CSDA: A novel attention-based LSTM approach for code search

Leiming Ren¹, Shinmin Shan¹*, Kai Wang¹, Kun Xue¹

¹ School of Software, Dalian University of Technology, Liaoning Province, Dalian 116620, China
* Corresponding author’s e-mail: ssm@dlut.edu.cn

Abstract. Previous studies have proposed semantic-based approaches for code search over large-scale codebases, which has bridged the gap in understanding the semantics between natural language and source-code language. However, these studies either failed to determine an effective method for semantic representations or lacked the distinction of semantic features. In this study, we propose a novel attention-based LSTM neural network known as CSDA (Code Search based on Description Attention), which can effectively improve the code search performance. The proposed model can focus on different parts of a semantic feature when numerous aspects of a source code snippet are used as input. As opposed to assigning the same weight to different parts of the semantic vector, CSDA takes the semantics of natural language descriptions into account, so that the subtle differences hidden in the code snippet can be discriminated and associated with the corresponding queries. We compare CSDA with the existing state-of-the-art approach CODEnn, which uses a jointly embedding technique for code search. Our experimental evaluation demonstrates that CSDA outperforms previous methods and achieves superior code search performance to CODEnn, with higher success rates and mean reciprocal ranks. This study provides significant insights into the use of semantic representation methods in deep learning-based code search approaches.

1. Introduction
Over numerous years of software development, a substantial amount of repositories have been created and promoted. These repositories contain hundreds or even thousands of source code snippets, which are used to aid programmers in implementing certain functionalities or modules of software projects. An effective code search tool over a large-scale codebase can help developers to reuse previously written source code snippets, thereby improving the efficiency and quality of software development.

Various code search approaches have been proposed. For example, Linstead et al. proposed a Lucene (traditional text search engine)-based tool known as Sourcerer [1], which uses text properties and code characteristics for large-scale code searches. Hill et al. proposed a PC-based approach that can combine the context information of source code and query descriptions to calculate the relevance of program elements [2]. Lv et al. introduced API understanding into an integrated retrieval model that combines the text similarity and API information [3]. Certain work has also focused on query expansion and regeneration. For example, Lu et al. applied a method of expanding a query with synonyms obtained from WordNet to improve the code search performance [4]. Nie et al. proposed QECK, a query expansion method based on the crowd knowledge approach, which can generate an expansion query by adding the software-specific words retrieved from pseudo relevance feedback documents to the original query [5]. The majority of these approaches have been based on information...
retrieval (IR) techniques. These methods treat the source code and natural language query as textual documents of the same nature, so that the relevant code snippet that matches a given query can be retrieved according to the text similarity.

However, a fundamental problem exists when adopting IR-based approaches in code search tasks: it is difficult to match the semantic features between the query descriptions and code snippets owing to their heterogeneity. In other words, if the source code does not include relevant textual descriptions, such as comments and annotations, IR-based code search engines tend to return unsatisfactory or irrelevant results owing to the lack of query terms or their synonyms in the source code. For example, a relevant snippet for the query *serialize the image object* is presented in Figure 1.

![Figure 1. Relevant code snippet of query serialize the image object.](image)

It is almost impossible for existing approaches to return this code snippet, as it contains neither keywords such as *serialize* or its synonyms, nor relevant API information. Furthermore, even if the source code contains certain keywords, code search engines often return inaccurate results. A typical example is the query *convert json to string*, where certain returned results actually relate to the query *convert string to json*. This is because search engines lack a deep understanding of the semantics of different query word orders.

Therefore, a significant demand exists for a tool that developers can use to bridge the gap in understanding of the semantics between natural language and source code language. Previous studies have demonstrated that semantic-based approaches can effectively improve the code search performance. For example, Majumdar et al. proposed Comment-Mine, a semantic-based architecture of a knowledge graph that combined the knowledge of the program design, implementation, and evolution [6]. Yang et al. presented a context-based approach that could automatically identify semantically relevant words in code snippets by using the context in the code and comments [7]. They considered that expanding a query with synonyms is limited in the programming domain, because programming and natural languages have unequal semantic representations. A typical example is the frequent use of abbreviations, which are difficult to match with semantically relevant words in WordNet or other English dictionaries.

