2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Deep-AutoCoder: Learning to Complete Code Precisely with Induced Code Tokens

Xing Hu*†‡, Rui Men*†‡, Ge Li*†§ and Zhi Jin*†§
*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
† Institute of Software, EECS, Peking University, Beijing, China
Email: {huxing0101,menrui,lige,zhijin}@pku.edu.cn

***Abstract*—Code completion is an essential part of modern IDEs. It assists the developers to speed up the process of coding and reducing typos. In this paper, we exploit the deep learning technique called LSTM to learn language models over large code corpus and make predictions of code elements. Unlike natural language, the innumerable identifiers lead to the vocabulary explosion and more difficult to predict. Therefore, we propose a new approach, the Induced Token based LSTM, to deal with the massive identifiers, thus decrease the vocabulary size. In order to induce the code tokens, we present two approaches, one is a constraint character-level LSTM and the other one is encoding identifiers with various preceding context before feeding them into a token-level LSTM. Based on the two approaches, a tool named *Deep-AutoCoder* is developed and evaluated in two classic completion scenarios, that is, method invocation completion and random completion. The experiment results indicate that *Deep-AutoCoder* outperforms the state-of-the-arts on method invocation completion and random code completion. Additionally, the empirical results of *Deep-AutoCoder* indicate that reducing the size of vocabulary can effectively improve the precision of code completion.**

***Index Terms*—Code Completion; Deep Learning; Language Models;**

I. Introduction

Code completion, as a vital part of code generation, has become an essential feature in IDEs. It is one of the top 10 commands used by developers [1]. Intelligent code completion can help to accelerate software development by eliminating costly errors and suggesting accurate code elements including identifiers, method invocations, constants, etc. However, existing intelligent code completion engines list candidates in the alphabetical order or the usage frequency. In most cases, the desired code elements are not listed in first few candidates and programmers have to choose from these candidates.

To improve the performance of code completion, a number of related techniques have been proposed. Asaduzzaman et al. [2] present a context-sensitive example-based tool, CSCC, to support the method invocations completion. It adopts the top four lines to determine the context of a method call. However, the features captured from the four lines are limited for making predictions of method invocations. Some studies have tried artificial intelligence techniques for code completion. Artificial intelligence techniques especially language models have been exploited to important software engineering tasks including code generation [3, 4, 5, 6, 7], source code summarization [8, 9], code clone detection [10], etc. Language models have been widely used in various Natural Language Processing (NLP) problems, such as Machine Translation [11], Text Summarization [12], and Dialogue System [13]. Recent attempts to addressing code completion using language models include n-gram models [14, 15, 16], log-bilinear models [17, 18], etc. These language models make predictions of program elements based on a fixed set relatively features (e.g., the n code tokens that precede the prediction). Such features are often a poor choice because they capture only local dependencies of the element to be predicted.

Programming language and natural language are very different. Identifiers take up the majority of source code tokens [19], hence play an important role in code completion. These identifiers can be single characters or composed of several words, thus lead to identifier explosion. Identifier explosion has disastrous influence on the vocabulary size of language models. As shown in Table I, we count the total tokens, the number of lines, and the vocabulary size of different languages. For natural language, we take the widely used machine translation dataset WMT-14 [20]. The statistic results indicate that the vocabulary ratio of programming languages are much larger than natural language. The vocabulary size of programming languages is at least twice as much as natural languages'. Furthermore, the vocabulary size increases rapidly as the total tokens growing.

In case of the explosion of identifiers, we propose a new approach, Induced Tokens based Long-Short Term Memory (LSTM), to code completion. Inducing tokens, i.e., encoding identifiers, aims to decrease the number of identifiers. To induce tokens, we take an empirical study on various approaches including character-level LSTM and token-level LSTM with identifier encoders. By inducing tokens, the vocabulary size declines significantly. Based on the Induced Tokens based LSTM, we develop a tool called *Deep-AutoCoder*. *Deep-AutoCoder* is applied to classic code completion tasks, method invocations completion on Java language and random completion on C language. The results demonstrate that *Deep-AutoCoder* outperforms prior works on method invocations completion. Furthermore, the *Deep-AutoCoder* can effectively complete code elements randomly. The main contributions of this paper are as follows:

- This paper proposes a new approach, Induced Tokens

‡Equal contribution.
§Corresponding authors.

TABLE I
STATISTICS FOR TOTAL TOKENS TOGETHER WITH VOCABULARY SIZE FOR PROGRAM LANGUAGE AND NATURAL LANGUAGE

	Total Tokens	Number of Lines	Vocabulary Size	Ratio[1](%)
WMT-14 English dataset	304,000,000	12,000,000	160,000	0.05
WMT-14 French dataset [20]	348,000,000	12,000,000	80,000	0.02
C Programs from POJ	9,210,000	1,740,000	11,872	0.13
Python Projects from Github[2]	10,573,158	2,893,030	110,856	1.05
Eclipse 3.5.2	12,095,591	2,542,271	291,671	2.41

[1] $Ratio = \frac{Vocabulary\ Size}{Total\ Tokens} \times 100\%$
[2] Python projects that exploit Tensorflow framework

based LSTM, to code completion problems. The language models learned from large codebases can accurately predict method calls and random code completion.

