Improving Code Search with Co-Attentive Representation Learning

Jianhang Shuai
Chongqing University
Chongqing, China
shuaijianhang@cqu.edu.cn

Ling Xu*
Chongqing University
Chongqing, China
xuling@cqu.edu.cn

Chao Liu
Zhejiang University
Hangzhou, Zhejiang, China
liuchaoo@zju.edu.cn

Meng Yan
Chongqing University
Chongqing, China
mengy@cqu.edu.cn

Xin Xia
Monash University
Melbourne, VIC, Australia
xin.xia@monash.edu

Yan Lei
Chongqing University
Chongqing, China
yanlei@cqu.edu.cn

ABSTRACT

Searching and reusing existing code from a large-scale codebase, e.g, GitHub, can help developers complete a programming task efficiently. Recently, Gu et al. proposed a deep learning-based model (i.e., DeepCS), which significantly outperformed prior models. The DeepCS embedded codebase and natural language queries into vectors by two LSTM (long and short-term memory) models separately, and returned developers the code with higher similarity to a code search query. However, such embedding method learned two isolated representations for code and query but ignored their internal semantic correlations. As a result, the learned isolated representations of code and query may limit the effectiveness of code search.

To address the aforementioned issue, we propose a co-attentive representation learning model, i.e., Co-Attentive Representation Learning Code Search-CNN (CARLCS-CNN). CARLCS-CNN learns interdependent representations for the embedded code and query with a co-attention mechanism. Generally, such mechanism learns a correlation matrix between embedded code and query, and co-attends their semantic relationship via row/column-wise max-pooling. In this way, the semantic correlation between code and query can directly affect their individual representations. We evaluate the effectiveness of CARLCS-CNN on Gu et al.'s dataset with 10k queries. Experimental results show that the proposed CARLCS-CNN model significantly outperforms DeepCS by 26.72% in terms of MRR (mean reciprocal rank). Additionally, CARLCS-CNN is five times faster than DeepCS in model training and four times in testing.

CCS CONCEPTS

• **Software and its engineering** → *Search-based software engineering*.

KEYWORDS

code search, co-attention mechanism, representation learning

ACM Reference Format:
Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving Code Search with Co-Attentive Representation Learning. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389269

1 INTRODUCTION

During software development, developers often spend 19% of their time searching some reusable code examples to save their development efforts [6, 24, 43]. To improve the development efficiency, developers frequently search and reuse existing code from large-scale open source repositories [33, 42, 64], such as GitHub [1].

However, developing a satisfactory code search engine is challenging [11, 24, 40]. Early studies start by leveraging Information Retrieval (IR) techniques, such as the Lucene-based models Koders[1], Krugle[2], and Google code search [28, 33, 46]. But these models simply treat code and search query as plain texts as common web search engine, and miss the programming information in the context [38]. To capture programming semantics in code and query, researchers proposed many models [5, 9, 15, 17, 19, 31]. One of the representative models is Sourcerer proposed by Erik et al. [28] that integrated Lucene with code structural information. And the other is CodeHow proposed by Fei et al. [32], which recognized a user query as relevant APIs, and performed code retrieval by using an Extended Boolean model.

Nevertheless, the aforementioned models fail to bridge the semantic gaps between programming language in code and natural language in query [14, 15, 41]. To tackle this issue, Gu et al. [56] proposed a deep learning-based model called DeepCS[3] (Deep Code Search). It is one of the state-of-the-art approaches. DeepCS embedded code and query into vector spaces by two independent LSTM (long and short-term memory) models, learned a joint embedding to align two vector spaces, and finally returned the code with higher cosine similarity to an embedded search query. Their experimental results showed that DeepCS outperforms traditional models significantly [56], including Sourcerer [28] and CodeHow [32].

*Corresponding author

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389269

[1] www.koders.com
[2] www.krugle.com
[3] https://github.com/guxd/deep-code-search

