
[6] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, P. Baldi, Sourcerer: mining and searching internet-scale software repositories, Data Min. Knowl. Discov. 18 (2) (2009) 300–336.
[7] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, J. Zhao, Codehow: Effective code search based on api understanding and extended boolean model (e), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 260–270.
[8] T.J. Biggerstaff, B.G. Mitbander, D.E. Webster, Program understanding and the concept assignment problem, Commun. ACM 37 (5) (1994) 72–82.
[9] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, Q. Xie, Exemplar: A source code search engine for finding highly relevant applications, IEEE Trans. Softw. Eng. 38 (5) (2011) 1069–1087.
[10] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, X. Luo, Rosf: Leveraging information retrieval and supervised learning for recommending code snippets, IEEE Trans. Serv. Comput. 12 (1) (2016) 34–46.
[11] R. Sirres, T.F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, Y. Le Traon, Augmenting and structuring user queries to support efficient free-form code search, Empir. Softw. Eng. 23 (2018) 2622–2654.
[12] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, 2014, arXiv preprint arXiv:1409.0473.
[13] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.
[14] X. Gu, H. Zhang, S. Kim, Deep code search, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 933–944.

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

[16] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Netw. 61 (2015) 85–117.

[17] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, Y. Lei, Improving code search with co-attentive representation learning, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 196–207.

[18] W. Li, H. Qin, S. Yan, B. Shen, Y. Chen, Learning code-query interaction for enhancing code searches, in: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 115–126.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.

[20] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in Neural Information Processing Systems, 2013, pp. 3111–3119.

[21] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013, arXiv preprint arXiv:1301.3781.

[22] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: International Conference on Machine Learning, 2014, pp. 1188–1196.

[23] R. Kiros, Y. Zhu, R.R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, S. Fidler, Skip-thought vectors, in: Advances in Neural Information Processing Systems, 2015, pp. 3294–3302.

[24] Z. Lin, M. Feng, C.N.d. Santos, M. Yu, B. Xiang, B. Zhou, Y. Bengio, A structured self-attentive sentence embedding, 2017, arXiv preprint arXiv:1703.03130.

[25] M. Sahlgren, The distributional hypothesis, Italian J. Disabil. Stud. 20 (2008) 33–53.

[26] T. Mikolov, M. Karafiát, L. Burget, Jančernocky, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model, in: Eleventh Annual Conference of the International Speech Communication Association, 2010, pp. 1045–1048.

[27] H. Palangi, H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, R. Ward, Deep sentence embedding using the long short term memory network: Analysis and application to information retrieval, 2015, arXiv. org.

[28] D.J. Montana, L. Davis, Training feedforward neural networks using genetic algorithms, in: IJCAI, vol. 89, 1989, pp. 762–767.

[29] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (ARTICLE) (2011) 2493–2537.

[30] A. Frome, G.S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, T. Mikolov, Devise: A deep visual-semantic embedding model, in: Advances in Neural Information Processing Systems, 2013, pp. 2121–2129.

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

[32] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, R. Soricut, Albert: A lite bert for self-supervised learning of language representations, 2019, arXiv preprint arXiv:1909.11942.

[33] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, 2017, arXiv preprint arXiv:1710.10903.

[34] A.W. Yu, D. Dohan, M.-T. Luong, R. Zhao, K. Chen, M. Norouzi, Q.V. Le, Qanet: Combining local convolution with global self-attention for reading comprehension, 2018, arXiv preprint arXiv:1804.09541.

[35] H. Zhang, I. Goodfellow, D. Metaxas, A. Odena, Self-attention generative adversarial networks, in: International Conference on Machine Learning, 2019, pp. 7354–7363.

[36] V. Nair, G.E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th International Conference on Machine Learning, 2010, pp. 807–814.

[37] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014, arXiv preprint arXiv:1412.3555.

[38] J. Gehring, M. Auli, D. Grangier, D. Yarats, Y.N. Dauphin, Convolutional sequence to sequence learning, 2017, arXiv preprint arXiv:1705.03122.

[39] J. Gehring, M. Auli, D. Grangier, Y.N. Dauphin, A convolutional encoder model for neural machine translation, 2016, arXiv preprint arXiv:1611.02344.

[40] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT press, 2016.

[41] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.

[42] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, H. Mei, Relationship-aware code search for JavaScript frameworks, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 690–701.

[43] M. Raghothaman, Y. Wei, Y. Hamadi, Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 357–367.

[44] Y. Wang, L. Wang, Y. Li, D. He, T.-Y. Liu, A theoretical analysis of NDCG type ranking measures, in: Conference on Learning Theory, 2013, pp. 25–54.

[45] X. Ye, R. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 689–699.

