Learning Embeddings of API Tokens to Facilitate Deep Learning Based Program Processing

Yangyang Lu, Ge Li(✉), Rui Miao, and Zhi Jin(✉)

Key Lab of High-Confidence Software Technology,
Ministry of Education, Peking University, Beijing, China
{luyy,lige,miaorui,zhijin}@pku.edu.cn

Abstract. Deep learning has been applied for processing programs in recent years and gains extensive attention on the academic and industrial communities. In analogous to process natural language data based on word embeddings, embeddings of tokens (e.g. classes, variables, methods etc.) provide an important basis for processing programs with deep learning. Nowadays, lots of real-world programs rely on API libraries for implementation. They contain numbers of API tokens (e.g. API related classes, interfaces, methods etc.), which indicate notable semantics of programs. However, learning embeddings of API tokens is not exploited yet. In this paper, we propose a neural model to learn embeddings of API tokens. Our model combines a recurrent neural network with a convolutional neural network. And we use API documents as training corpus. Our model is trained on documents of five popular API libraries and evaluated on a description selecting task. To our best knowledge, this paper is the first to learn embeddings of API tokens and takes a meaningful step to facilitate deep learning based program processing.

Keywords: API tokens · Embeddings · Program processing · Deep neural networks

1 Introduction

Deep learning has achieved significant breakthroughs in a number of fields, such as image processing [9,11], speech recognition [5] and natural language processing [2,19]. The mainstream models of deep learning, *deep neural networks (DNNs)*, can extract complex features from raw data with little human engineering knowledge. Recently, the advantages of deep learning have also been exploited in program processing and gains more and more attention on the academic and industrial communities [1,15,22].

When processing programs with DNNs, it's usually required to represent tokens (e.g. classes, variables, methods, etc.) in programs as real-value embeddings so that DNNs could accept programs as inputs. This is analogous to processing natural language sentences and paragraphs based on word embeddings [4,10]. A few works have been proposed to learning embeddings of tokens in

F. Lehner and N. Fteimi (Eds.): KSEM 2016, LNAI 9983, pp. 527–539, 2016.
DOI: 10.1007/978-3-319-47650-6_42

programs, such as learning embeddings of identifiers based on programs' abstract syntax trees [15] or learning embeddings of keywords in programs for software document retrieval [21].

However, learning embeddings of *API*[1] tokens has not been exploited yet, while they appear in lots of real-world programs. API tokens here refer to the program artifacts related to API libraries, like classes, interfaces, methods etc. Nowadays more and more programs rely on API libraries for implementation. Numbers of API tokens are used to leverage functions provided by API libraries. These API tokens indicate important semantics of programs and play a vital role in processing and analyzing the corresponding programs.

Figure 1 shows an example of a code snippet *SimpleHostConnectionPool.java* of the *Astyanax* project. It implements an internal method to wait for a connection on the available connection pool. As shown in Fig. 1(a), API tokens are related to core functions of this program, such as obtaining the start time, trying to get a free connection, throw timeout exception and interrupting threads of waiting for connections. We can infer that these API tokens have more important influences than other components (Fig. 1(b)) of programs. Learning embeddings of API tokens is meaningful in processing programs that rely on API libraries.

In this paper, we propose a neural model to learn embeddings of API tokens to facilitate deep learning based program processing. Our model is composed of a recurrent neural network with a convolutional neural network to learn embeddings of API tokens from API documents. API documents provide detailed descriptions of functions for API tokens. Our dataset contains documents of five popular Java API libraries to train our model. Finally, we use a description selecting task to evaluate learnt embeddings. To the best of our knowledge, our work is the first to learn embeddings of API tokens and takes a meaningful step on processing programs with deep learning.

The rest of this paper is organized as follows. Section 2 introduces related work of deep learning based program processing. Section 3 describes our model for learning embeddings of API tokens with deep neural networks. Section 4 illustrates the dataset information and evaluation results. Section 5 presents the conclusion.