- We propose two approaches to induce tokens, i.e., the constraint character-level LSTM and the token-level LSTM with identifier encoders to deal with massive identifiers.
- An implementation of our approaches called *Deep-AutoCoder* and evaluation the approaches on large-scale programs.

II. MOTIVATION AND BACKGROUND

We begin by introducing the problem of Code Completion and its challenge. Then, we explain the language model that we use in this paper.

A. *Code Completion*

Code completion is one of the main features in modern IDEs, such as Eclipse and IntelliJ IDEA. It is widely used in method invocations completion and keywords completion. Most code completion techniques take advantage of static information to analyze the code elements. Then the code completion engines list all candidates in alphabetical order. Programmers spend a lot of time searching the right code elements through up and down keys from numerous candidates.

To motivate our approach and obtain a better intuition, consider the following code snippet.

```
void dispose (boolean remove){
      if(index<0)
            return;
      if(remove)
            browser.webBrowser.
                  destroyFunction(this);
      browser=null;
      name=functionString=null;
      index=-1;
}
```

It is a method that aims to "remove a browser" and it should invoke the method *destroyFunction* of *webBrowser* object. Through analyzing the static information of the object "webBrowser", the code completion engine of Eclipse lists all methods that *webBrowser* declares in alphabetical order. However, the desired method *destroyFunction* doesn't occur in the Top10 method candidates. Furthermore, it is impossible for existing code completion engines to predict identifiers or keywords without typing the first few letters of them. These code completion engines omit the context information of the partial programs and list the candidates in alphabetical order. In addition, they cannot complete code elements randomly which is an urgent issue. In this paper, we exploit deep learning approaches to address these issues. We focus on two scenarios of code completion: (1) The completion of method invocations because method invocations account for a large proportion of the programs; (2) The random completion including the identifiers, keywords, etc.

B. *Embedding for identifiers*

Naturally, neural networks take words from a vocabulary as input and embed them as vectors into a lower dimensional space, that is, word embedding [21, 22]. Word embeddings are the weights of the first layer, which is usually referred to as Embedding Layer. Generating word embeddings with a very deep architecture is simply too computationally expensive for a large vocabulary. Unlike words in natural language, programmers define various identifiers. As shown in Table I, the vocabulary ratio of programming languages significantly higher than natural language. With the same order of magnitude, the vocabulary size of programming languages is at least twice that of natural language. The vocabulary size increases even more significantly along with the increasing of total tokens. Building a language model for source code with the huge vocabulary is challenging. Therefore, inducing tokens is an urgent issue for better exploiting the language models to code completion.

Identifiers take up the majority of source code tokens, as shown in Table II. Inducing identifiers is an effective approach to decrease vocabulary. An identifier can be a combination of several words or just a meaningless character. These definition rules of identifiers are helpful to encode the identifiers. In this paper, we propose two induced tokens approaches.

- A constraint character-level LSTM to model the source code. Characters in source code are significantly less than the tokens. When applied to method invocations completion, the generation is limited into a smaller space by adding constraints.
- A token-level LSTM with an encoder that encodes identifiers based on preceding context. The identifiers are represented by the preceding context, e.g., their types, previous tokens, and the index of the identifier.

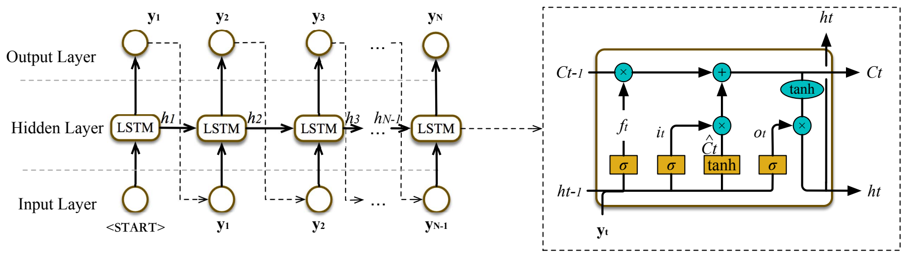


Fig. 1. An illustrate of a basic LSTM

TABLE II
STATISTICS OF VOCABULARY TOKENS FOR C PROGRAMS

Token Type	Token counts	Proportion
Identifiers	11,746	98.6%
Keywords	126	1.4%
Total Tokens	11,872	100%

C. Language Model

Our work is inspired by the language models on NLP problems. Hindle et al. [23] have addressed programming languages are repetitive and predictable like natural language. A language model essentially assigns a probability to an utterance. And for us, "utterances" are programs. By calculating a probability distribution of code sequences and decreasing the entropy of this distribution, we will often be able to guess with high confidence what follows the prefix of a code sequence.

The language model is a probabilistic model leaning how to generate a language. For a sequence $y = (y_1, y_2, ..., y_N)$, the language model aims to estimate the probability of it.

$$p(y_1 y_2 ... y_N) = \prod_{t=1}^{N} p(y_t | y_1 ... y_{t-1}) \tag{1}$$

where N is the number of tokens in the sequence. Therefore, how to compute $p(y_t|y_1...y_{t-1})$ is the vital problem we should consider.