In spite of the DeepCS' advantages over traditional models, we observe that its joint embedding cannot fully capture the semantic correlations between code and query. For example, in Fig. 1, DeepCS can only relate the word "file" in query to two APIs in code, "createNewFile()" and "FileWriter()", because they both contain the keyword "file". However, in developers' understanding, four other APIs should also strongly correlate to the keyword "file", including "BufferedWriter()", "write()", "flush()", and "close()". In practice, there will be semantic gaps between the words used in the task (or query) description and the APIs relevant to the task. [8, 11]. Therefore, we assume that without fully understanding such semantic correlations, DeepCS is unlikely to return an expected code to a developer's query.

```
/**
 *  query: how to append string to file
 */
protected void append (String s, File LogFile) {
    if (!LogFile.exists()) {
        LogFile.createNewFile();
    }
    try {
        FileWriter fw = new FileWriter(LogFile, true);
        BufferedWriter writer = new BufferedWriter(fw);
        writer.write(s);
        writer.flush();
        writer.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 1: The correlated words between query and code

In order to address the limitation of DeepCS, we propose a co-attentive representation learning model, called Co-Attentive Representation Learning Code Search-CNN (CARLCS-CNN). Specifically, it first embeds code and query respectively using a Convolutional Neural Network (CNN) [26] instead of LSTM used in DeepCS since CNN can capture the informative keywords in query and code than LSTM better [61], which is also verified in Section 5. Then, CARLCS-CNN learns interdependent representations for the embedded code and query by a co-attention mechanism. Generally, the co-attention mechanism learns a correlation matrix based on the embedded code and query, and leverages row/column-wise max-pooling on the matrix to enable model focusing on the strongly correlated keywords between code and query. In this way, their semantic correlations can directly influence respective representations.

To evaluate the effectiveness of the proposed CARLCS-CNN model, we conduct experiments on Gu et al.'s [56] dataset with 10k queries. Experimental results show that CARLCS-CNN significantly outperforms DeepCS by 26.72% in terms of MRR (mean reciprocal rank). Additionally, running CARLCS-CNN is 4 times faster than DeepCS in code search, and 5 times faster in model training under the same optimization settings.

In summary, the main contributions of this paper are as follows.

- We propose a new co-attentive representation learning-based model for code search named CARLCS-CNN. It learns interdependent representations for code and query by leveraging the CNN and the co-attention mechanism.
- We evaluate the effectiveness of CARLCS-CNN on a public dataset. The results show that CARLCS-CNN outperforms the state-of-the-art model DeepCS significantly.

The remainder of the paper is structured as follows. Section 2 introduces the background of code search. Section 3 presents our proposed model CARLCS-CNN in detail. Section 4 describes our experiment setup, followed by the experimental results in Section 5 and discussion in Section 6. Section 7 presents the related works and Section 8 concludes the study and presents future works.

2 BACKGROUND

Our goal is to build an improved deep learning-based model for code search. This section presents the background on how to leverage deep learning models like DeepCS to embed code and query respectively, build a code search engine with embedded code and query, and evaluate the model in the practical scenario.

2.1 Code and Query Embedding

In code search, code in programming language and query in natural language are required to be transformed into vector space so that their semantic similarity can be measured. In specific, the embedding is conducted as follows.

Code Embedding. A code like Fig. 1 can be divided into three components: 1) method name, a list of camel split tokens; 2) API sequence, a list of API words in method body; 3) tokens, a bag of words in method body. Before embedding, tokens in components are first encoded by a vocabulary with top-n frequently appeared words in context, then transformed them into vectors with the same dimension ($\boldsymbol{v}_{name}$, $\boldsymbol{v}_{API}$, $\boldsymbol{v}_{token}$). Finally, each component vector is processed by a neural network model for embedding. Usually, the method name and API sequence are embedded by an LSTM model because it memorizes the sequential relationship between words. Meanwhile, the tokens are embedded by a common multilayer perceptron (MLP) due to the bag-of-words assumption. Then, the embedded code ($\boldsymbol{v}_{code}$) can be represented by Eq. (1).

$$\boldsymbol{v}_{code} = LSTM_1(\boldsymbol{v}_{name}) + LSTM_2(\boldsymbol{v}_{API}) + MLP(\boldsymbol{v}_{token}). \quad (1)$$

Query Embedding. Similarly, the query in natural language, such as "how to append string to file" in Fig. 1, is treated as a list of words. The list is encoded by a vocabulary with a fixed length, transformed into a same dimensional vector ($\boldsymbol{v}_q$), and finally embedded by an LSTM model. Therefore, the embedding result ($\boldsymbol{v}_{query}$) can be represented by Eq. (2).

$$\boldsymbol{v}_{query} = LSTM_3(\boldsymbol{v}_q). \quad (2)$$

2.2 Deep Learning for Code Search

In code search, a deep learning-based model learns a joint embedding between code ($\boldsymbol{v}_{code}$) and query ($\boldsymbol{v}_{query}$), so that the code relevant to a query can be measured by their cosine similarity,

$$cos = \frac{\boldsymbol{v}_{code} \cdot \boldsymbol{v}_{query}}{||\boldsymbol{v}_{code}|| \, ||\boldsymbol{v}_{query}||}. \quad (3)$$

To optimize the parameters (θ) of LSTM and MLP in code/query embedding, the loss function aims to maximize the similarity between a code ($\boldsymbol{v}_{code}$) and a relevant query ($\boldsymbol{v}^{+}_{query}$) while minimize the similarity with an irrelevant query ($\boldsymbol{v}^{-}_{query}$). Therefore, the loss function can be denoted by Eq. (4), where β is a small margin constraint.

$$L(\theta) = max(0, \beta - cos(\boldsymbol{v}_{code}, \boldsymbol{v}^{+}_{query}) + cos(\boldsymbol{v}_{code}, \boldsymbol{v}^{-}_{query})). \quad (4)$$

In the model training, Gu et al.[56] extracted over 18 million commented Java code methods from GitHub to generate query-code pairs, where the first line in Javadoc comment is regarded as a query.

2.3 Evaluating Code Search

To simulate the code search in practices, for a developer's query, a code search model recommends a list of code from the codebase. Then the model effectiveness can be estimated by analyzing the rank of the code relevant to the query. In DeepCS' dataset shared by Gu et al. [56], there are more than 10k queries with corresponding ground-truth code.

3 THE PROPOSED MODEL: CARLCS-CNN

In this section, we present the design and implementation details of our proposed model.

3.1 Overall Structure

Fig. 2 illustrates the overall structure of the proposed model CARLCS-CNN. This model takes a commented code as model inputs. Then, it performs individual embedding on code and corresponding description (i.e., the comment) respectively, where code components (method name, API sequence, and tokens) are processed separately. Afterwords, CARLCS-CNN learns co-attentive representations.

The following subsections present the model details. Specifically, Section 3.2 and 3.3 describe the individual embedding for code and description respectively. Section 3.4 presents the co-attentive representation learning for embedded code and query. Section 3.5 and 3.6 provide the details on model optimization and testing separately.

3.2 Code Embedding

Each code consists of three features: method name, API sequence, and tokens. We implement the code embedding according to the following four steps:

3.2.1 Method Name Embedding. The word sequence for method name is extracted by splitting it on the camel-case[4]. For example, the method name "readFile" is split into words "read" and "file". It is easy to find that the size of the split method name is short. We empirically find that the average length of each method name sequence is 2.3 in our training data. Therefore it is difficult to utilize LSTM to extract the sequential features for each method name. However, the split method name sequence is a brief but exhaustive summarization of the code's functionality, which means that the method name sequence contains the abstract semantic features of the code. As CNN is supposed to be good at extracting robust and abstract features, we carry out the method name embedding through CNN instead of LSTM.

