Language to Code with Open Source Software

Lei Tang,Xiaoguang Mao
College of Computer
National University of Defense Technology
Changsha, Hunan Province, China
Email:{tanglei17,xgmao}@nudt.edu.cn

Zhuo Zhang
College of Computer
National University of Defense Technology
Changsha, Hunan Province, China
Email: zz8477@126.com

Abstract—**With the development of deep learning, it has been applied in various field of computer science. Generating computer executable code from natural language descriptions is an urgent problem in the artificial intelligence. This paper proposed a solution based on deep learning for code generation. Encoder-Decoder model is used in our method to convert natural language description into target code. Because of the rapid development of information technology, all aspects of software resources have been greatly enriched. The deep learning model we designed takes the natural language description as input and generates the corresponding object code by extracting the code from the open source software library. We collected natural language descriptions of 20 problems that undergraduate students often encounter in their daily programming. Experimental results show that our method is practicable. Our approach also provides a good idea to extract useful code from open resource for code generation.**

Index Terms—**code generation, open resource software, LSTM, Encoder-Decoder**

I. Introduction

With the popularity of the Internet and the explosive growth of information technology, demand for new software is exponentially growing, but the productivity of software is far below the needs of people. At the same time, the scale and complexity of software systems are also increasing. However, the resources used to develop such software are not kept in sync. This leads to a software crisis [1]. The traditional software development method exposes the defects in the new technology environment, such as long development cycle, large work repetition, and great difficulties in system evolution and maintenance. How to improve the efficiency of software development to meet people's growing needs is an urgent problem to be solved. At this point, code generation [2][3] technology is born at the right moment. Code generation refers to the program that generates the code, more detailed refers to the use of structured or unstructured program description as input [4], the process of generating standard source code that a computer can understand.

With the prosperity of artificial intelligence, how to make the machine automatically generate programs is an important but challenging problem that we want to solve [4][5][6]. It makes a lot of sense for machine to generate code by itself. This can free the vast majority of coders from the heavy task of programming that allows the programmers has more time to pay attention to the design of system architecture. It will greatly improve the efficiency of software development. In recent years, deep neural networks (DNNs) [7]widely used in various fields of computer science. Including machine translation [8][9] , speech recognition [10]and visual object recognition [11], etc. And achieved very good results, so we wondered if we could apply DNNs to code generation. The idea of using deep learning method to generate code automatically comes from machine translation that is the process of converting a natural language into another natural language using a computer. The essence of code generation is machine translation, but this particular machine translation takes a structured or unstructured description as the source language, the executable code we want to generate as the target language.

As a new way of software development, code generation technology is still in constant development. Research in the academic field that uses deep learning to generate code has become increasingly popular, although code generation is still inadequate in many ways. But in the future, code generation will play a significant role in promoting the development of software, and greatly improved the efficiency of software development. The benefits of code generation are as follows:

- **Reduce repetitive coding efforts**. Automatic code generation can reduce unnecessary repetitive code writing, improve software development efficiency and optimize software development process.
- **The code generation style is consistently good**. When different programmers write modules of the same function, the source code that is finally written will be very different. However, the source code generated by automatic code has good consistency and standardization, as well as good readability.
- **Easy to modify and upgrade**. Another advantage of code generation is scalability. In the long run, automatic code generation technology makes software easier to modify and upgrade.

The classes and amount of software repositories become rich and abundant with the maturing of software industry. Software development cycle is getting shorter and shorter. Writing industrial-scale software from scratch has been difficult to reproduce. Integrated transformation development or evolution based on large-scale reuse has become the mainstream

978-1-7281-0945-9/19/$31.00©2019 IEEE

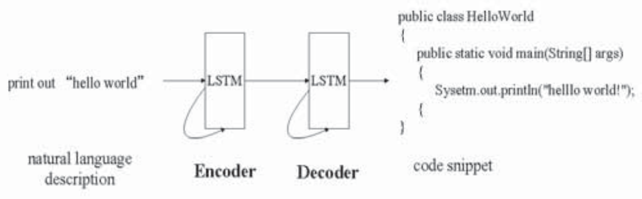
development mode. On the other hand, the comprehensive intervention of information technology in human society. All aspects of the software resources have been greatly enriched. From the perspective of code generation, the possibility of encountering a completely new software feature that has never been seen before is very low. Therefore, it is of great value to learn from existing resources to extract code and generate code automatically.