The study of Reiss suggested characterizing the search requirements by using predefined specifications and used transformations to return the search results that met the programmer needs [8]. Instead of matching the query with code comments, they took into account the code identifiers and achieved satisfactory results. However, the approach of Reiss required users to provide both the syntax and semantics of the target query, which was inflexible and unfriendly for users. Gu et al. proposed a deep neural network known as CODEnn, a jointly embedding-based model that unifies the vector representation of a code snippet and its corresponding description [9]. Using the unified vector representation, the semantically relevant code snippet can be retrieved according to its vector similarity to that of the query description.
Figure 2. Examples illustrating subtle differences in code snippets: (a) code snippet of query \textit{save image as png} and (b) code snippet of query \textit{save image as jpg}.

However, the two core modules of CODEnn, namely code embedding and description embedding, work individually and only consider the embedded vectors in the respective networks, ignoring the contributions of different parts of the source code features (such as the method name, API invocation sequence, and tokens in the method body) for semantic understanding. For example, CODEnn may return the same code snippet for the queries \textit{save image as png} and \textit{save image as jpg}. This is because the two expected code snippets have the same method name \textit{saveImage} and API invocation sequence \texttt{javax.imageio.ImageIO}. The main difference in the tokens in the method body (see Figures 2(a) and (b)) is not captured by the model, and this is precisely the only feature that can match the difference between the two queries. Furthermore, when several relevant results exist in the returned code snippets, partially relevant results are sometimes ranked higher than the correct results owing to the lack of code feature distinction. Therefore, an effective code search tool must have the ability to distinguish the subtle differences among features, and to match the semantic relationship between the code snippets and natural language queries.

In this study, we propose a novel approach known as Code Search based on Description Attention Neural Network (CSDA) for the code search problem. Inspired by existing approaches, CSDA uses the jointly embedding technique to implement the unified semantic representation, making it possible to provide improved measurements of the semantic similarity between the code snippet and natural language query. Previous works generally lacked the ability to discriminate semantic features. To overcome this drawback, CSDA is designed based on the attention mechanism. By assigning attention weights to different word locations of semantic features according to their importance, the subtle differences in the semantic features, such as the API invocation sequence and token orders, can be captured by CSDA.

The purpose of code search is to retrieve the relevant code snippets that match a given query. To evaluate the effectiveness of CSDA, we compare CSDA with the existing state-of-the-art approach CODEnn using the same datasets, which are collected from 18,233,872 Java methods. Our experimental results demonstrate that CSDA outperforms previous approaches and achieves superior code search performance in terms of the SR@k (success rate), MRR@k (mean reciprocal rank), and FR@k (best hit rank).
The main contributions of this work are as follows:

- We propose a novel attention-based LSTM neural network known as CSDA to use the embedding representations of descriptions to determine the attention weights along with the vector representations of code snippets.
- We compare CSDA with the existing state-of-the-art approach CODEnn over a large-scale codebase, thereby demonstrating the effectiveness of the proposed method.

The remainder of this paper is organized as follows. Section 2 describes the background of LSTM based on the attention mechanism. Section 3 presents the detailed design of the proposed attention-based LSTM neural network for code search. Section 4 explains the experimental preparation, followed by Section 5, which presents the evaluation results. We conclude the paper in Section 6.

2. Background

In this section, we introduce the background of LSTM based on the attention mechanism in natural language processing (NLP), which is discussed and adopted in our work.

**Attention-based LSTM**

Humans can make use of limited attention resources to screen high-quality information from a large amount of information rapidly. The attention mechanism has been proposed to solve various existing problems in deep learning by imitating this human characteristic [10].

LSTM is explicitly designed to avoid the problem of long-term dependencies [11, 12]. However, it is difficult to obtain a final reasonable vector representation when the model input sequence is lengthy. That is, it cannot capture the key parts in the final representation [13]. To amplify the contribution of the important parts of the input information, the attention mechanism is designed to focus on the important parts and ignore the other, useless ones. Figure 3 presents the basic structure of LSTM with attention.