2 Related Work

Deep learning has been applied to program processing of different application scenarios. In this section, we first introduce the most relevant work to ours, then discuss other related work.

Mou et al. [15–17] proposed a tree-based convolutional neural network (TBCNN) for classifying programs with different algorithm labels. TBCNN was constructed based on programs' abstract syntax trees (ASTs) to capture programs' structural features. Also, TBCNN learned embeddings for identifiers of AST trees and program vectors. The dataset was composed of *C* programs which

[1] https://en.wikipedia.org/wiki/Application_programming_interface.

```
private Connection<CL> waitForConnection(int timeout) throws ConnectionException {
        Connection<CL> connection = null;
        long startTime = System.currentTimeMillis();
        try {
            blockedThreads.incrementAndGet();
            connection = availableConnections.poll(timeout, TimeUnit.MILLISECONDS);
            if (connection != null)
                return connection;

            throw new PoolTimeoutException("Timed out waiting for connection")
                .setHost(getHost())
                .setLatency(System.currentTimeMillis() - startTime);
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new InterruptedOperationException("Thread interrupted waiting for connection")
                .setHost(getHost())
                .setLatency(System.currentTimeMillis() - startTime);
        }
        finally {
            blockedThreads.decrementAndGet();
        }
    }
```

(a) API tokens are notated in blue.

```
private Connection<CL> waitForConnection(int timeout) throws                    {
        Connection<CL> connection = null;
        long startTime =                    ;
        try {
                                            ;
            connection =                    (timeout,                    );
            if (connection != null)
                return connection;

            throw new PoolTimeoutException("Timed out waiting for connection")
                .setHost(getHost())
                .setLatency(                    - startTime);
        }
        catch (                    e) {
                                            ;
            throw new InterruptedOperationException("Thread interrupted waiting for connection")
                .setHost(getHost())
                .setLatency(                    - startTime);
        }
        finally {
                                            ;
        }
    }
```

(b) API tokens are hidden in grey.

Fig. 1. Code from SimpleHostConnectionPool.java of the Astyanax project (Color figure online)

were written by students to solve algorithm assignments. However, TBCNN is inappropriate for processing real-world programs that contain numbers of API tokens. API tokens will be transferred to identifiers of AST trees, which leads to the lost of important semantics in such programs. We propose to learn embeddings of API tokens based on its corresponding descriptions in API documents, which can capture the tokens' semantics.

Ye et al. [21] split code and text into sets of keywords, then adapted Skip-gram model to learn embeddings of keywords. The learnt embeddings were used to compute document similarities for software document retrieval tasks. They took documents as bag-of-words and used word-based equations to measure similarities between documents. API documents were used as a part of training datasets. Differently, we take API tokens and their descriptions as word sequences, capture the sequential context rather than bag-of-words and output embeddings for API tokens completely.

Despite the above work, deep learning was also used to predict programs' execution results or generate programs. Zaremba et al. [22] presented a character-level recurrent neural network to predict outputs of short python code. Allamanis et al. [1] proposed an attentional convolutional neural network to generate short, descriptive function name-like phrases for given code snippets. Ling et al. [13] presented a neural network architecture to generate programs for two card games with a mixed natural language and structured specification Gu et al. [6] modified the sequence-to-sequence network of machine translation [19] to generate API call sequences for given queries.

3 Learning Embeddings of API Tokens

3.1 Overview

We use API documents as training corpus to learn embeddings of API tokens. Because API documents provide mappings of API tokens and descriptions, which contain useful semantic information about API tokens' function.

Table 1 shows a mapping example. As for the API token part, we involve the package prefix ("java.awt.color") as a part of the entire API token. Because in real-world programs, packages should be imported before using API tokens. As for the description part, it provides a natural language summary of the corresponding API token's function, which is useful for program processing.