In this paper, we adopt a language model based on the deep neural network called LSTM. LSTM is one of the state-of-the-art Recurrent Neural Networks (RNN). Unlike N-grams that predict a token based on a fixed number of preceding tokens, RNN [24] can predict a token by preceding tokens with longer distances. RNN is a natural generalization of feedforward neural networks to sequences. It can in principle map from the entire history of previous inputs to each output. As the Figure 1 shown, its chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. It's the natural architecture of neural network to use for such data. Additionally, LSTM outperforms general RNN because it is capable of learning long term dependencies. It works tremendously well on a large variety of problems and is now widely used [20, 25]. Therefore, we adopt the LSTM to learn the language models for source code.

The LSTM includes three layers, i.e., an input layer, a hidden layer and an output layer. The input layer maps each token in the sequence to a vector. By reading each vector, the hidden layer computes and updates the hidden states. Then the output layer estimates the probabilities of the following token given the current hidden state. LSTM introduces a structure called the *memory cell* to solve the problem that ordinary RNN is difficult to learn long-term dependencies in the data. A *memory cell* is composed of four main elements: an input gate, a neuron with a self-recurrent connection (a connection to itself), a forget gate and an output gate.

At time stamp t, the memory c_t and the hidden state h_t is updated with the following equations

$$i_t = \sigma(W_i y_t + U_i h_{t-1}) \tag{2}$$

$$f_t = \sigma(W_f y_t + U_f h_{t-1}) \tag{3}$$

$$o_t = \sigma(W_o y_t + U_o h_{t-1}) \tag{4}$$

where i, f and o are gate activations. σ is the sigmoid layer that outputs numbers between zero and one, describing how much of each component should be let through. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The old cell state c_{t-1} is updated into the new cell state c_t:

$$\hat{c_t} = \tanh(W_g y_t + U_g h_{t-1}) \tag{5}$$

$$c_t = i_t \otimes \hat{c_t} + f_t \otimes c_{t-1} \tag{6}$$

where $\hat{c_t}$ is a vector of new candidate values created by a tanh layer. Finally, the $P(y_t|y_1...y_{t-1})$ is predicted according to the current hidden state h_t

$$h_t = o_t \odot tanh(c_t) \tag{7}$$

$$p(y_t|y_1...y_{t-1}) = g(h_t) \tag{8}$$

During training, g is learned from data to minimize the error rate of the estimate y.

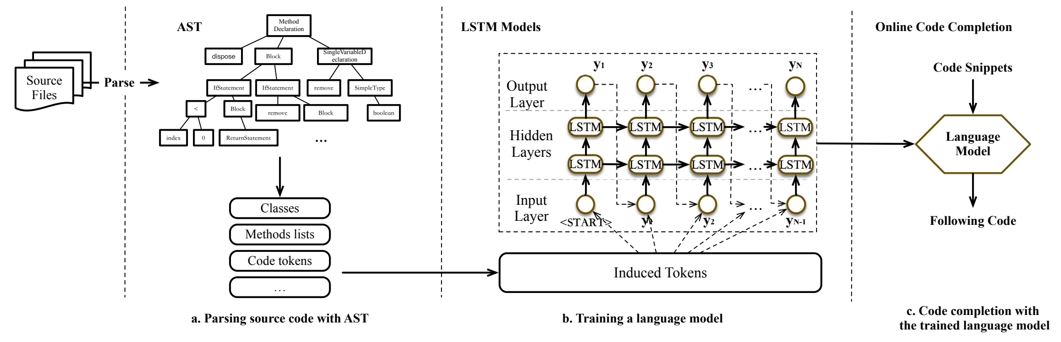


Fig. 2. The Overall Workflow of code completion

III. Deep-AutoCoder

A. Overview

In this section, we describe *Deep-AutoCoder*, an implementation of the induced tokens based LSTM. Deep learning approaches have been proved to be good at capturing useful features and building the maps from input to output automatically. *Deep-AutoCoder* adopts the Induced Tokens based LSTM for the task of code completion. The language model is trained over the programs extracted from large-scale code corpora and then predict the code elements.

Figure 2 shows the overall architecture of *Deep-AutoCoder*. It mainly consists of three steps, source code processing phase, offline training phase, and an online completion phase. The source code have to be parsed into specific forms that are suitable for various approaches. In this paper, we adopt Abstract Syntax Tree (AST) to parse the source code. Then induce the parsed tokens with different approaches, i.e., a constraint character-level LSTM and a token-level LSTM with preceding context identifier encoder, in different scenarios respectively in the following subsections. The programs in training corpus are trained by deep learning models, i.e., the two-layer LSTM as describe in Section II. During the prediction phase, the inputs of the learned model are partial programs and the outputs are the recommended code elements for the specific program contexts.

B. Constraint character-level LSTM

Deep-AutoCoder adopts a constraint character-level LSTM with induced tokens to method invocations completion. Recommendation of method invocations is a vital part of code completion because of the high frequency usages [26]. Many IDEs, e.g., Eclipse and IntelliJ, automatically list all the available member functions when programmers types a dot after an object. The programmers can then select the appropriate method call from the list. These method candidates are listed in alphabetical order or according to the frequency considering the programmers usage. Choosing the appropriate method from the listed candidates is time consuming for programmers. In order to make more accurate recommendations, *Deep-AutoCoder* adopts a constraint character-level LSTM model for method calls prediction. The Induced Tokens based LSTM model is a variant of the basic LSTM model that introduced in Section II.