Let $\boldsymbol{m}_i \in \mathbb{R}^k$ be the k-dimensional word vector corresponding to the i-th word in the method name sequence. A sequence of length n is represented as

$$\boldsymbol{m}_{1:n} = \boldsymbol{m}_1 \oplus \boldsymbol{m}_2 \oplus ... \oplus \boldsymbol{m}_n, \quad (5)$$

where $\oplus$ is the concatenation operator. In general, let $\boldsymbol{m}_{i:i+j}$ refers to the concatenation of words $\boldsymbol{m}_i$, $\boldsymbol{m}_{i+1}$, ..., $\boldsymbol{m}_{i+j}$. A convolution operation involves a *filter* $\mathrm{W}_M \in \mathbb{R}^{k \times h}$, which is applied to a window of h words to produce a feature. For example, a feature $\boldsymbol{c}_i$ is generated from a window of words $\boldsymbol{m}_{i:i+h-1}$ by

$$\boldsymbol{c}_i = f(W_M * \boldsymbol{m}_{i:i+h-1} + \boldsymbol{b}), \quad (6)$$

where $\boldsymbol{b} \in \mathbb{R}$ is a bias term, $*$ is the convolution operator and f is a non-linear function such as the hyperbolic tangent. This filter is applied to each possible window of words in the method name sequence $\boldsymbol{m}_{1:h}$, $\boldsymbol{m}_{2:h+1}$, ..., $\boldsymbol{m}_{n-h+1:n}$ to produce a *feature map*.

$$M_h = [\boldsymbol{c}_1, \boldsymbol{c}_2, ..., \boldsymbol{c}_{n-h+1}]. \quad (7)$$

We have described the process by which *one* feature is extracted from *one* filter. Our model use three types of filters with varying window size h from 2 to 4. The number of each type of filter is d. We implement the convolution operation through these filters to extract three distinctive feature maps namely M_{h_1}, M_{h_2}, M_{h_3} $\in \mathbb{R}^{d \times (n-h+1)}$, respectively. Then, three feature maps are concatenated into a summarized one. In other words, method name is finally embedded to a feature matrix M:

$$M = M_{h_1} \oplus M_{h_2} \oplus M_{h_3}. \quad (8)$$

3.2.2 Tokens Embedding. Tokens are bags of words that are parsed from the method body. We notice that duplicate words, stop words, and Java keywords have been removed during the data preprocessing, which means, tokens are the informative keywords of code. According to the above considerations, tokens embedding is implemented by using CNN.

A set of k-dimensional tokens of length n are concatenated as $\boldsymbol{t}_{1:n}$. We use three types of filters $\mathrm{W}_T \in \mathbb{R}^{k \times h}$ with various window size h from 2 to 4 to perform the convolution. The number of each type of filter is d. Three types of filters are applied to a window of h words to derive three feature maps T_{h_1}, T_{h_2}, $\mathrm{T}_{h_3} \in \mathbb{R}^{d \times (n-h+1)}$. Likewise, tokens are eventually embedded into a feature matrix T:

$$\boldsymbol{c}_i = f(W_T * \boldsymbol{t}_{i:i+h-1} + \boldsymbol{b}), \quad (9)$$

$$T_h = [\boldsymbol{c}_1, \boldsymbol{c}_2, ..., \boldsymbol{c}_{n-h+1}], \quad (10)$$

$$T = T_{h_1} \oplus T_{h_2} \oplus T_{h_3}. \quad (11)$$

3.2.3 API Sequences Embedding. Considering the dynamic sequential features of the API sequence, we implement LSTM to do the embedding. $\boldsymbol{a}_i \in \mathbb{R}^k$ is the k-dimensional word vector of the i-th word in the API sequence. An API sequence of length n is concatenated as $\boldsymbol{a}_{1:n}$. The hidden state $\boldsymbol{h}_i \in \mathbb{R}^d$ denotes the representation of time step i can be obtained by the bi-directional LSTM. Here, d is the units of each hidden state. Normally, in the bi-directional LSTM, the hidden state $\overrightarrow{\boldsymbol{h}_i}$ of the forward LSTM is updated by considering its former memory cell $\overrightarrow{\boldsymbol{c}_{i-1}}$, the preceding hidden state $\overrightarrow{\boldsymbol{h}_{i-1}}$ and

[4]Camel case, https://en.wikipedia.org/wiki/camelcase

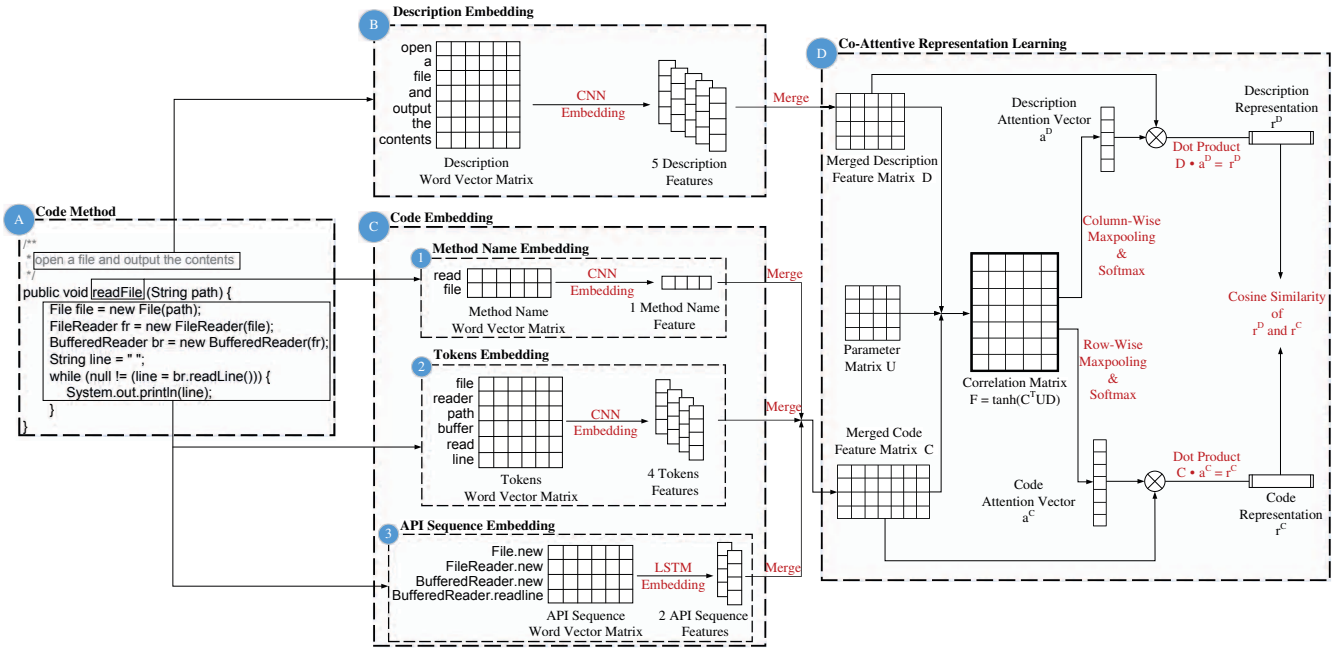

Figure 2: overall structure of CARLCS-CNN

the current input word vector $\boldsymbol{a}_i$. Meanwhile, the hidden state $\overleftarrow{\boldsymbol{h}_i}$ of the backward LSTM is formed by its later memory cell $\overleftarrow{\boldsymbol{c}_{i+1}}$, next hidden state $\overleftarrow{\boldsymbol{h}_{i+1}}$ and the input word vector $\boldsymbol{a}_i$. We can separately formulate the hidden stats as follows:

$$\overrightarrow{\boldsymbol{h}_i} = \tanh\left(\overrightarrow{\boldsymbol{c}_{i-1}}, \overrightarrow{\boldsymbol{h}_{i-1}}, \boldsymbol{a}_i\right), \tag{12}$$

$$\overleftarrow{\boldsymbol{h}_i} = \tanh\left(\overleftarrow{\boldsymbol{c}_{i+1}}, \overleftarrow{\boldsymbol{h}_{i+1}}, \boldsymbol{a}_i\right). \tag{13}$$

Then, the ultima hidden state $\boldsymbol{h}_i$ of time step i is the concatenation of both forward LSTM and backward LSTM:

$$\boldsymbol{h}_i = \overrightarrow{\boldsymbol{h}_i} \oplus \overleftarrow{\boldsymbol{h}_i}. \tag{14}$$

Finally, the API sequence is embedded by concatenating all the output hidden states to a feature matrix $\mathrm{A} \in \mathbb{R}^{d \times n}$:

$$A = \boldsymbol{h}_1 \oplus \boldsymbol{h}_2 \oplus \ldots \oplus \boldsymbol{h}_n, \tag{15}$$

where n is the number of hidden states.

3.2.4 Code Features Fusion. After embedding three code features to three matrixes. We eventually merge them into one matrix $\mathrm{C} \in \mathbb{R}^{d \times p}$ as the feature matrix for code:

$$C = M \oplus T \oplus A. \tag{16}$$

3.3 Description Embedding

We find that the length of the description is usually short. In Section 4.2, we perform statistical research over the 10k evaluation data and find that 95.48% of the descriptions contain no more than 20 words. However, the description contains informative keywords that reflect the intention of the developers. Under the circumstances, we implement CNN in description embedding. $\boldsymbol{d}_i \in \mathbb{R}^k$ is the k-dimensional word vector corresponding to the i-th word in the description. A description of length n is represented as $\boldsymbol{d}_{1:n}$. The same as method name embedding and tokens embedding, we use three types of filters $\mathrm{W}_D \in \mathbb{R}^{k \times h}$, whose number is d, to get three corresponding feature maps $\mathrm{D}_{h_1}, \mathrm{D}_{h_2}, \mathrm{D}_{h_3} \in \mathbb{R}^{d \times (n-h+1)}$. The window size h of each type of filter is varied from 2 to 4. Description embedding is accomplished by merging three feature maps into one feature matrix $\mathrm{D} \in \mathbb{R}^{d \times q}$:

$$\boldsymbol{c}_i = f(W_D * \boldsymbol{d}_{i:i+h-1} + \boldsymbol{b}), \tag{17}$$

$$D_h = [\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_{n-h+1}], \tag{18}$$

$$D = D_{h_1} \oplus D_{h_2} \oplus D_{h_3}. \tag{19}$$

3.4 Co-Attention Mechanism

After embedding code and its paired description, we can get two feature matrices $\mathrm{C} \in \mathbb{R}^{d \times p}$ and $\mathrm{D} \in \mathbb{R}^{d \times q}$ for code and description. Here p and q represent the sizes of the embedded code and description, respectively. By introducing a parameter matrix $\mathrm{U} \in \mathbb{R}^{d \times d}$, which is to be learned by neural networks, we compute the correlation matrix $\mathrm{F} \in \mathbb{R}^{p \times q}$ as follows:

$$F = \tanh(C^T U D). \tag{20}$$

The correlation matrix F is able to have a co-attentive sight on embedded code and description's words semantic correlations. Note that, each element $\mathrm{F}_{i,j}$ in F represents the semantic correlation between two aligned vectors, i.e., C_i for the i-th code word and D_j for the j-th description word. Specifically, the i-th row in F represents the semantic correlations of each word in description to the i-th code word. Likewise, the j-th column in F represents the semantic correlations of each word in code to the j-th description word.

Next, we conduct max-pooling operations along rows and columns over F to obtain semantic vectors $\boldsymbol{g}^C \in \mathbb{R}^p$ and $\boldsymbol{g}^D \in \mathbb{R}^q$ for code and description. The i-th element of $\boldsymbol{g}^C$ represents *importance score* between the i-th word in code C and its most relevant word in description D. Likewise, the i-th element of $\boldsymbol{g}^D$ represents *importance score* between the i-th word in description D and its most relevant word in code C. According to our experiments, max-pooling performs better than mean-pooling. This can be interpreted that max-pooling can capture the most important semantic correlation (one with the highest value) between each word in code and description. Thus, we employ max-pooling operation as follows:

$$\begin{aligned} \boldsymbol{g}_i^C &= maxpooling[F_{i,1}, ..., F_{i,q}], \\ \boldsymbol{g}_i^D &= maxpooling[F_{1,i}, ..., F_{p,i}]. \end{aligned} \tag{21}$$

The semantic vectors $\boldsymbol{g}^C$ and $\boldsymbol{g}^D$ are obtained as follows:

$$\begin{aligned} \boldsymbol{g}^C &= [\boldsymbol{g}_1^C, ..., \boldsymbol{g}_p^C]^T, \\ \boldsymbol{g}^D &= [\boldsymbol{g}_1^D, ..., \boldsymbol{g}_q^D]^T. \end{aligned} \tag{22}$$

After that, we employ softmax function on semantic vectors $\boldsymbol{g}^C$ and $\boldsymbol{g}^D$ to generate attention vectors $\boldsymbol{a}^C \in \mathbb{R}^p$ and $\boldsymbol{a}^D \in \mathbb{R}^q$ for code and description. The softmax function transforms the j-th element $\boldsymbol{g}_j^C$ and $\boldsymbol{g}_j^D$ to the attention ratio $\boldsymbol{a}_j^C$ and $\boldsymbol{a}_j^D$. For instance, the j-th element in $\boldsymbol{a}^C$ and $\boldsymbol{a}^D$ are computed as follows:

$$\begin{aligned} \boldsymbol{a}_j^C &= \frac{\exp(\boldsymbol{g}_j^C)}{\sum_{l=1}^{p} \exp(\boldsymbol{g}_l^C)}, \\ \boldsymbol{a}_j^D &= \frac{\exp(\boldsymbol{g}_j^D)}{\sum_{l=1}^{q} \exp(\boldsymbol{g}_l^D)}. \end{aligned} \tag{23}$$

Finally, we implement dot product between the feature matrices C, D and attention vectors $\boldsymbol{a}^C$, $\boldsymbol{a}^D$ to generate co-attentive representations $\boldsymbol{r}^C \in \mathbb{R}^d$ and $\boldsymbol{r}^D \in \mathbb{R}^d$ for code and description, respectively:

$$\begin{aligned} \boldsymbol{r}^C &= C\boldsymbol{a}^C, \\ \boldsymbol{r}^D &= D\boldsymbol{a}^D. \end{aligned} \tag{24}$$

3.5 Model Optimization

Now we describe how to train CARLCS-CNN to learn co-attentive representations for code and description. The basic assumption is to learn a mapping that produces more similar representations for description and the corresponding code, which moves the representations for description and correct code closer together while minimizing the pairwise ranking loss:

$$L(\theta) = \sum_{\langle c, d^+, d^- \rangle \in G} \max(0, \beta - sim(\boldsymbol{c}, \boldsymbol{d}^+) + sim(\boldsymbol{c}, \boldsymbol{d}^-)), \tag{25}$$

where θ denotes the model parameters, G denotes the training dataset. For each code snippet $\boldsymbol{c}$, there is a positive description $\boldsymbol{d}^+$ (a correct description of $\boldsymbol{c}$) and a negative description $\boldsymbol{d}^-$ (an incorrect description of $\boldsymbol{c}$) randomly chosen from the pool of $\boldsymbol{d}^+$'s. *sim* denotes the similarity score between code and description. β is a small margin constraint. We conduct the cosine similarity measure and set the fixed β value to 0.05.

we use the Adam algorithm [25] to minimize the loss function. At training time, the co-attention mechanism learns the similarity measure over the representations of code and description. Such a similarity measure is used to compute attention vectors for code and description in both directions. Next, the attention vectors are used to guide the pooling layer to perform column-wise and row-wise maxpooling over correlation matrix F. In the gradients descent phase, the model parameters θ is updated by back propagation and the representations $\boldsymbol{r}^C$ and $\boldsymbol{r}^D$ for code and description are learned simultaneously.

3.6 Model Prediction for Code Search

After the model optimization, we can deploy CARLCS-CNN online for code search by embedding a large-scale codebase, where each code is represented by a vector ($\boldsymbol{c}$). For a developer's search query, such as "how to append string to file", the CARLCS-CNN model embeds the query as a vector ($\boldsymbol{q}$). Then, the semantic similarity between a query ($\boldsymbol{q}$) and a code ($\boldsymbol{c}$) can be measured by their cosine similarity, as Eq. (26). Finally, the model recommends the top-k code highly related to the query for code search.

$$cos(\boldsymbol{c}, \boldsymbol{q}) = \frac{\boldsymbol{c}^T \boldsymbol{q}}{\|\boldsymbol{c}\| \|\boldsymbol{q}\|} \tag{26}$$

4 EXPERIMENT SETUP

This section presents five investigated research questions, our experimental dataset, the compared baseline models, and two widely used evaluation measures.

4.1 Research Questions

This study investigates the following five research questions (RQs):

RQ1. How effective is our proposed CARLCS-CNN?

The first RQ investigates whether the proposed model CARLCS-CNN outperforms the state-of-the-art code search model DeepCS [56]. If CARLCS-CNN shows advantages over DeepCS, then the co-attentive representations learned by CARLCS-CNN is beneficial for code search.

RQ2. How efficient is our proposed CARLCS-CNN?

RQ2 compares the training and testing time between our CARLCS-CNN and the DeepCS, and tests if the proposed model can save computation resources substantially. Faster model indicates more valuable application in practices.

RQ3. How does the CNN component affect the model effectiveness?

As described in Section 3.2 and 3.3, we utilized the CNN to embed code and description instead of the LSTM used by DeepCS. This RQ aims to evaluate if CNN can better understand word semantics in query and code, comparing with LSTM.

RQ4. How do three code features affect the model effectiveness?

Table 1: Thirty examples of 10k queries in the automatic evaluation

No.	Description	No.	Description
1	start the statistics timer	16	get header portion of packet
2	handle a key pressed event	17	sort an int array into ascending order
3	convert a string to char	18	release a direct buffer
4	close a writer object	19	write an attribute with the prefix
5	read a string from the stream	20	check if list contains the given attribute name
6	delete a folder on the hdfs	21	concatenate two byte arrays
7	create a new instance	22	log the details of a file
8	initialize the standard objects	23	create the ancestor listener
9	read the next line	24	initialize this key store from the provided input stream
10	extract url of the request	25	load the activities
11	format a label as a title	26	unwrap a key
12	retrieve an email list	27	print out the classifier
13	build a new cache	28	filter an error event
14	parse a version string	29	close the proxy
15	download json data from url	30	test if the file exists

In the CARLCS-CNN, a code method is respectively represented by three features (i.e., method name, API sequence, and tokens), as detailed in Section 3.2. To analyze their impacts on model effectiveness, we run CARLCS-CNN with individual feature separately, and investigate whether using these three features together is the best choice.

RQ5. How do different parameter settings affect the model effectiveness?

CARLCS-CNN contains two important parameters that affect the model effectiveness substantially. One is the number of filters in CNN that learns deep representations from code and query, as described in Section 3.2 and 3.3. The other is the description length that determines how much information in code description can be used for model optimization, as shown in Section 3.3.

4.2 Dataset

Following Gu et al. [56], we conducted experiments on their training and testing data[5]. The training data contains 18,233,872 commented Java code methods from GitHub repositories created from August, 2008 to June, 2016 with at least one star. In Gu et al.'s [56] testing data, there are 10,000 code-query pairs. To evaluate CARLCS-CNN, the 10k queries are used as model inputs while the corresponding codes are regarded as the ground-truth. This automatic evaluation can avoid the bias in manual checking and ensure the testing scale. Table 1 shows 30 query examples and Fig. 3 illustrates two corresponding code examples. Table 2 presents the distribution of word count in queries. We can observe that 95.48% of queries contain no more than 20 words because developers prefer to informative keywords for code search [46].

4.3 Baseline Models

This study compares the following models:

DeepCS, the state-of-the-art model proposed by Gu et al.[56] that embeds code and query separately with two LSTM models, as described in Section 2. We re-ran the DeepCS by using the source code shared on the GitHub.

[5]https://pan.baidu.com/s/1U_MtFXqq0C-Qh8WUFAWGvg

Table 2: Word statistics of 10k queries in the automatic evaluation

#Words in Query	#Query	%Query
1-20	9,548	95.48%
20-40	399	3.99%
40-60	30	0.30%
60-80	23	0.23%