![Figure 3. Basic structure of LSTM with attention.](image)

Let $H \in R^{d \times T}$ be the hidden state vector $[h_1, h_2, ..., h_T]$ produced by the last layer of LSTM, where $d$ is the size of the hidden layers of LSTM and $T$ is the length of the given information, such as a sentence, which is denoted by $[x_1, x_2, ..., x_T]$. The attention mechanism assigns a weight $a_t$ to each input term, so that the important features of the input information are captured. In the following section, we describe how the attention mechanism operates in our model and explain how the proposed model can significantly improve the code search performance in principle.

3. Code search based on description attention

To address the problems of the unreasonable semantic representation and poor distinction of semantic features in previous works, we propose a novel attention-based LSTM network known as CSDA to improve the code search performance.

3.1. Architecture

We believe that a relevant code snippet that can be retrieved by a query should rely more on the semantic similarity between the code snippet and query description than their text similarity. Therefore,
two embedding networks are necessary to embed the code snippet and query description into semantic vectors. Gu et al. conducted a substantial amount of valuable work on the analysis and extraction of code semantics [9]. Based on their study, the code snippet features are embedded into several semantic vectors individually and then combined as a unified semantic representation of the source code. Moreover, an attention mechanism-based module is used to assign different attention weights to the input information, so that the important parts of the semantic features can be captured by the model. Finally, we use a similarity module to measure the semantic similarity between the code snippet and description. The overall architecture of CSDA is presented in Figure 4. The following subsections describe the specific implementation details of CSDA.

3.1.1. Code embedding network. This module is used to embed the code snippet information into a code semantic vector. Unlike natural language descriptions, the code snippet features can be mined from various aspects of the snippet. Therefore, simply using single aspect embedding of the code snippet is not sufficient to represent its semantics. Three code snippet aspects are used in this study: the method name, API invocation sequence, and tokens in the method body.

To make optimal use of the code information, we propose learning a semantic representation for each aspect individually. Let $C = [M, A, T]$ be the input code snippet, where $M$, $A$, and $T$ denote the method name, API invocation sequence, and tokens in the method body, respectively. We present the specific designs of the three embedding networks in the following.

3.1.2. Method name embedding. The input of method name embedding is a sequence of camel split words, which is extracted from the method name of the code snippet. Let $M = [m_1, m_2, ..., m_{N_m}]$ denote the sequence of the camel split words. The embedding network embeds the input information of the method name using bidirectional LSTM (BiLSTM):

$$f_t = \text{sigmoid}(W_f m_t + U_f h_{t-1} + b_f)$$  \hspace{1cm} (1)

$$i_t = \text{sigmoid}(W_i m_t + U_i h_{t-1} + b_i)$$  \hspace{1cm} (2)

$$c_t = \text{tanh}(W_c m_t + U_c h_{t-1} + b_c)$$  \hspace{1cm} (3)

$$c_t = f_t c_{t-1} + i_t c_t$$  \hspace{1cm} (4)
\[ o_t = \text{sigmoid}(W_o m_t + U_o h_{t-1} + b_o) \]  
\[ h_t = o_t \times \tanh(c_t), \]

where \( m_t \in \mathbb{R}^d \) is the embedding vector of the \( t \)-th word \( m_t \) in \( M \); and \( f_t, i_t, \hat{c}_t, c_t, o_t, \) and \( h_t \) represent the forget threshold, memory threshold, temporary cell state, current cell state, output threshold, and hidden layer state in the LSTM network, respectively. Furthermore, \( W_m, U_m, b_m \), and \( m \in \{ f, i, c, o \} \) all constitute the matrix of trainable parameters.

To capture the bidirectional semantic dependencies, we use BiLSTM in the network. In the bidirectional architecture, two separate LSTMs capture the semantic features in two directions, and produce the forward hidden state \( \overrightarrow{h_m} \) and backward hidden state \( \overleftarrow{h_m} \). We represent the final semantic vector \( v_m \) of the method name by means of the concatenation of \( \overrightarrow{h_m} \) and \( \overleftarrow{h_m} \):

\[ v_m = \overrightarrow{h_m} \mid \overleftarrow{h_m}, \]

where \( \overrightarrow{h_m} \) and \( \overleftarrow{h_m} \) are the forward and backward hidden states, respectively, with \( t \in \{1,2,\ldots,N_m\} \), and \( v_m \) is the final semantic vector of the method name.