Table 1. An example of API tokens-description mappings

Token	Description
java.awt.color.ColorSpace	This abstract class is used to serve as a color space tag to identify the specific color space of a Color object or, via a ColorModel object, of an Image, a BufferedImage, or a GraphicsDevice

Here we explain several pre-processing steps for the actual inputs of our neural model first. We process the raw data of tokens and descriptions into word sequences. First, we remove punctuations and split the text into sequences by whitespaces. Then we decompose the words of the CamelCase[2] format

[2] https://en.wikipedia.org/wiki/CamelCase.

(e.g. names of classes, interfaces, etc.) into componential words by breaking at the positions of the capital letter or the underline (Sect. 4.1). There are two reasons for this operation. First, CamelCase rules use word combinations for token naming. The chose componential words usually help to express semantics of the entire token. Second, API tokens under the same package often share componential words as the common prefix, which indicates important lexical context of API tokens. For example, "InputMethod","InputMethodContext" and "InputMethodDescriptor" under the package "java.awt.im.spi" have the common componential words "Input" and "Method". The CamelCase decomposition helps to capture the common prefix and makes the semantic space to be respectively dense. After that, we do the lowercase and stemming operations on each word. Finally, we get word sequences of the token and description. They are notated as *tSeq* and *dSeq* respectively.

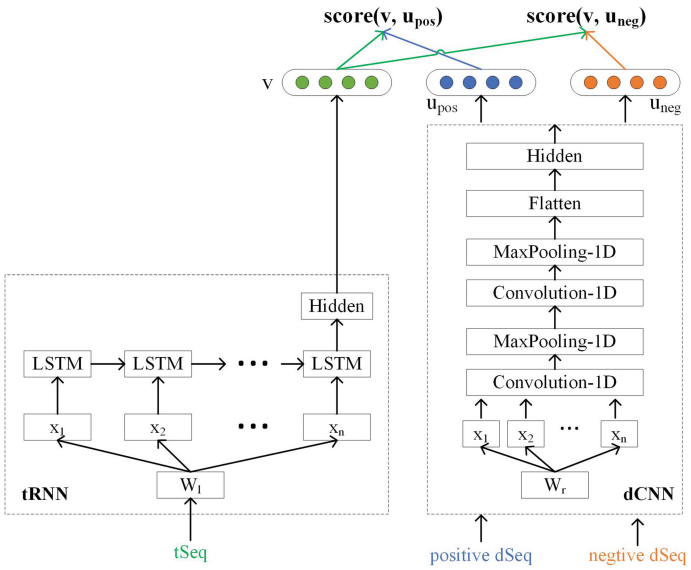

Fig. 2. Overview of the model architecture

The overview of the model architecture is shown as Fig. 2. Our neural model consists of a recurrent neural network (notated as tRNN) and a convolutional neural network (notated as dCNN). (The basic knowledge of networks is introduced in Sect. 3.2). tRNN accepts tokens as inputs and outputs embeddings of tokens. Correspondingly, dCNN accepts descriptions as inputs and outputs embeddings of descriptions. To capture the semantic mapping relation between tokens and descriptions, we use a *score* function to compute the similarity between tRNN's outputs and dCNN's outputs. Then to train the model in an unsupervised manner, we introduce noise-contrastive estimation [7] into our model, which is widely used in natural language modeling [14] and proved to

be fast and effective on model training [4,23]. Its basic idea is to train a logistic regression classifier to discriminate between samples from the data distribution (i.e. positive samples) and samples from some "noise" distribution (i.e. negative/noisy samples). In this paper, the positive sample is the correct token-description pair and the noisy pair is a wrong description paired with a given token. Then positive and negative samples share dCNN in our architecture. Finally, the objective function of our model is to minimize the max-margin loss J as Eq. 1.

$$J(\theta) = \sum_{(i,j)\in P} \sum_{k\in S_i} \max(0, \Delta - score(v_i, u_j) + score(v_i, u_c)) \tag{1}$$

In Eq. 1, v_i is the learnt embedding for the token part. u_j and u_c are the learnt embeddings for the correct and noisy descriptions respectively. P is the set of all matched API token-description pairs (i, j) and S is the set of unmatched descriptions for i_th token.