```
/**
 * convert a string to a char
 */
public void stringToChar (String str, char[] ch) {
    int j = 0;
    for (int i = 0; i < str.length(); i++)
    {
        ch[i] = str.charAt(j++);
    }
}
```

(a) The code method of the query "convert a string to a char"

```
/**
 * close a writer object
 */
private void closeWriter (Writer objWriter) {
    if (objWriter == null) return;
    try {
        objWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

(b) The code method of the query "close a writer object"

Figure 3: Two code examples with queries in the automatic evaluation

CARLCS-LSTM, the proposed model that incorporates the co-attentive representation learning (as detailed in Section 3.4) to DeepCS with the same LSTM embedding as DeepCS.

CARLCS-CNN, the proposed model that replaces the LSTM model in CARLCS-LSTM by the CNN model and uses the same co-attention mechanism as detailed in Section 3.4. CARLCS-CNN is the full solution to our model proposed in Section 3.

4.4 Evaluation Metrics

To evaluate the effectiveness of the proposed model CARLCS-CNN, we utilize two common evaluation metrics, recall and MRR (mean reciprocal rank). Details show as follows.

Recall@k, the proportion of queries that the relevant code method could be found in the top-k ranked lists. In specific, Recall@k is calculated as follows:

$$Recall@k = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\sigma\left(Q_i \le k\right), \tag{27}$$

where Q is the 10,000 queries in our automatic evaluation, as referred to in Section 4.2; σ is an indicator function that returns 1 if

the i-th query (Q_i) could be found in the top-k ranked list, otherwise it returns 0. Following Gu et al. [56], we evaluate Recall@1, Recall@5 and Recall@10 respectively.

MRR, the average of the reciprocal ranks of all queries. The computation process of MRR is:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{Rank_{Q_i}}, \tag{28}$$

where Q is the 10k queries in the automatic evaluation; $Rank_{Q_i}$ is the rank of the ground-truth code related to the i-th query (Q_i) in the ranked list. Different from recall, MRR uses the reciprocal rank as the weight of measurement. Meanwhile, as developers prefer to find the expected code method with short code inspection, we only test MRR on the top-10 ranked list following Gu et al. [56]. In other words, when the rank of Q_i is out of 10, then $1/Rank_{Q_i}$ equals to 0.

5 RESULTS

This section presents the experimental results on the five research questions, as described in Section 4.1, in sequence.

5.1 RQ1: Model Effectiveness

Table 3 compares the code search effectiveness between the state-of-the-art model DeepCS and our CARLCS-CNN model described in Section 3. Results show that DeepCS obtains an MRR of 0.408, a Recall@1/5/10 of 0.413/0.591/0.683 respectively. CARLCS-CNN achieves an MRR of 0.517 and a Recall@1/5/10 of 0.528/0.698/0.773. Our proposed CARLCS-CNN improves DeepCS by 27.84%, 18.10%, 13.17%, and 26.72% in terms of Recall@1, Recall@5, Recall@10, and MRR. Furthermore, to analyze the statistical difference between CARLCS-CNN and DeepCS, we apply the Wilcoxon signed-rank test [55] on MRR between CARLCS-CNN and DeepCS at a 5% significance level. The p-value is less than 0.01, indicating the improvements of CARLCS-CNN over DeepCS are substantial in statistical significance. These results indicate that the co-attentive representation learning is beneficial for code search.

Table 3: Effectiveness comparison of DeepCS and CARLCS-CNN in terms of Recall@1/5/10 and MRR

Model	Recall@1	Recall@5	Recall@10	MRR
DeepCS	0.413	0.591	0.683	0.408
CARLCS-CNN	**0.528**	**0.698**	**0.773**	**0.517**

Result 1: *Our proposed CARLCS-CNN improves DeepCS in terms of MRR and Recall@1/5/10 substantially.*

5.2 RQ2: Model Efficiency

Table 4 compares the training and testing time on Gu et al.'s [56] dataset. The efficiency comparison is conducted under the same experimental setup. Results show that DeepCS takes about 50 hours for optimization and 1.2 seconds for responding each code search query. Meanwhile, the proposed model CARLCS-CNN takes 10 hours for training and 0.3 second for each query. Thus, comparing with DeepCS, CARLCS-CNN is 5 times faster in model training and 4 times faster in model testing. These results imply that CARLCS-CNN is a better choice considering practical usage. CARLCS-CNN is faster because it is a CNN-based model. Its network structure is simpler than LSTM-based DeepCS so that the whole working process can be faster [3, 59, 61, 62]. All the experiments are implemented on a server with one Nvidia Titan V GPU with 256 GB memory.

Table 4: Time cost for model training and testing of DeepCS and CARLCS-CNN

Model	Training	Testing
DeepCS	50 hours	1.2s/query
CARLCS-CNN	10 hours	0.3s/query

Result 2: *Comparing with DeepCS, CARLCS-CNN is 5 times faster in model training and 4 times faster in model testing.*

5.3 RQ3: The Impact of CNN Component

CNN is an important component to embed code and query for CARLCS-CNN as shown in Section 3. Different from DeepCS, we replace LSTM by CNN. Because we assume that CNN can better capture the informative words in query and code. To investigate the impact of the aforementioned replacement on the model effectiveness, we also implement a Co-Attentive Representation Learning model with the original LSTM as DeepCS. We name such a model as CARLCS-LSTM.

Table 5 shows that CARLCS-LSTM achieves an MRR of 0.482, a Recall@1/5/10 of 0.490/0.661/0.741, respectively. We can also notice that the MRR of CARLCS-LSTM decreases by 6.77% comparing with CARLCS-CNN, indicating that combining CNN embedding with co-attentive representation learning can further enhance the code search effectiveness. Moreover, we can observe that although CARLCS-LSTM and DeepCS share the same embedding framework, the co-attentive representation in CARLCS-LSTM shows advantageous, whose MRR outperforms that of DeepCS by 18.14%.