**API embedding**: The input of the API embedding is a sequence of consecutive API method invocations, which is extracted from the method body of the code snippet. Let \( A = [a_1, a_2, \ldots, a_{N_a}] \) denote the sequence of the API method invocations. Likewise, the embedding network embeds the input information of the API sequence using BiLSTM. Finally, the API sequence can be embedded into a semantic vector \( v_a \):

\[ v_a = \overrightarrow{h_a} \mid \overleftarrow{h_a}, \]

where \( \overrightarrow{h_a} \) and \( \overleftarrow{h_a} \) are the forward and backward hidden states produced by the final layer of LSTM in the API embedding, with \( t \in \{1,2,\ldots,N_a\} \), and \( v_a \) is the final semantic vector of the API sequence.

**Token embedding**: The input of the token embedding is a sequence of tokens, which is extracted from the method body of the code snippet. Let \( T = [t_1, t_2, \ldots, t_{N_t}] \) denote the sequence of tokens. Unlike the above embedding models, token embedding uses the embedding representation of the descriptions to determine the attention weights along with the vector representation of the tokens, so that the subtle differences in the tokens can be captured. The design of the token embedding is illustrated in Figure 5.

---

**Figure 5.** Detailed design of token embedding with attention.
The attention mechanism generates an attention weight vector $\alpha = [\alpha_1, \alpha_2, ..., \alpha_{N_t}]$. Thereafter, we use the weight vector $\alpha$ to compute the weighted sum of the token semantic vectors at each input token as the final semantic representation of the overall token information:

$$
\alpha_t = \text{softmax} \left( \left[ \overrightarrow{h_{kt}}, \overleftarrow{h_{kt}} \right] W_k v_d \right)
$$

$$
v_t = \sum_{t=1}^{N_t} \alpha_t \left[ \overrightarrow{h_{kt}}, \overleftarrow{h_{kt}} \right]
$$

where $\overrightarrow{h_{kt}}$ and $\overleftarrow{h_{kt}}$ are the forward and backward hidden states produced by the final layer of LSTM, $v_d$ is the final semantic vector of the tokens. Furthermore, $W_k$ is the matrix of trainable parameters.

Finally, the semantic vectors $v_m, v_a$, and $v_t$ are fused into $v_c$:

$$
v_c = \tanh \left( W_{fm} v_m + W_{fa} v_a + W_{ft} v_t \right),
$$

where $W_{fm}, W_{fa}$, and $W_{ft}$ constitute the matrix of trainable parameters. The output vector $v_c$ is the final semantic vector of the source code.

### 3.1.3 Description embedding network

This module is used to embed the information of the query descriptions into a semantic vector. The input is a sequence of natural language words, which is extracted from the documentation comment of the Java method. Let $D = [d_1, d_2, ..., d_{N_d}]$ denote the sequence of words. The embedding network embeds this sequence into a semantic vector $v_d$ using BiLSTM:

$$
v_d = \overrightarrow{h_{dt}}, \overleftarrow{h_{dt}}
$$

where $\overrightarrow{h_{dt}}$ and $\overleftarrow{h_{dt}}$ are the forward and backward hidden states produced by the final layer of LSTM in description embedding, with $t \in \{1, 2, ..., N_d\}$, and $v_d$ is the final semantic vector of the description sequence.

### 3.1.4 Similarity module

This module is used to measure the semantic similarity between the code semantic vector $v_c$ and description semantic vector $v_d$. Numerous calculation methods for the vector similarity have been used in previous studies [9, 14, 15, 16, 17]; for example, the cosine similarity, sigmoid similarity, and Euclidean distance. The cosine similarity is selected in our model.

### 3.2 Model training

The training corpus is composed of code snippets, positive query descriptions, and negative query descriptions. The triple instance $<C, D+, D->$ is trained in CSDA as the input information. An effective code search tool retrieves the relevant code snippets that match a given query. Therefore, the goal of training CSDA is to minimize the ranking loss, which is defined as:

$$
\text{loss}(\theta) = \sum_{<C, D+, D-> \in \text{Corpus}} \max(0, \epsilon - \cos(v_c, v_{d+}) + \cos(v_c, v_{d-}))
$$

where $\theta$ represents the model parameters; $\text{Corpus}$ represents the training corpus; $v_c$, $v_{d+}$, and $v_{d-}$ are the semantic vectors of $C$, $D+$, and $D-$, respectively; and $\epsilon$ is a constant margin, which is set to 0.6 in all of the experiments.