3.2 Basic Knowledge of RNNs and CNNs

Recurrent neural networks (RNNs) are widely used to process sequential data across a wide range of applications in speech recognition [5] and NLP [18]. For a given sequence, RNNs do the same operation for every element in it, which means "recurrent". Usually, the basic RNNs [18] read one element at one time step, then send both the output of the previous time step and the current element input into the recurrent layer at the next step. The basic RNNs suffer from the gradient vanish problem in training, which leads to the lost of long history information. Then gated recurrent units, like LSTM [8] and GRU [3] cells, are proposed to memorize longer dependencies. They use neural gates that could read or forget information via internal memory states and have been successfully applied to numbers of NLP tasks, such as machine translation [19] and answer selecting [20].

Convolutional neural networks (CNNs) are firstly used in image classification [12] and have been applied to NLP tasks successfully [4,10] in recent years. CNNs contain two core operations: convolution and pooling. The convolution operation uses a filter with a fixed window size to extract local features of the input data. The pooling operation deals with the variant size of the input data and tailors them into the same size with given functions like maximal or average. CNNs usually process inputs with groups of convolutional layers and pooling layers, then use a layer to flatten all the feature maps into one fixed-size embedding for the supervised classification tasks.

3.3 tRNN

Here we first explain the reason for choosing the LSTM-based recurrent neural network to learn embeddings for the token part. The token part keep a hierarchical structure, which indicates different levels of concept abstraction. It is

shown in the package prefix. Then after decomposing tokens of the CamelCase format into words, there's also a hierarchy among these componential words. Since RNNs have been proved powerful on capturing sequential context [18] and LSTM cells could memorize long sequential context better [8], we use a LSTM-based recurrent neural network (tRNN) to capture tSeq's hierarchical context and learn the embedding of the token part.

Then we introduce the process of learning embeddings for tokens via tRNN. As shown in Fig. 2(left), tRNN reads words in a direction from the general package prefix snippet to the specific word after CamelCase decomposition. Each word of the *tSeq* is transferred to an embedding via a look-up matrix W_l, then sent to a *LSTM* layer. When tRNN reads one word, it moves one time step. At each time step, the *LSTM* layer accepts current word's embedding and the output of the previous time step, then sends its output to the next step. After reading the last word, tRNN sends the output of the *LSTM* layer in the last time step into a fully connected hidden layer, which outputs a d-dimensional embedding for the token part.

3.4 dCNN

Here we also explain why we use a convolutional neural network to encode the description part first. According to our observations and statistics on API documents of five API libraries (shown in Sect. 4.1), descriptions usually contain a long sentence, even a small paragraph. Then we note that the core function is usually related to several local phrases in the long descriptions. Since CNNs are good at extracting local features across words without any syntactic parsing operations, we apply CNNs to learn the semantics of the description into embeddings.

The architecture of dCNN is shown in Fig. 2 (right). Words of the *dSeq* are transferred to embeddings via a look-up matrix W_r first. Then we process the sequence of word embeddings via two groups of 1D-convolutional layers and max-pooling layers. Here we use the 1D-convolutional layer in [10] to extract local features of phrases. It contains several convolution kernels. Each kernel slides a window of m words on the entire word sequence and outputs a filtered feature map of the input sequence. Then max-pooling layers are arranged after the 1D-convolutional layer and used to remain maximal values of each dimension based on the given window, so that the features of key phrase can be captured. After that, a flatten layer is used to concat features of different feature maps into one fixed-size embedding. Finally, dCNN sends the flatten layer's output into a fully connected hidden layer and gets the d-dimensional vector of the description part.