Table 5: Effectiveness comparison of CARLCS-LSTM and CARLCS-CNN in terms of Recall@1/5/10 and MRR

Model	Recall@1	Recall@5	Recall@10	MRR
CARLCS-LSTM	0.490	0.661	0.741	0.482
CARLCS-CNN	**0.528**	**0.698**	**0.773**	**0.517**

Result 3: *For Co-Attentive Representation Learning-based code search, CNN is a better choice for word embedding than LSTM.*

5.4 RQ4: The Impact of Code Features

All the compared models DeepCS, CARLCS-LSTM, and CARLCS-CNN use three code features as their inputs, including method name

(M), API sequence (A), and tokens (T). To investigate the relative importance of these three features, we ran three models with one individual feature at a time. From Table 6, we can observe that the three models show similar results. Specifically, when only using one feature as model input, their effectiveness decrease substantially, where the MRR of DeepCS (M/A/T) drops by more than 26.72% comparing with DeepCS (M+A+T) from 0.408; the MRR of CARLCS-LSTM (M/A/T) decreases by at least 30.71% vs. CARLCS-LSTM (M+A+T) from 0.482; and the MRR of CARLCS-CNN (M/A/T) slides by over 27.66% comparing with CARLCS-CNN (M+A+T) from 0.517. Hence, combining three code features is advantageous over using only one. Also, we can notice that the feature M (method name) affects the model effectiveness most. This is because method name is a brief summary of a code and it usually uses the same words as query [20, 38].

Furthermore, to investigate the necessity of these three features, Table 7 shows the sensitivity analysis on them by removing one feature at a time. We can observe that all features are beneficial to the model effectiveness because when removing the features on the method name, API sequence, and tokens, the MRR decreased by 39.07%, 20.12%, and 11.80% respectively.

Table 6: Effectiveness comparison of DeepCS, CARLCS-LSTM, and CARLCS-CNN with four different feature settings (M, method name; A, API sequence; T, tokens) in terms of Recall@1/5/10 and MRR

Model	Recall@1	Recall@5	Recall@10	MRR
DeepCS(M)	0.297	0.426	0.498	0.299
DeepCS(A)	0.118	0.223	0.368	0.169
DeepCS(T)	0.177	0.300	0.387	0.186
DeepCS(M+A+T)	**0.413**	**0.591**	**0.683**	**0.408**
CARLCS-LSTM(M)	0.334	0.484	0.571	0.334
CARLCS-LSTM(A)	0.132	0.252	0.369	0.184
CARLCS-LSTM(T)	0.237	0.391	0.484	0.245
CARLCS-LSTM(M+A+T)	**0.490**	**0.661**	**0.741**	**0.482**
CARLCS-CNN(M)	0.375	0.523	0.601	0.374
CARLCS-CNN(A)	0.144	0.261	0.382	0.213
CARLCS-CNN(T)	0.285	0.442	0.533	0.290
CARLCS-CNN(M+A+T)	**0.528**	**0.698**	**0.773**	**0.517**

Table 7: Sensitivity analysis of CARLCS-CNN on three different features (M, method name; A, API sequence; T, tokens) in terms of Recall@1/5/10 and MRR

Model	Recall@1	Recall@5	Recall@10	MRR
CARLCS-CNN(A+T)	0.309	0.479	0.574	0.315
CARLCS-CNN(M+A)	0.416	0.582	0.670	0.413
CARLCS-CNN(M+T)	0.465	0.637	0.717	0.456
CARLCS-CNN(M+A+T)	**0.528**	**0.698**	**0.773**	**0.517**

Result 4: *It is necessary to use three code features (method name, API sequence, and tokens) together for model inputs, while the method name affects the effectiveness most.*

5.5 RQ5: The Impact of Parameter Settings

In the CNN of CARLCS-CNN, the description length and the number of filters are two important parameters influence the code search effectiveness, as described in Section 3. The length determines how much information in description is considered for model optimization. Fig. 4 shows that the MRR of CARLCS-CNN reaches the optimum when the length is set to 60. As increasing the length will no longer improve the effectiveness but raise the model complexity, 60 is the best choice for description length. As to the DeepCS, we can observe that the ideal description length is 30, but not like CARLCS-CNN adding the length has a negative effect on model effectiveness. This comparison implies CARLCS-CNN's robustness to the noises in the description. Fig. 5 presents the effectiveness of CARLCS-CNN with various number of filters in CNN. We can observe that, in most cases, CARLCS-CNN shows a stable effectiveness regardless of the dramatically increased number of filters, but it obtains the best effectiveness when the number of filters is 250. Therefore, choosing an appropriate number of filters is necessary for model optimization.

Result 5: *For CARLCS-CNN, the best choice for description length is 60 while setting the number of filters to 250 is beneficial to code search effectiveness.*

6 DISCUSSION

This section first discusses the strength of the proposed CARLCS-CNN model with examples in Section 6.1, followed by the threats to model validity in Section 6.2.

6.1 Why does CARLCS-CNN work?

Fig. 6 shows the first retrieved result of our CARLCS-CNN vs. DeepCS for the query "detect network connection status". We can notice that in Fig. 6 (a) DeepCS returned an irrelevant code, although its method name and API contain the keyword "connection" in query. However, the DeepCS' simple joint embedding between code and query is not enough. In contrast, our CARLCS-CNN can retrieve the expected code in Fig. 6 (b) because it involves a lot of query related programming words, such as "checkConnect", "socket", "host", and etc. This result implies that the co-attentive representation learning is advantageous over simple joint embedding like DeepCS.

6.2 Threats to Validity

The proposed model CARLCS-CNN may suffer from three validity. One is the baseline replication. Replicating the state-of-the-art model DeepCS may contain some errors. To mitigate this threat, we re-ran the source code provided by the author in GitHub with shared training and testing data. The second threat is the evaluation metrics. In the experiment, we did not use the commonly used metrics like precision and NDCG (Normalized Discounted Cumulative Gain) [40, 52], because the automatic evaluation only has one ground-truth code for each query. However, manually identifying other ground-truth code will not only cost many efforts but also introduce human bias. To make a fair comparison, we use the other two standard metrics including recall and MRR as Gu et al. [56].

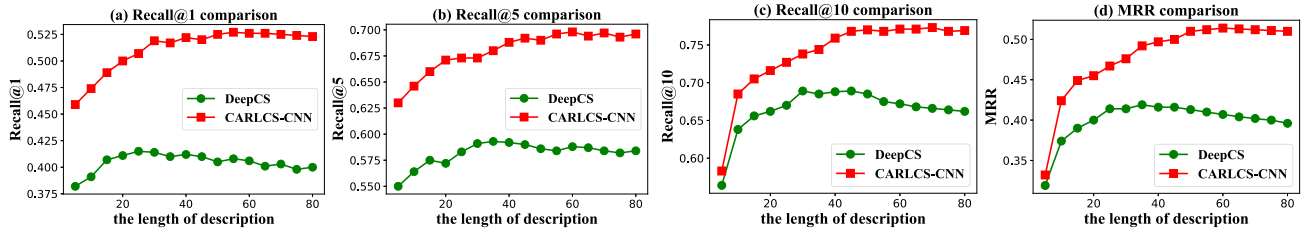

Figure 4: The effectiveness comparison of CARLCS-CNN and DeepCS with various description length in terms of Recall@1/5/10 and MRR

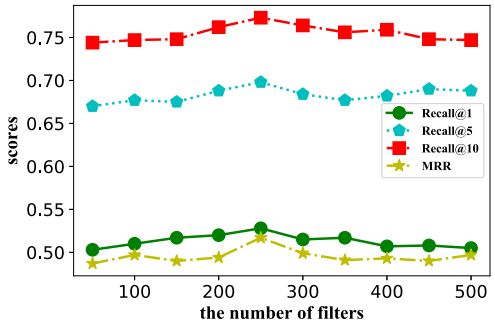

Figure 5: The effectiveness of CARLCS-CNN with various number of filters in CNN

The third threat lies in the model generalizability. The testing data provided by Gu et al. [56] only involve about 10k Java open-source projects from GitHub. The experimental results and conclusions may be different for enterprise projects or open source projects in other programming languages. We plan to extend the dataset in the near future.