[46] A. Al-Maskari, M. Sanderson, P. Clough, The relationship between IR effectiveness measures and user satisfaction, in: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 2007, pp. 773–774.

[47] E. Kanoulas, J.A. Aslam, Empirical justification of the gain and discount function for nDCG, in: Proceedings of the 18th ACM Conference on Information and Knowledge Management, 2009, pp. 611–620.

[48] S. Hochreiter, J. Schmidhuber, LSTM can solve hard long time lag problems, in: Advances in Neural Information Processing Systems, 1997, pp. 473–479.

[49] J. Liu, G. Wang, P. Hu, L.-Y. Duan, A.C. Kot, Global context-aware attention lstm networks for 3d action recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 1647–1656.

[50] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (Jan) (2003) 993–1022.

[51] T. Hofmann, Probabilistic latent semantic indexing, in: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 1999, pp. 50–57.

[52] G.F. Montúfar, J. Morton, When does a mixture of products contain a product of mixtures? SIAM J. Discrete Math. 29 (1) (2015) 321–347.

[53] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin, A neural probabilistic language model, J. Mach. Learn. Res. 3 (Feb) (2003) 1137–1155.

[54] F. Morin, Y. Bengio, Hierarchical probabilistic neural network language model., in: Aistats, vol. 5, Citeseer, 2005, pp. 246–252.

[55] J.H. Lau, T. Baldwin, T. Cohn, Topically driven neural language model, 2017, arXiv preprint arXiv:1704.08012.

[56] W. Wang, Z. Gan, W. Wang, D. Shen, J. Huang, W. Ping, S. Satheesh, L. Carin, Topic compositional neural language model, in: International Conference on Artificial Intelligence and Statistics, 2018, pp. 356–365.

[57] W.-K. Chan, H. Cheng, D. Lo, Searching connected API subgraph via text phrases, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.

[58] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, T. Menzies, Automatic query reformulations for text retrieval in software engineering, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 842–851.

[59] E. Hill, M. Roldan-Vega, J.A. Fails, G. Mallet, NL-based query refinement and contextualized code search results: A user study, in: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, 2014, pp. 34–43.

[60] Y. Ke, K.T. Stolee, C. Le Goues, Y. Brun, Repairing programs with semantic code search (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 295–306.

[61] M. Lu, X. Sun, S. Wang, D. Lo, Y. Duan, Query expansion via wordnet for effective code search, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 545–549.

[62] K.T. Stolee, S. Elbaum, D. Dobos, Solving the search for source code, ACM Trans. Softw. Eng. Methodol. 23 (3) (2014) 1–45.

[63] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, C. Fu, Portfolio: finding relevant functions and their usage, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 111–120.

[64] J. Cambronero, H. Li, S. Kim, K. Sen, S. Chandra, When deep learning met code search, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 964–974.

[65] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, P. Yu, Multi-modal attention network learning for semantic source code retrieval, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 13–25.

[66] T. Nguyen, P. Vu, T. Nguyen, Code recommendation for exception handling, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1027–1038.

[67] S. Yan, H. Yu, Y. Chen, B. Shen, L. Jiang, Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language queries, in: Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 344–354.

[68] C. Ling, Z. Lin, Y. Zou, B. Xie, Adaptive deep code search, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 48–59.

[69] C. Liu, X. Xia, D. Lo, Z. Liu, A.E. Hassan, S. Li, Simplifying deep-learning-based model for code search, 2020, arXiv preprint arXiv:2005.14373.

[70] Y. Lin, X. Peng, Z. Xing, D. Zheng, W. Zhao, Clone-based and interactive recommendation for modifying pasted code, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 520–531.

[71] Y. Lin, G. Meng, Y. Xue, Z. Xing, J. Sun, X. Peng, Y. Liu, W. Zhao, J. Dong, Mining implicit design templates for actionable code reuse, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 394–404.

[72] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, S. Zagoruyko, End-to-end object detection with transformers, 2020, arXiv preprint arXiv:2005.12872.

[73] S. Xingjian, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, W.-c. Woo, Convolutional LSTM network: A machine learning approach for precipitation nowcasting, in: Advances in Neural Information Processing Systems, 2015, pp. 802–810.

[74] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, Y. Bengio, Show, attend and tell: Neural image caption generation with visual attention, in: International Conference on Machine Learning, 2015, pp. 2048–2057.

[75] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., Google's neural machine translation system: Bridging the gap between human and machine translation, 2016, arXiv preprint arXiv:1609.08144.

[76] A.M. Rush, S. Chopra, J. Weston, A neural attention model for abstractive sentence summarization, 2015, arXiv preprint arXiv:1509.00685.

[77] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C.H. So, J. Kang, BioBERT: a pre-trained biomedical language representation model for biomedical text mining, Bioinformatics 36 (4) (2020) 1234–1240.