Figure 3 illustrates the details of the constraint character-level LSTM model using the program example shown above. The model takes code characters instead of tokens of source code as inputs. For example, the input sequence is the characters *browser.webBrowser* and its one-hot vectors are $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_T}$ and $\mathbf{x_i} \in \{0,1\}^{|D|}$, where D is the number of the characters in vocabulary. We assume that each line of the code has at most 100 characters. Therefore, $T = N \times 100$ where N represents the Line of Code (LOC). h_i represents the hidden state of LSTM cell at the current time stamp and is computed based on the previous LSTM cell hidden state h_{i-1} and the current input embedding as

$$h_i = f(\mathbf{x_i}^T \mathbf{E}, h_{i-1}; \theta)$$

where $\mathbf{E} \in \mathbb{R}^{|D| \times H}$ is a word embedding matrix and H is the embedding dimensionality for code. Finally, the partial program $\mathbf{x_1}, ..., \mathbf{x_T}$ is encoded into a fixed-length vector $\mathbf{c}$.

When generating a method name character by character, we add constraints into *Deep-AutoCoder*. *Deep-AutoCoder* extracts the object that intents to invoke a method and the class it is. Then, it gets all methods the class declares by traversing the AST. The details of the extraction process will be given later. By adding constraints, the generating space is limited into these possible methods. Within the generating scope, *Deep-AutoCoder* predicts the first character of the method name $y = g(\mathbf{c})$. The following characters are predicted by the procedure described in Section II. Instead of generating one character sequence, *Deep-AutoCoder* lists all the possible candidates according to their probabilities. Each branch in the predicting procedure is an LSTM and in each time step *Deep-AutoCoder* sorts the probabilities of the generated characters. As shown in Figure 3, $(d, e, j, ...)$ are first characters of method name candidates sorted by their probabilities and the following characters are generating with the same rules. Finally, the Top 1 method name is *destroyFunction* which is the appropriate method invocation with the context.

C. Token-level LSTM with preceding context identifier encoder

Completion at all possible positions while programming is an ideal result of artificial intelligence for code completion. It

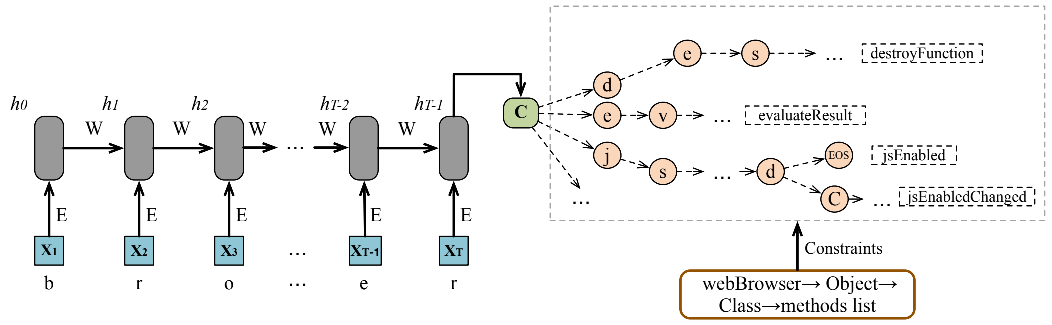


Fig. 3. The constraint character-level LSTM for method calls completion

is much harder than the completion of method calls, because the method invocation space can be limited into the declared methods of the specific class. The generation over the large-scale vocabulary is challenge for LSTM. In order to decrease the vocabulary size, we present several approaches to induce identifiers. These approaches aim to encode identifiers with preceding context.

Programmers often declare these identifiers according to their context information. For example, identifiers in C language are defined by simple characters $for(int\, i; i < n; i{+}{+})$, tokens related to their types *char *chars*, or combination of words *FILE *file_to_read*. Their textual information is meaningless in expressing programs' semantics. Therefore, the context information is able to represent identifiers to a large extent. *Deep-AutoCoder* adopts the preceding context to encode the identifiers and greatly reduces the user defined identifiers in the vocabulary. We give empirical results for different preceding context to encode identifiers. In this paper, we evaluate the following approaches to encode identifiers.

Index The identifiers in a program are represented by index 1, 2, ..., n. The same identifiers for a program occur in different positions are denoted with the same index. For example, the code snippet $for(int\ i;\ i < 100;\ i++)$ is represented by $for(int\ ID\_1;\ ID\_1 < 100;\ ID\_1++)$.

Type+Index Additionally, we integrate Index with identifiers' Types. Therefore, the code that presents above is represented by $for(int\ INT\_1;\ INT\_1 < 100;\ INT\_1{+}{+})$. By adding identifiers' types, the identifiers can be distinguished by not only positions but also types.

Previous tokens The third approach that *Deep-AutoCoder* adopts to encode identifiers is taking their previous tokens (these tokens are connected by _) to represent them. We evaluate one, two, and three previous tokens to encode the identifiers respectively.

ID To evaluate the upper bound precision of the token-level LSTM, *Deep-AutoCoder* replaces all identifiers with the token **ID**. The for expression is represented as $for(int\ ID;\ ID < 100;\ ID++)$. This encoding method doesn't concern the differences between identifiers. And it give the ceiling of code completion at all possible positions by treating source code as natural language.