3.5 Tree-Based Negative Sampling

In this section, we illustrate our negative sampling method. To train a model via noise-contrastive estimation, negative samples are needed as the input. Since API documents only provide positive pairs, which are matched token-description

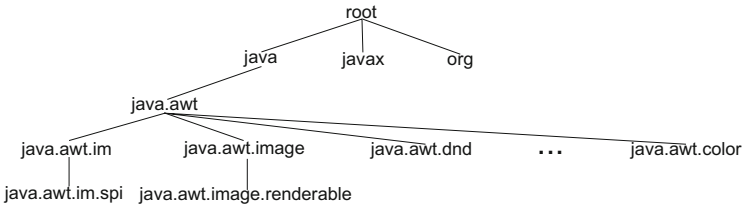

Fig. 3. A multiway tree constructed from package prefixes of Java SE 8

pairs, we need to extract negative samples by ourselves. For a given API token, we expect to find appropriate noisy descriptions. If they were too hard to distinguish on lexical information, it would take a long time for our model to converge to the optimal point. If they were too easy via identifying just from the type information, the robustness of our model would be weak.

Here we present a tree-based negative sampling method. We note that the package prefixes in tokens hold hierarchical structures that indicate different abstraction levels. Descriptions under the same package prefix share more common words than those under different package prefix. Then descriptions of API tokens with different types, such as *Class*,*Exception*, *Enum*, are usually easier to be distinguished from the sentence pattern that those with same types. So construct a multiway tree based on all the package prefixes of the corpus to sample noisy descriptions, like the one shown in Fig. 3. Each node presents a part of the package prefix. A virtual root node is added to connect all the package prefixes into a universal tree. Token-description mappings are saved under leaf nodes. For tokens or descriptions under different leaf nodes, we assume that the more common ancestor nodes they share, the more similar they are. So for a given API token T, we trace from the leaf node L of its package prefix, and go up with h depth. Then we find the find the ancestor A with h depth far from the leaf node L. Now we collect descriptions saved under the leaf nodes of the ancestor A except for the subtrees containing L. Then we filter descriptions holding the same type of T as candidates. Finally, we sample k descriptions randomly from the collected candidates as noisy descriptions for the given API token.

4 Evaluation

4.1 Datasets

In this paper, we use API reference documents of five Java API libraries as datasets: Jave SE 8, Eclipse Platform 4.3, Spring 4.3, Lucene 6.1 and Java EE 7. We download API documents from their official websites and use an HTML parser to extract mappings of API tokens and descriptions in raw text.

Several pre-processing steps are done to transfer these raw text of mapping into word sequences, so that they can be fed into the model in Sect. 3. First we remove punctuations and numerical digits, then split the text into word

sequences by whitespaces. Then we do CamelCase decomposition operations on tokens of the CamelCase format, of which the reasons have been explained in Sect. 3.1. We break tokens of the CamelCase format into words at the position of the first capital letter followed by a lowercase letter or the position of the underline symbol. For example, "ICC_ColorSpace" is transferred to a word sequence of "ICC", "Color" and "Space". Finally, we do lowercase transformation and stemming on all the words.

Table 2 shows the basic statistical information of the five datasets. We remove the mappings of one-letter template classes and mappings of empty descriptions from the datasets, of which the number is tagged by "Filtered". Then we split the data into the training, dev, test subsets with the ratio of 8:1:1. "AvgLength" and "MaxLength" show the average and maximal length of the word sequences – tSeq and dSeq. "VocabSize" gives the vocabulary size for words in the token and description part.