### 4 Experiments

#### 4.1 Dataset
As illustrated in Figure 4, the two core modules of CSDA are the code embedding network and description embedding network. To make the CSDA model comparable and persuasive, we adopt the exact same datasets as those used in a previous study [9].

The datasets are constructed using 18,233,872 commented Java methods, which are obtained from open-source projects on GitHub. In this case, the documentation comments of the Java methods are necessary because the CSDA description embedding network requires the query description as input. The first sentence of the comment is regarded as the query description according to the Javadoc guidance¹ and the Eclipse JDT compiler² is applied to extract the target description. Moreover, we use the method declaration as the code snippet, which includes the method name and method body.

To meet the needs of the model and make optimal use of the information in the code snippet, the processed corpus is presented in the form of <method name, API sequence, tokens, description>. Owing to space constraints, we do not present the specific data processing details. Readers can find a detailed description thereof in the study of Gu [9]. Figure 6 presents the extracted result of a sample instance, which describes a Java method `getPercentInstance` using `java.text.NumberFormat` and `java.util.Locale`.

4.2. Task definition
As described in Section 3, the key module in the CSDA model, namely the code embedding network, is composed of `method name embedding`, `API embedding`, and `token embedding`. To distinguish the importance of the different words of a token sequence, we introduce the attention mechanism in `token embedding`. Therefore, we name the training task TDA, which is used to evaluate the code search performance when a set of queries is provided.

In theory, the attention mechanism can also be adopted in `method name embedding` and `API embedding`. To provide an improved evaluation of how the attention mechanism affects the CSDA performance, we conduct comparative experiments: MDA (using the attention mechanism in `method name embedding`) and ADA (using the attention mechanism in `API embedding`).

---

¹ http://www.oracle.com/technetwork/articles/java/index-137868.html
² http://www.eclipse.org/jdt/
4.3. Baseline
We compare CSDA with the existing state-of-the-art approach CODEnn, which uses the jointly embedding technique for code search.

Before deep learning was used extensively, the code search approaches were mainly based on IR techniques. CODEnn is the first deep learning-based model, which has achieved great success in improving the code search performance. By using the unified vector representation, the semantically relevant code snippet can be retrieved according to its vector similarity to the query description. As in CODEnn, CSDA is designed to improve the semantic understanding between code snippets and natural language descriptions. Therefore, CODEnn is the ideal baseline for our experiments.

5. Evaluation

5.1. Experimental setup

5.1.1. Hyper-parameters. To provide a fair comparison, we adopt exactly the same experimental settings as those in CODEnn in our proposed model, and the detailed parameters are as follows: The dimension of the word embedding is set to 100. All BiLSTMs in our experiments have 200 hidden units in each direction. The maximum lengths of the method name, API, and token sequences are 6, 30, and 50, respectively, whereas the maximum length of the query description is set to 30.

CSDA is trained via the mini-batch Adam algorithm. The batch size is set to 128, which is consistent with CODEnn. Moreover, we use the same vocabulary that is most frequently used in the training corpus. We also use dropout to turn off neurons in the network randomly, thereby preventing the co-adaptation of neurons. The dropout is set to 0.25 at both the embedding and LSTM layers.

All of the models are trained on a server with one Tesla V100 GPU and the total cost of the training time is nearly one month.

5.1.2. Performance measures. We use three common metrics to measure the effectiveness of the code search, namely $FR@k$, $SR@k$, and $MRR@k$. These metrics are used extensively in the IR and code search literature [3, 4, 5, 9].

The metric $FR@k$ (also known as the best hit rank at $k$) is the rank of the first correct code snippet in the predicted snippet list. For example, if the real snippet is 36 and the top 5 predicted snippet list is [51, 27, 9, 36, 31], the value of $FR@5$ is 4. In this study, it is calculated as follows:

$$FR@k = \frac{1}{|Q|} \sum_{q=1}^{Q} \rho(index_q < k) + 1,$$

where $Q$ is the set of queries and $index_q$ represents the index of the real snippet in the predicted snippet list; if the real snippet does not exist in the predicted snippet list, the value of $index_q$ is $k$. Moreover, $\rho(\cdot)$ is a function that returns $index_q$ if the input is true and $k$ otherwise. $FR@k$ is important for evaluating the code search performance as it determines whether developers can discover the target code snippet from the returned result in a short time. A lower metric value means a lower rank of the correct result and better code search performance.