7 RELATED WORK

This section provides the related works on code search in Section 7.1, code embedding in Section 7.2, and attention mechanism in Section 7.3.

7.1 Code Search

Currently, many code search models have been proposed to help users find relevant code [5, 9, 15, 17–19, 23, 31–33, 39, 49]. General web search engines such as Ohloh[6], Krugle and Google can return code snippets containing the keywords (or Regular Expressions) specified in a query. The major limitation of these general web search engines is that they simply treat code and search query as plain texts so they cannot capture the programming semantics in context and support specific programming tasks [49].

To capture the programming semantics in code and query, researchers proposed many models [5, 9, 15, 17, 19, 31]. Given a natural language query, Portfolio [33] retrieved functions and their usage chains that perform the task specified in the query. Sourcerer [28] is a code search framework over open–source projects available on the Internet. It worked by extracting fine–grained structural features from source code and using Lucene to manage the search. Chan et al. [9] proposed an approach that returned code represented

[6]https://code.ohloh.net/

Figure 6: The retrieved first result of DeepCS and CARLCS-CNN for the query "detect network connection status"

```
public static Connection getConnection (String url, String user,
String password) {
    Properties p = new Properties();
    if (user != null) {
        p.setProperty("user", user);
    }
    if (password != null) {
        p.setProperty("password", password);
    }
    return getConnection(url, p);
}
```

(a) DeepCS's first retrieved result for query "detect network connection status"

```
public void checkConnect (String host, int port,  Object context) {
    if (!(context instanceof AccessContext)) {
        throw new SecurityException("Missingcontext");
    }
    AccessContext ac = (AccessContext)context;
    if (port == -1) {
        ac.check(new Socket(host, "resolve"));
    } else {
        ac.check(new Socket(host.toString() + ":" + port));
    }
}
```

(b) CARLCS-CNN's first retrieved result for query "detect network connection status"

as a graph whose nodes denote keyword-matched classes or methods and whose edges represent the invocation relationship between the nodes. CodeHow is a search engine proposed by Fei et al. [32], which recognized a user query as relevant APIs, and performed code retrieval by using an Extended Boolean model. However, the above traditional approaches fail to bridge the semantic gaps between programming language in code and the natural language in query.

In order to overcome the above issue, Gu et al. [56] proposed a deep learning-based model called DeepCS (deep code search). It is one of the state-of-the-art search engines that embedded code and query into vector spaces by two independent LSTM models. It learned a joint embedding to align two vector spaces, and finally returned the code with high cosine similarity to an embedded search query. However, DeepCS' independent embedding only learned two isolated representations for code and query but ignored their internal relationships. To address this challenge, this paper proposes an improved model called CARLCS-CNN. It learns interdependent

representations for code and query by leveraging the CNN and the co-attention mechanism. Section 5 proved the CARLCS-CNN's substantial advantages over DeepCS.

Based on the DeepCS, researchers recently proposed two improved models [50, 60]. One is the MMAN (multi-model attention network) model proposed by Wan et al. [50], which enhanced the code representation by incorporating a tree-based RNN model and forming a fusion of two code representations. The other is the CoaCor model developed by Yao et al. [60], which brought the loss function of code summarization [53] to improve the representation of query in natural language. However, both of them only focused on one representation improvement, code or query, ignoring their interaction. In this study, our CARLCS-CNN aims to learn interdependent representations for code and query. Our experimental results also show a substantial advantage over DeepCS. Therefore, we believe that it would be valuable to incorporating the benefits of MMAN and CoaCor into CARLCS-CNN. We plan to confirm this in the near future.

7.2 Code Embedding

Bilingual embedding has widely studies in natural language processing (NLP) area [2, 10, 12]. In recent years, researchers proved that it is promising to apply this technique for code embedding in software engineering [4, 21, 22, 45]. For example, Sachdev et al. [45] developed an unsupervised embedding model NCS for code search, where code/query embedding were derived from a large-scale code corpus. Meanwhile, Gu et al. [56] leveraged LSTM models to embed code and query independently, which showed an advantage over NCS on code search [7]. Other than code search, code embedding also becomes prevalent. In a code clone study, White et al. [54] proposed a learning-based detection algorithm that embedded tokens and fragments in source code. Liu et al. [29] proposed an ordinal embedding hashing, which embedded given ordinal relations among data points to learn the ranking-preserving binary codes. In code summarization, Wan et al. [51] proposed a hybrid embedding approach which consists of an LSTM model and an AST-based LSTM model to represent the lexical level and syntactic level of code. Nguyen et al. [37] embedded source code to discover program modifications and did prediction tasks such as propagating feedback. Bayou [35] is a system that used deep neural networks to synthesize code programs by embedding API calls and natural language descriptions.

7.3 Attention Mechanism

Attention mechanism allows deep learning-based models learning to focus on the most important parts in data [13, 34, 47, 48, 57, 58]. It has been successfully applied in image generation [13], image classification [34, 48], visual question answering [47, 57, 58], etc. In the NLP area, the attention mechanism also earns its value and popularity. For example, to overcome the difficulty of the long sentence translation, Bahdanau et al. [3] proposed attention-based model RNNSearch to learn an alignment over sentences. Hermann et al. [16] proposed an attention model to circumvent the bottleneck caused by fixed-width hidden vector in text reading and comprehension. And a more fine-grained attention mechanism was proposed by Rocktäschel et al. [44], who employed a word-by-word neural attention mechanism to reason about the entailment in two sentences. However, these attention approaches use separate attention distributions for each modality, such as image or sentence, neglecting their interactions [30, 36].

To address the above issue, researchers began to work on the co-attention mechanism [30, 36]. Lu et al. [30] proposed a hierarchical co-attention model for visual question answering which is able to attend to different regions of the image as well as different fragments of the question. Li et al. [27] proposed a context-aware neural network that leveraged a co-attention mechanism to characterize the degree of matching between users' contextual preferences and items' context-aware aspects. Recently, Yu et al. [63] reduced the co-attention method into two steps, self-attention for a question embedding and the question-conditioned attention for a visual embedding. We applies the co-attention mechanism to address the semantic gap between code and query. And our experiment results indicate this appliance is valuable and promising for the code search.

8 CONCLUSION AND FUTURE WORK

Developing a satisfactory code search engine for developers has long been a challenging problem. Recently, a deep learning-based code search model DeepCS provided by Gu et al. [56] is proved to be advantageous over traditional IR-based models, such as Sourcerer [28] and CodeHow [32]. Generally, DeepCS embeds code and query separately by two LSTM models and optimizes the model with combined embedding, so that the code search can be done by returning the code with higher cosine similarity to a search query. However, the individual embedding can hardly capture the interdependent relationship between code and query.

To overcome this challenge, we propose an improved model named CARLCS-CNN[7], which leverages CNN associated with a co-attention mechanism to learn interdependent representations for code and query after the individual embedding. An automatic evaluation shows that the proposed CARLCS-CNN significantly outperforms DeepCS by 26.72% in terms of MRR. Moreover, running CARLCS-CNN is 5 times faster than DeepCS on model training and 4 times faster on model testing, due to its much less complex network structure. Therefore, co-attentive representation learning is beneficial for the code search.