After inducing identifiers, the code sequence is fed into the two-layer LSTM that introduced in Section II-C. The language model generates the following token according to the probability distributions given the partial program.

IV. Evaluation

We conduct experiments to answer the following research questions:

- **RQ1.** Does the language model character-level LSTM with constrains have affects on completing the method invocations?
- **RQ2.** Does token-level LSTM with the induced tokens contribute to random code completion? What's the relationship between vocabulary size and completion precision?

A. RQ1. Constraint character-level LSTM for method invocation completion

The most relevant work to recommend method invocations is Asaduzzaman et al. [2]. They develop a code completion tool called CSCC and the tool collects the context of the current method call request and matches it from the example code base. These matching method calls are presented to programmers in descending order of similarity values. However, CSCC only considers top four lines to determine the context of a method call and the limit context information may result in high similarity with lots of matches.

To address these issues, *Deep-AutoCoder* adopts a constraint character-level LSTM to predict the method name to be invoked.

The details of the evaluation are introduced as follows:

1) Dataset Details: For comparison, we use the same dataset Eclipse 3.5.2 and focus on the library API named Standard Widget Toolkit (SWT)[1]. The other seven projects that they evaluate, e.g., Vuze and NetBeans, are too small to train. Comparing different approaches on the Eclipse can still prove that *Deep-AutoCoder* performs well on Java method invocations completion.

The data processing is as follows:

- Parse source code files of Eclipse project into Abstract Syntax Trees (ASTs) using Eclipse's JDT compiler[2];

[1]https://www.eclipse.org/swt/
[2]http://www.eclipse.org/jdt

- Find all method invocations of SWT in each source code files;
- For each method invocation $o.m()$ where o is an instance of a SWT class, we find its object o;
- By traversing the AST, get the class C of the object o;
- Extract all methods $C.m_1, C.m_2, ..., C.m_l$ that C declares.

As describe above, we extract the classes of SWT library and their methods. These extracted methods of specific classes are used as the constraints to the LSTM, that is, the generation space of the language model. Then, we remove all the comments and import statements. The remaining pure source code are used to train and test the language models.

2) Accuracy Measure: To measure the performance of *Deep-AutoCoder* in method invocations completion, we compute the precision, recall and F-measure for Top1, Top3, and Top10 on the test set respectively.

These evaluation criteria are computed as follows:

$$Precision = \frac{|\#Accepted\ recommendation|}{|\#Generated\ recommendation|} \tag{9}$$

$$Recall = \frac{|\#Accepted\ recommendation|}{|\#Method\ invocations|} \tag{10}$$

$$F-measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{11}$$

where $\#Accepted\ recommendation$ is the number of correct recommendations out of the total test cases. $\#Generated\ recommendation$ is the number of recommendations. $\#Method\ invocations$ is the number of method invocations in our test data. Consequently, a recommendation is accepted if and only if the recommended method is identical to the current method appearing in the source code. The higher $Precision$, the more correct code are generated. And the higher $Recall$, the higher coverage the appropriate code elements following the given code snippets.

3) Training Details: Deep-AutoCoder splits all data into a train set, a validation set, and a test set with the proportion of $8 : 1 : 1$. As described in Section III-B, we adopt the constraint character-level LSTM for method invocations' completion with different LOCs varying from 1 to 12. Additionally, to improve the training performance, we exploit a two-layer LSTM to learn the source code.

We set the batch size (i.e., the number of instances per batch) as 20. All models are optimized by RMSprop [27] which divides the gradient by running average of its recent magnitude. The learning rate is set to 5×10^{-4} and it decays after 10 epochs. We set the word embedding and hidden state to 1024. The vocabulary size is 97 and contains the lowercase and uppercase letters and numbers and other symbols that code uses.

For implementation, we use Torch [3], an open source deep learning framework. The model is trained in a server with one Nvidia GTX1080 GPU. The training runs for 50 epochs and takes about 10 hours.

[3]http://torch.ch/

4) Precision of method invocations completion: We evaluate the accuracy of SWT APIs completion on the Eclipse 3.5.2 which is the dataset CSCC used. We calculate the Top 1, Top 3, and Top 10 precision respectively. Table III shows the precision, recall, and F-measure values for four code completion systems including CSCC, a token-level LSTM with constraints, a character-level LSTM with constraints, and *Deep-AutoCoder* in which *Deep-AutoCoder* is a character-level LSTM with constraints. The number of LOC in the brackets means that the model can achieve the highest precision with the LOC. For example, *Deep-AutoCoder* performs best when trained by 12 LOC code snippets.

For 77.9% method invocations, *Deep-AutoCoder* can directly generate the accurate recommendations. And 94.5% method invocations can be recommended in Top 10 candidates. It significantly outperforms the traditional method CSCC. The token-level LSTM with constraints has a lower precision because of the larger vocabulary size of 20,798. The precision of Top1 increases 8.1% by decreasing the vocabulary size. For *Deep-AutoCoder*, the generating space is limited into the classes' methods. However, the generation space of a general character-level LSTM is all values of the source code vocabulary. Compared to general character-level LSTM, the precision of *Deep-AutoCoder* improves 11%. The results indicate that the generation space has a larger influence than vocabulary size for method invocations completion.