The $SR@k$ metric (also known as the success/accurate percentage at $k$) evaluates the percentage of queries for which more than the correct code snippet could exist in the top $k$ returned snippet list. For example, if the value of $SR@10$ is 87%, the correct code snippet exists in the top 10 returned results for 87% of the queries. $SR@k$ is calculated as follows:

$$SR@k = \frac{1}{|Q|} \sum_{q=1}^{Q} \delta(index_q < k),$$

where $Q$ is the set of queries, and $\delta(\cdot)$ is a function that returns 1 if the input is true and 0 otherwise. $SR@k$ is important because developers aim to determine the target code snippet from few candidate
sets. An effective code search tool should reduce the possibility and difficulty of secondary retrieval. A higher metric value indicates better code search performance.

The $MRR$ metric represents the mean reciprocal rank of a couple of predicted results. In fact, $MRR@k$ is determined by $FR@k$, and the difference is that a higher $MRR@k$ value indicates superior code search performance.

5.2. Results

Based on the same datasets and experimental settings, we evaluate the performance of CSDA and the state-of-the-art approach CODEnn.

Table 1. Overall evaluation of CSDA and CODEnn.

| Models    | SR@1 | SR@5 | SR@10 | MRR@5 | MRR@10 | FR@5 | FR@10 |
|-----------|------|------|-------|-------|--------|------|-------|
| CODEnn    | 0.48 | 0.91 | 0.95  | 0.58  | 0.43   | 2.32 | 3.81  |
| MDA       | 0.66 | 0.85 | 0.91  | 0.55  | 0.40   | 2.57 | 4.17  |
| ADA       | 0.86 | 0.95 | 0.97  | 0.63  | 0.45   | 2.10 | 3.52  |
| TDA       | 0.91 | 0.97 | 0.99  | 0.67  | 0.46   | 1.92 | 3.41  |

Table 1 presents the evaluation results of the four models, measured in terms of $SR@k$, $MRR@k$, and $FR@k$. The columns $SR@1$, $SR@5$, and $SR@10$ indicate the values of $SR@k$ when $k$ is set to 1, 5, and 10, respectively. Similarly, the columns $MRR@k$ and $FR@k$ display the values of $MRR$ and $FR$ when $k$ is 5 and 10, respectively (when $k$ is set to 1, $SR@k$ and $MRR@k$ have the same value according to their definition, and $FR@k$ is always greater than $k$). The result indicates that TDA exhibits improvements in all metrics compared to CODEnn. It can be clearly observed that TDA has a higher $SR@k$ value when $k$ is 1 and the improvement over CODEnn is 90%. The $SR@1$ value is 0.91, which means that for 91% of the queries, the first result returned by TDA is the correct code snippet, whereas the value is 0.48 for CODEnn. In particular, for $SR@k$, the improvements over CODEnn are 7% and 4% when $k$ is set to 5 and 10, respectively. For $MRR@k$, the improvements over CODEnn are 16% and 7% when $k$ is set to 5 and 10, respectively. For $FR@k$, the improvements over CODEnn are 17% and 10% when $k$ is set to 5 and 10, respectively.

To demonstrate the effects of the attention mechanism, we evaluate the performance of two other attention mechanism-based models, namely MDA and ADA. MDA uses the attention mechanism in method name embedding and ADA uses the attention mechanism in API embedding. The results indicate that ADA also outperforms the baseline on all evaluation metrics, but the performance is not as good as TDA. This is because the tokens in the method body contain more recognizable words and prominent features than the method name and APIs. These words and features are easier to be captured by attention mechanism, so that the model can understand the key parts of the input information. The evaluation results of ADA indicate that not all models with attention mechanism operate better than those without attention mechanism. The reason is that the attention mechanism performs better when the input sequence is long, whereas the length of the input method name is set to a small value of 6 in all of the experiments.

Overall, our proposed model TDA is superior to CODEnn on all evaluation metrics and exhibits state-of-the-art performance.