Table 2. API documents for training and test

API library	Java SE 8	Eclipse Platform 4.3	Spring 4.3	Lucene 6.1	Java EE 7
#Sample (filtered/original)	4252/4457	3883/4001	3821/3875	2659/2672	2038/2134
#Train/#Dev/#Test	3401/425/426	3106/388/389	3056/382/383	2127/266/266	1630/204/204
#AvgLength (TOKEN/DESC)	5/14	7/13	7/14	8/11	5/15
#MaxLength (TOKEN/DESC)	11/69	15/55	21/57	20/73	13/93
#VocabSize (TOKEN/DESC)	1268/2268	885/1683	959/3229	1025/1838	729/1341

4.2 Training

We pre-process our datasets with python and implement our model by Keras 1.0.3 with the backend of Theano 0.8.3. All the weights in our model, including the word embeddings, are randomly initialized. The dimension is set to 256 for embeddings of words, tokens and descriptions. The number of units in *LSTM* layer is set to 256. The two 1D-convolution layers use convolutional kernels in the number of 32 and 64 respectively. The size of the convolutional window is set to 3 words for both convolutional layers. And the max-pooling window is set to be 2 words. Two kinds of score functions are used to measure the similarity of token-description pairs: cosine similarity and flipped Euclidean distance (i.e. negative values of the original euclidean distance). We train the model with 100 epochs for each dataset.

4.3 Results of Description Selecting

Here we explore the quantitative metrics to evaluate the learnt API embeddings of API tokens via our model. Since we expect to capture the mapping relations of API tokens and descriptions after learning embeddings for the two part, we propose a description selecting task.

The description selecting task is defined as follows. For each API token in the test set, we mix its original matched API description with k noisy descriptions sampled in Sect. 3.5 as candidates. Using the tRNN's output as embeddings of API tokens and the dCNN's output as embeddings of the descriptions, we compute the similarity *score* between the given API token and candidate descriptions. Then we rank their matching scores and compute the accuracy by identifying the correct description as the top one result.

We use Bag-of-Words (BoW) model as our baseline. BOW uses one-hot embeddings for words in tokens and descriptions and takes the average embeddings of all words in a sequence as the embedding of the entire sequence.

Table 3. Accuracy (%) of description selecting task

API library		Java SE 8	Eclipse Platform 4.3	Spring 4.3	Lucene 6.1	Java EE 7
Cos Score	BoW	87.90	88.43	90.19	90.20	86.15
	OurWork	93.45	93.35	91.71	91.43	87.69
EucScore	BoW	72.54	72.25	78.10	62.44	72.30
	OurWork	86.65	81.50	87.86	86.94	75.90

Table 3 shows the accuracy on five datasets. Here we mixed one correct description and three noisy descriptions as candidates. Our model runs 100-epoch training on each dataset. "Cos Score" means that the model is trained with the score function of cosine similarity and "EucScore" with the flipped Euclidean distance. Our work outperforms the baseline on both cosine score and Euclidean score. Then we can infer that using cosine similarity gets better results in all the datasets and more training samples lead to better results after the same training epochs. Also, more training samples the model is trained with, more improvement our model gets than the baseline.

We present results of several test cases in Table 4. The correct description is marked with * and bold fonts. *CaseA* shows a correct selecting case by both BoW and our work. Because the correct description share some keywords like "channel" and "write" with the given token, while other candidates do not. *CaseB* gives a case that can be identified correctly by our work while Bow cannot. In this case, there's no shared keywords between the given token and all the candidates. And our work could capture the semantics expressed by the token itself and its description effectively. *CaseC* shows a case failed by BoW and our work. Both models capture the semantics from keywords like "error","factory", "configuration", but they both failed to identify the core semantics of the keyword "transformer".

All in all, the results of Tables 3 and 4 show that the learnt API embeddings could identify the correct description with high accuracies and our model could capture semantics of tokens and descriptions effectively.