We consider two future works. First, we plan to enhance the code representation by leveraging tree-based embedding as Wan et al. [50] and improve the query representation by incorporating the loss function of code summarization as Yao et al. [60]. Second, we plan to improve the quality of training data, as the code comment is not always representing the ground-truth of developers' queries.

9 ACKNOWLEDGMENTS

The work described in this paper was partially supported by the National Key Research and Development Project (Grant no. 2018YFB2101201), the National Natural Science Foundation of China (Grant no. 61602504), the Fundamental Research Funds for the Central Universities (Grant no. 2019CDYGYB014) and the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021).

[7] source code can be found at: https://github.com/cqu-isse/CARLCS-CNN

REFERENCES

[1] 2019. GitHub. Retrieved November 5, 2019 from https://www.github.com

[2] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2017. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 451–462.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *computer science* 1409 (09 2014).

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[5] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

[6] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.

[7] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. 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*. 964–974.

[8] Claudio Carpineto and Giovanni Romano. 2012. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.* 44, 1, Article 1 (Jan. 2012), 50 pages. https://doi.org/10.1145/2071389.2071390

[9] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching Connected API Subgraph via Text Phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/2393596.2393606

[10] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. 2017. Word translation without parallel data. (2017).

[11] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The Vocabulary Problem in Human-system Communication. *Commun. ACM* 30, 11 (Nov. 1987), 964–971. https://doi.org/10.1145/32206.32212

[12] Edouard Grave, Armand Joulin, and Quentin Berthet. 2018. Unsupervised alignment of embeddings with Wasserstein Procrustes. (2018).

[13] Karol Gregor, Ivo Danihelka, Alex Graves, and Daan Wierstra. 2015. DRAW: A Recurrent Neural Network For Image Generation. *CoRR* abs/1502.04623 (2015). arXiv:1502.04623

[14] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[15] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 842–851.

[16] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *Advances in neural information processing systems*. 1693–1701.

[17] Emily Hill, Manuel R.Vega, Jerry A.Fails, and Greg Mallet. 2014. 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)*. 34–43. https://doi.org/10.1109/CSMR-WCRE.2014.6747190

[18] Raphael Hoffmann, James Fogarty, and Daniel S Weld. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 13–22.

[19] Reid Holmes, Rylan Cottrell, Robert J Walker, and Jorg Denzinger. 2009. The end-to-end use of source code examples: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 555–558.

[20] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code.. In *IJCAI*. 1606–1612.

[21] Hamel Husain and Ho-Hsiang Wu. 2018. How to create natural language semantic search for arbitrary objects with deep learning. Retrieved November 5, 2019 from https://towardsdatascience.com/semantic-code-search-3cd6d244a39c

[22] Hamel Husain and Ho-Hsiang Wu. 2018. Towards natural language semantic code search. Retrieved November 5, 2019 from https://githubengineering.com/towards-natural-languagesemantic-code-search/

[23] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[24] Katja Kevic and Thomas Fritz. 2014. Automatic search term identification for change tasks. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 468–471.

[25] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[27] Lei Li, Ruihai Dong, and Li Chen. 2019. Context-Aware Co-attention Neural Network for Service Recommendations. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 201–208.

[28] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (2009), 300–336.

[29] Hong Liu, Rongrong Ji, Yongjian Wu, and Wei Liu. 2016. Towards optimal binary code learning via ordinal embedding. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[30] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. 2016. Hierarchical Question-Image Co-Attention for Visual Question Answering. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 289–297. http://papers.nips.cc/paper/6202-hierarchical-question-image-co-attention-for-visual-question-answering.pdf

[31] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings* (04 2015), 545–549. https://doi.org/10.1109/SANER.2015.7081874

[32] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. 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, 260–270.

[33] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding relevant functions and their usage. *Proceedings - International Conference on Software Engineering*, 111–120. https://doi.org/10.1145/1985793.1985809

[34] Volodymyr Mnih, Nicolas Heess, Alex Graves, and koray kavukcuoglu. 2014. Recurrent Models of Visual Attention. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2204–2212. http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.pdf

[35] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian sketch learning for program synthesis. *CoRR, abs/1703.05698* (2017).

[36] Hyeonseob Nam, Jung-Woo Ha, and Jeonghee Kim. 2017. Dual Attention Networks for Multimodal Reasoning and Matching. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[37] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*. ACM, 491–502.

[38] Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Learning to rank code examples for code search engines. *Empirical Software Engineering* 22, 1 (2017), 259–291.

[39] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 357–367.

[40] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 465–475.

[41] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. 2019. Automatic query reformulation for code search using crowdsourced knowledge. *Empirical Software Engineering* 24, 4 (01 Aug 2019), 1869–1924. https://doi.org/10.1007/s10664-018-9671-0

[42] Steven P. Reiss. 2009. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 243–253. https://doi.org/10.1109/ICSE.2009.5070525

[43] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[44] Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, and Phil Blunsom. 2015. Reasoning about entailment with neural attention. *arXiv preprint arXiv:1509.06664* (2015).

[45] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 31–41.

[46] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855

[47] Kevin J. Shih, Saurabh Singh, and Derek Hoiem. 2016. Where to Look: Focus Regions for Visual Question Answering. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[48] Marijn F Stollenga, Jonathan Masci, Faustino Gomez, and Jürgen Schmidhuber. 2014. Deep Networks with Internal Selective Attention through Feedback Connections. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3545–3553. http://papers.nips.cc/paper/5276-deep-networks-with-internal-selective-attention-through-feedback-connections.pdf

[49] Jeffrey Stylos and Brad A. Myers. 2006. Mica: A Web-Search Tool for Finding API Components and Examples. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 4-8 September 2006, Brighton, UK*.

[50] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[51] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering(ASE)*. 397–407.

[52] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. 2013. A theoretical analysis of NDCG type ranking measures. In *Conference on Learning Theory*. 25–54.

[53] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Advances in Neural Information Processing Systems*. 6559–6569.

[54] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *the 31st IEEE/ACM International Conference*.

[55] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[56] Gu Xiaodong, Zhang Hongyu, and Kim Sunghun. 2018. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 933–944. https://doi.org/10.1145/3180155.3180167

[57] Caiming Xiong, Stephen Merity, and Richard Socher. 2016. Dynamic memory networks for visual and textual question answering. (2016), 2397–2406.

[58] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alex Smola. 2016. Stacked Attention Networks for Image Question Answering. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[59] Xuchen Yao, Benjamin Van Durme, and Peter Clark. 2013. Automatic Coupling of Answer Extraction and Information Retrieval. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Sofia, Bulgaria, 159–165. https://www.aclweb.org/anthology/P13-2029

[60] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *The World Wide Web Conference*. ACM, 2203–2214.

[61] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. 2017. Comparative study of CNN and RNN for natural language processing. *arXiv preprint arXiv:1702.01923* (2017).

[62] Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. 2016. Abcnn: Attention-based convolutional neural network for modeling sentence pairs. *Transactions of the Association for Computational Linguistics* 4 (2016), 259–272.

[63] Zhou Yu, Jun Yu, Chenchao Xiang, Jianping Fan, and Dacheng Tao. 2018. Beyond Bilinear: Generalized Multimodal Factorized High-Order Pooling for Visual Question Answering. *IEEE Transactions on Neural Networks and Learning Systems* 29 (04 2018), 5947–5959. https://doi.org/10.1109/TNNLS.2018.2817340

[64] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Expanding queries for code search using semantically related api class-names. *IEEE Transactions on Software Engineering* 44, 11 (2017), 1070–1082.