6. Conclusions

We have proposed a novel approach known as CSDA for the code search problem. CSDA uses the jointly embedding technique to implement unified semantic representations, making it possible to provide improved measurements of the semantic similarity between the code snippet and natural language query. To address the problem of the lack of semantic distinction exhibited in previous work, CSDA is designed based on the attention mechanism. As opposed to assigning the same weight to different parts of the semantic vector, CSDA takes the semantics of natural language descriptions into account, so that the subtle differences hidden in the code snippet can be discriminated and associated
with the corresponding queries. The experiments demonstrated that our proposed model can achieve excellent code search performance and outperforms previous approaches.

The significant success of the embedding technique in NLP provides us with important insight into whether deep learning-based code search approaches can be improved effectively if more appropriate semantic representation methods are used. In the future, we will identify more effective semantic representation methods so that the rich semantics contained in the source code can be fully expressed.

Acknowledgments
This work is supported in part by the Natural Science Foundation of 453 China grant 61502069, 61672128, and by the Fundamental Research Funds 454 for the Central Universities grant DUT18JC39, DUT18GF108.

References
[1] Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P. (2009). Sourcerer: mining and searching internet-scale software repositories. Data Min. Knowl. Disc., 18(2): 300–336.
[2] Hill, E., Pollock, L., Vijay-Shanker, K. (2011). Improving source code search with natural language phrasal representations of method signatures. In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 524–527, IEEE.
[3] Lv, F., Zhang, H., Lou, J. G., Wang, S., Zhang, D., Zhao, J. (2015). Codehow: Effective code search based on api understanding and extended boolean model. In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), pp. 260–270, IEEE.
[4] Lu, M., Sun, X., Wang, S., Lo, D., Duan, Y. (2015). Query expansion via wordnet for effective code search. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pp. 545–549, IEEE.
[5] Nie, L., Jiang, H., Ren, Z., Sun, Z., Li, X. (2016). Query expansion based on crowd knowledge for code search. IEEE Transactions on Services Computing, 9(5): 771–783.
[6] Majumdar, S., Papdeja, S., Das, P. P., Ghosh, S. K. (2020). Comment-Mine—A Semantic Search Approach to Program Comprehension from Code Comments. In Advanced Computing and Systems for Security, pp. 29–42, Springer, Singapore.
[7] Yang, J., Tan, L. (2012). Inferring semantically related words from software context. In 2012 9th IEEE Working Conference on Mining Software Repositories (MSR 2012), pp. 161–170, IEEE.
[8] Reiss, S. P. (2009). Semantics-based code search. In 2009 IEEE 31st International Conference on Software Engineering, pp. 243–253, IEEE.
[9] Gu, X., Zhang, H., Kim, S. (2018). Deep code search. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE 2018), pp. 933–944, IEEE.
[10] Wang, Y., Huang, M., Zhu, X., Zhao, L. (2016). Attention-based LSTM for aspect-level sentiment classification. In Proceedings of the 2016 conference on empirical methods in natural language processing, pp. 606–615.
[11] Zheng, J., Zheng, L., Yang, L. (2019). Research and Analysis in Fine-grained Sentiment of Film Reviews Based on Deep Learning. J. Phys. Conf. Ser., 1237(2): 022152, IOP Publishing.
[12] Qu, Y., Zhao, X. (2019). Application of LSTM Neural Network in Forecasting Foreign Exchange Price. J. Phys. Conf. Ser., 1237(4): 042036, IOP Publishing.
[13] Zhou, X., Wan, X., Xiao, J. (2016). Attention-based LSTM network for cross-lingual sentiment classification. In Proceedings of the 2016 conference on empirical methods in natural language processing, pp. 247–256.
[14] Zhang, Y., Liu, Y. D., Ji, Z. (2009). Vector similarity measurement method. Technical Acoustics, 28(4): 532–536.
[15] Si, S., Zheng, W., Zhou, L., Zhang, M. (2019). Sentence Similarity Computation in Question Answering Robot. J. Phys. Conf. Ser., 1237(2): 022093, IOP Publishing.

[16] Corley, C. S., Damevski, K., Kraft, N. A. (2015). Exploring the use of deep learning for feature location. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME 2015), pp. 556–560, IEEE.

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