Table 4. Results of several test cases

CaseA: **java.nio.channels.NonWritableChannelException**
***Unchecked exception thrown when an attempt is made to write to a channel that was not originally opened for writing**
Checked exception thrown when an input character (or byte) sequence is valid but cannot be mapped to an output byte (or character) sequence
Runtime exception thrown when a file system cannot be found
Checked exception thrown when an input byte sequence is not legal for given charset, or an input character sequence is not a legal sixteen-bit Unicode sequence
CaseB: **java.nio.charset.Charset**
***A named mapping between sequences of sixteen-bit Unicode code units and sequences of bytes**
A multiplexor of SelectableChannel objects
A selectable channel for stream-oriented connecting sockets
A token representing the membership of an Internet Protocol (IP) multicast group
CaseC: **javax.xml.transform.TransformerFactoryConfigurationError**
Thrown when a problem with configuration with the Schema Factories exists
Thrown when a problem with configuration with the Parser Factories exists
An error class for reporting factory configuration errors
***Thrown when a problem with configuration with the Transformer Factories exists**

5 Conclusion

In this paper, we propose a neural model to learn embeddings of API tokens, which is important for processing API dependent programs with deep learning. Our model combines a recurrent neural network with a convolutional neural network and integrates a tree-based negative sampling method in training. The experimental results show that learnt embeddings of API tokens capture semantics expressed by API documents effectively.

Acknowledgements. This research is supported by the National Basic Research Program of China (the 973 Program) under Grant No. 2015CB352201 and the National Natural Science Foundation of China under Grant Nos. 61232015, 91318301, 61421091, and 61502014.

References

1. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. arXiv preprint arXiv:1602.03001 (2016)
2. Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. J. Mach. Learn. Res. **3**, 1137–1155 (2003)

3. Chung, J., Gülçehre, C., Cho, K., Bengio, Y.: Gated feedback recurrent neural networks. CoRR, abs/1502.02367 (2015)
4. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. J. Mach. Learn. Res. **12**, 2493–2537 (2011)
5. Graves, A., Mohamed, A.R., Hinton, G.: Speech recognition with deep recurrent neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 6645–6649. IEEE (2013)
6. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API learning (2016)
7. Gutmann, M.U., Hyvärinen, A.: Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. J. Mach. Learn. Res. **13**, 307–361 (2012)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
10. Kalchbrenner, N., Grefenstette, E., Blunsom, P.: A convolutional neural network for modelling sentences (2014)
11. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
12. Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. ICML **14**, 1188–1196 (2014)
13. Ling, W., Grefenstette, E., Hermann, K.M., Kocisky, T., Senior, A., Wang, F., Blunsom, P.: Latent predictor networks for code generation. arXiv preprint arXiv:1603.06744 (2016)
14. Mnih, A., Teh, Y.W.: A fast and simple algorithm for training neural probabilistic language models. arXiv preprint arXiv:1206.6426 (2012)
15. Mou, L., Li, G., Jin, Z., Zhang, L., Wang, T.: TBCNN: a tree-based convolutional neural network for programming language processing. arXiv preprint arXiv:1409.5718 (2014)
16. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Thirtieth AAAI Conference on Artificial Intelligence (2016)
17. Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., Jin, Z.: Building program vector representations for deep learning. In: Zhang, S., Wirsing, M., Zhang, Z. (eds.) KSEM 2015. LNCS (LNAI), vol. 9403, pp. 547–553. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25159-2_49
18. Schmidhuber, J.: A local learning algorithm for dynamic feedforward and recurrent networks. Connect. Sci. **1**(4), 403–412 (1989)
19. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
20. Wang, B., Liu, K., Zhao, J.: Inner attention based recurrent neural network for answer selection. In: The Annual Meeting of the Association for Computational Linguistics (2016)

21. Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C.: From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering, pp. 404–415. ACM (2016)
22. Zaremba, W., Sutskever, I.: Learning to execute. arXiv preprint arXiv:1410.4615 (2014)
23. Zhou, G., He, T., Zhao, J., Hu, P.: Learning continuous word embedding with metadata for question retrieval in community question answering. In: Proceedings of ACL, pp. 250–259 (2015)