5) Precision along with LOCs: We evaluate the relationship between the precision of method invocations completion and LOCs. For token-level LSTM, we train and test models with 5, 10, 15, 20, and 25 LOCs respectively. And from 1 to 12 LOC for character-level LSTMs are trained and tested. The two kinds of LSTMs have different behaviors on the LOC variety. As shown in Figure 4(b) and Figure 4(c), character-level LSTMs are much more sensitive to LOC. Their accuracy has an obvious upward trend along with the increase of LOC. The longer code snippets are helpful to the method invocations recommendation. When LOC less than 6, there is a sharp increase of the precision. And the precision is tend to be plateau greater than 8. Additionally, the Top3 and Top10 precision of *Deep-AutoCoder* is much higher than general character-level LSTM with shorter code snippets (e.g., code snippets less than two lines).

For comparison, the token-level LSTM is trained and tested on 5, 10, 15, 20, 25 LOCs. Figure 4(a) indicates that token-level LSTM is in a plateau from five LOC. It has a slightly rising in 10 and 15 LOC, and in 15 LOC the precision rises to the highest point. Additionally, the precision begins to decline after that.

All these results demonstrate that the length that LSTM can deal with is limit. At first, the precision is increase along with the LOC grows. Because the longer code snippets include more information that method invocation completion needs. However, when the length achieves a maximal point the LSTM can handle, the precision has little change. Because

TABLE III
ACCURACY COMPARISON FOR SWT APIS COMPLETION ON ECLIPSE 3.5.2 BETWEEN PRIOR WORK AND OUR MODELS

	Top1		Top3		Top10		Recall
	Precision	F-Measure	Precision	F-Measure	Precision	F-Measure	
CSCC	0.6	0.75	0.8	0.88	0.9	0.94	1
Token-level LSTM with Constraint (15 LOC)	0.696	0.82	0.835	0.91	0.916	0.96	1
Character-level LSTM without Constraint (12 LOC)	0.669	0.80	0.732	0.85	0.752	0.86	1
Deep-AutoCoder (12 LOC)	**0.779**	**0.88**	**0.891**	**0.94**	**0.945**	**0.97**	1

information of the front code snippets will be lost if the code snippets are too long.

B. RQ2. Token-level LSTM with preceding context encoder for random completion

There are little researches on code completion at all possible positions. *Deep-AutoCoder* adopts the token-level LSTM with preceding context encoder to deal with massive identifiers.

1) Dataset Details: Deep-AutoCoder completes code at all possible positions on a new C language corpus collected from POJ[4]. POJ is an Online Judge System, which contains large amounts of problems from different programming contests. We collect the solutions that programmers submit and process the identifiers in these programs by different preceding context encoders.

The maximum code length is set to 512 and the longer programs will be cut. And all comments are removed before fed into the model.

2) Accuracy Measure: To evaluate the precision of the code completion at all possible positions, we present four precision measures.

- **Total Precision** illustrates the overall performance of various tokens in programs.

$$Total\ Precison = \frac{\#Correct\ tokens}{\#Total\ tokens} \tag{12}$$

- **Identifiers Precision** illustrates the ability of identifiers prediction.

$$Identifiers\ Precision = \frac{\#Correct\ identifiers}{\#Total\ identifiers} \tag{13}$$

- **Keywords Precision** Besides the identifiers, we also evaluate the completion of other tokens such as keywords.

$$Keywords\ Precision = \frac{\#Correct\ keywords}{\#Total\ keywords} \tag{14}$$

- **Distinction** illustrates the ability to distinct different identifiers.

$$Distinction = \frac{\sum Sample\ Distinction}{\#total\ samples} \tag{15}$$

[4]http://poj.org/

and

$$Sample\ Distinction = 1 - \frac{|\#predicted - \#original|}{\#original} \tag{16}$$

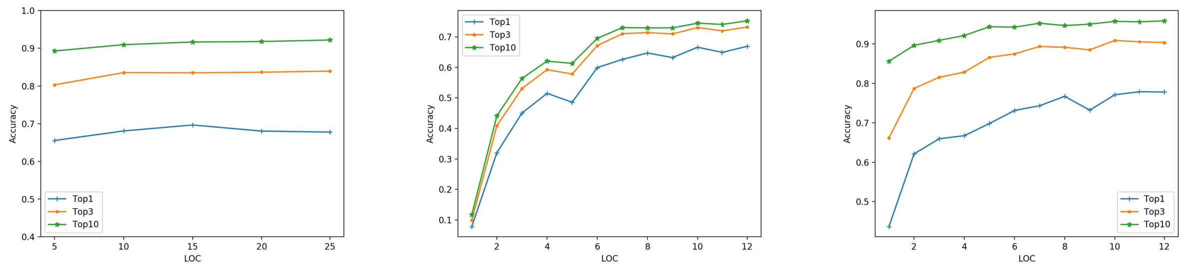
where #predicted and #original mean the identifier numbers in the predicted and original programs.

3) Training Details: Similar to the training process above, *Deep-AutoCoder* splits dataset into three parts and is trained by the Torch framework. *Deep-AutoCoder* trains token-level LSTM with different preceding context encoders respectively. We evaluate all models on 32 and 512 word embeddings respectively and 512 hidden states. Models are optimized by Adam with 32 samples each batch.

4) Precision for different induced tokens approaches: We evaluate different encoders to encode identifiers with different preceding contexts and the results are shown in Table IV. Each model is evaluate on two word embedding dimension, that is, 32 dimension and 512 dimension. Among them, the token-level LSTM takes source code tokens as input without any process. And a token occurring with a frequency of less than 3 is replaced by an *UNK* token and the vocabulary size is 7,439. The others are token-level LSTM that integrates different identifier encoders that represent identifiers with ID token, Index, Type+Index, and Previous tokens. By encoding identifiers with preceding context, the vocabulary size decreases significantly as Table IV demonstrates.

The model integrating ID identifier encoder has the highest precision including the identifier, kewords, and total precision. It can determine the location of 90.99% identifiers accurately. Additionally, it can accurately predict 83.18% keywords which outperforms the other models. However, the distinction is meaningless for the ID identifier encoder, because all the identifiers have no difference by representing as the ID token. In a word, this method is an upper bound of LSTM to code completion without any explicitly using of structural information.

The Type+Index encoder can effectively predict identifiers (80.37%) and keywords (82.16%). At the same time, it effectively differentiates identifiers. Leave the meanings of the identifiers, most programs that generated by *Deep-AutoCoder* can run without bugs. The results of the Type encoder and Type+Index encoders demonstrate that the type information is helpful for both identifiers, and keywords completion.



(a) Token-level LSTM (with constraints) (b) Character-level LSTM (without constraints) (c) Deep-AutoCoder: Character-level LSTM (with constraints)

Fig. 4. Accuracy of method invocation completion along with LOC

Compared to the type and the index information, exploiting the previous tokens to encode the identifiers has lower precision. Additionally, the more previous tokens that encoders use the lower precision.

Overall, reducing the size of vocabulary by encoding the identifiers can effectively improve the completion precision of both identifiers and keywords.

5) Influence of the word embedding dimension: The word embedding dimension also has influence on code completion. When the dimension drops to 32, the precision of all models declines. Among them, the model integrating ID encoder has the least influence. Its precision decreases by 0.5% and the other encoders decrease about 1%. However, the word embedding dimension has a large influence on the precision general token-level LSTM. Its identifiers, keywords, and total precision declines about 3% respectively. That's because the vocabulary of the general token-level LSTM is much larger than other models. The small word embedding dimension can not effectively handle the distribution of large dictionary. Multiple tokens in the dictionary can be mapped to embeddings that are too close to distinct.

6) Influence of vocabulary size: Figure 5 illustrates the relationship of the precision and vocabulary size. With the increase of vocabulary size, the precision of identifiers and keywords reduce significantly for both 32 and 512 word embedding dimensions.

V. DISCUSSION

A. Why does Deep-AutoCoder work?

A major challenge for code completion is innumerable identifiers. Existing context-based approaches and language models such as N-grams usually have no ability to process identifiers and lack a deep understanding of the distribution of source code. We have identified two advantages of *Deep-AutoCoder* that address these problems.

Preceding context based identifier encoder By encoding identifiers with their preceding contexts, the vocabulary size reduces significantly. It is effectively to train a model to complete the identifiers with a bit tokens. To estimate the upper bound of RNN for random code completion, we represent all identifiers as ID token. Although the identifiers of the program that generated by it can not be distinguished, it gives the evidence of at most 14% of the room for improvement. Additionally, we evaluate different encoders that can not only reduce the identifiers but also keep their distinction.

Learning code sequence Another advantage of our approach is that, it can learn common patterns of source code. The models themselves are language models and remember the likelihoods of different programs. The hidden layers of our models have the memory capacity. They consider not only the individual tokens, but also their relative positions in source code.

B. Threats to validity

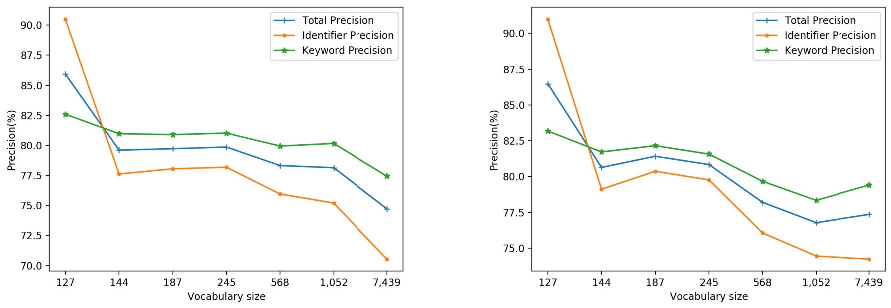
We have identified the following threats to validity:
Source code is statically-typed language The source code investigated in this paper is statically-typed languages. Hence, they might not be representative of dynamic programming languages, e.g., Python and JavaScript. In the further, we will extend the model to other programming languages.
Without explicit structural information In this paper, we leverage the language model to code completion. The language model is trained on the plain code without explicit structural information. In addition, we also estimate the upper bound of the language model for source code. In the future, we will investigate a better model to learn the structure of source code.

VI. RELATED WORK

The last few years have seen a renewed interest in various synthesis techniques which promise to simplify various software development tasks. Asaduzzaman et al. [2] describe a novel technique called Context Sensitive Code Completion (CSCC) for improving the performance of API method call completion. To recommend completion proposals, CSCC ranks candidate methods by the similarities between their contexts and the context of the target call. CSCC only considers framework or library APIs. Apart from method call completion, Asaduzzaman et al. [28] also present a study of the completion of method parameters. This paper leverages source code localness property, static types and previous code examples to recommend method parameters.

TABLE IV
ACCURACY OF DIFFERENT IDENTIFIER ENCODERS WITH PRECEDING CONTEXT

	Dimension	Total Precision(%)	Identifiers Precision(%)	Keywords Precision(%)	Distinction(%)	Vocabulary Size
Token-level	32	74.72	70.53	77.43	86.66	7,439
	512	77.36	74.23	79.41	**87.86**	
Index	32	79.60	77.62	80.97	85.13	144
	512	80.64	79.12	81.72	83.75	
Type+Index	32	79.72	78.05	80.89	84.46	187
	512	**81.42**	**80.37**	**82.16**	84.13	
Previous 1 token	32	79.86	78.18	81.02	86.07	245
	512	80.84	79.77	81.57	84.92	
Previous 2 tokens	32	78.32	75.95	79.94	87.03	568
	512	78.20	76.06	79.67	86.21	
Previous 3 tokens	32	78.14	75.20	80.16	86.46	1,052
	512	76.78	74.45	78.34	87.03	
ID	32	85.93	90.49	82.59	–	127
	512	**86.48**	**90.99**	**83.18**	–	



(a) Influence of vocabulary size on dimension 32 (b) Influence of vocabulary size on dimension 512

Fig. 5. The precision variety with the size of vocabulary on different word embedding dimension

Additionally, some language models have been used in building probabilistic model of code. N-gram model is the most popular and widely used probabilistic model of source code. Due to its simplicity and efficient learning, it is applied to code completion task [14, 15, 16, 23]. Hindle et al. [23] first exploited the N-gram to model source code and developed a code completion engine for Java. It substantially improves upon the existing suggestion facility in the widely-used Eclipse IDE. Soon after, various improvements are proposed to address some of the N-gram limitations. Nguyen et al. [15] develop a code suggestion engine that integrates the semantic N-grams, global concerns, and pairwise association. Tu et al. [16] introduce a cache language model that consists of an N-gram and a cache component to exploit localness. Raychev et al. [14] leverage N-gram and RNN to synthesize programs with holes using APIs.

Furthermore, more complicated language models are gradually used to code completion. Raychev et al. [4] introduce an approach that learns a probabilistic model of code as learning a decision tree in a Domain Specific Language (DSL) over ASTs. The probabilistic model of code automatically determines the right context when making a prediction. Bhoopchand et al. [29] introduce a neural language model with a sparse pointer network to complete the dynamically-typed programming language Python. The pointer network aims at capturing very long-range dependencies.

VII. CONCLUSION

In this paper, we propose a new approach, Induced Tokens based LSTM, to code completion. By inducing the tokens, the vocabulary size decreases significantly. We propose two approaches to decrease the vocabulary size, constraint character-level LSTM and token-level LSTM with identifier encoders. Additionally, a tool named *Deep-AutoCoder* is developed based on these approaches. *Deep-AutoCoder* is applied to two scenarios, method invocations completion and random code completion. Our empirical study has shown that the precision improves significantly by decreasing of vocabulary size. In the future, we will explore the models that can better fit the source code.

ACKNOWLEDGMENT

This research is supported by the National Key R&D Program under Grant No. 2017YFB1001804 and the National Natural Science Foundation of China under Grants No. 61832009, No. 61620106007 and No. 61751210.

REFERENCES

[1] G. C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the elipse ide?” *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.

[2] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "A simple, efficient, context-sensitive approach for code completion," *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 512–541, 2016.

[3] P. Bielik, V. Raychev, and M. T. Vechev, "Phog: probabilistic model for code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML*, 2016, pp. 19–24.

[4] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 731–747.

[5] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[6] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[7] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin, "On end-to-end program generation from user intention by deep neural networks," *arXiv preprint arXiv:1510.07211*, 2015.

[8] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, vol. 1, pp. 2073–2083.

[9] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," *arXiv preprint arXiv:1602.03001*, 2016.

[10] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[11] M. Auli, M. Galley, C. Quirk, and G. Zweig, "Joint language and translation modeling with recurrent neural networks." in *EMNLP*, vol. 3, no. 8, 2013, p. 0.

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

[13] O. Vinyals and Q. Le, "A neural conversational model," *arXiv preprint arXiv:1506.05869*, 2015.

[14] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[15] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 532–542.

[16] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 269–280.

[17] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[18] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language." in *ICML*, vol. 37, 2015, pp. 2123–2132.

[19] M. Broy, F. Deißenböck, and M. Pizka, "A holistic approach to software quality at work," in *Proc. 3rd World Congress for Software Quality (3WCSQ)*, 2005.

[20] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[21] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[22] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation." in *EMNLP*, vol. 14, 2014, pp. 1532–1543.

[23] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[24] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[25] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[26] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 317–326.

[27] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, 2012.

[28] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring api method parameter recommendations," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 271–280.

[29] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," *arXiv preprint arXiv:1611.08307*, 2016.