II. Related Work

In existing code generation techniques. The input can be divided into structured input or unstructured input. Structured input is a very precise input method that relies on structured storage of data, structured input requires users to be clear about their needs. At the same time, users are required to have a certain professional background knowledge, this puts high demands on the users of the system. Contrary to the structured input, unstructured input doesn't require users to have relevant background knowledge, just use your familiar language as input that automatically generate code. And make it possible for people without any programming experience to write programs. Code generation technology will great improve the efficiency of software development that makes automatic code generation possible. And ultimately realize the finally dream of software engineering field—— let the software generate the software.

Code generation is an important application field that artificial intelligence has always hoped to conquer. With the increasing popularity of deep learning, recently it has been widely used in code generation and has made great progress. Traditionally this problem can be called inductive program synthesis (IPS). The goal of the IPS problem is to generate a piece of code automatically from a given set of input-output examples that converts a given input into a given output.

Early research on code generation focused on domain specific language (DSL). Scott et al. proposed the neural programmer-Interpreters [12], it is a recurrent and compositional neural network that learn to represent and execute programs. Matej et al. put forward a method that can solve programming competition-style problems from input-output examples using deep neural networks that called DEEPCODER [13]. Dong et al. propose a general method based on encoder-decoder model with neural attention [14], it can encode input utterance into vector and generate their logical forms by conditioning the output sequences on the encoding vectors. These methods are based on rules and human-defined functions primarily, so it's very limited. Recently researchers have introduced neural networks to generate code in a common programming language. Ling et al. [4] present a novel neural network architecture which generate an output sequence conditioned on an arbitrary number of input functions. However, this method is based on recurrent neural network (RNN), but RNN has proven to be unsuitable for handing long sequence of input [15]. Therefore, the generalization of this method and the accuracy of generated code need to be improved. Based on this method, Sun et al. raised a grammar-based structural convolutional neural network [16] for code generation, and achieved impressive results. Mou et al. [17] envisions an end-to-end program generation scenario using recurrent neural networks. Users express their intention through natural language descriptions, and then RNN generates object code character-by-character. And the input of these two kinds of neural networks is structured or unstructured programming language description. This is very unfriendly for beginner programmers. To make code generation more convenient and practical, we use natural language descriptions as input to generate the executable snippets that we want. Rabinovich et al. present a modeling framework that called abstract syntax networks [5], the outputs in this model are represented as abstract syntax trees (ASTs) and constructed by a decoder. The approach take account for much richer structural constrains on outputs. Yin et al. propose a novel neural architecture powered by a grammar model [18] to explicitly capture the target syntax as a prior knowledge.



```
public class HelloWorld
{
    public static void main(String[] args)
    {
        Sysetm.out.println("helllo world!");
    {
}
```

Fig. 1. Encoder-Decoder model for our method

III. Model

In order to automatic generate code through natural language descriptions, first we need to transform the natural language description into a form that computers can understand through word embedding. Word embedding is then fed into the encoder and a coding vector is generated. Then target code is generated by the encoder. The Fig.1 demonstrate the overall architecture of our model.

Encoder-Decoder structure that based on neural network [8][14] has been widely used in natural language processing, from semantic analysis to machine translation, then to image description generation. The essence of code generation is machine translation, it's just that translate natural language description into code snippets. Therefore, encoder-decoder model is also used in our method to generate code. Encoder-Decoder model consists of two neural networks which are used as encoder and decoder respectively. The encoder maps the natural language descriptions into a vector with variable length. Then the decoder maps this vector back to the target code that we want to generate. The two neural networks trained together to maximize the conditional probabilities from natural language description to code. The neural networks we adoption in our method is Long Short-Term Memory (LSTM) [19], LSTM is an improved RNN. Experiments show that LSTM has a good effect on dealing with long-term dependencies.

A. Formulation

Our goal is to build a model that maps natural language descriptions $d = x_1 ... x_{|n|}$ to a piece of code $c = y_1 ... y_{|m|}$ The conditional probability $p\,(c|d)$ can be decomposed as:

$$p\,(c|d) = \prod_{t=1}^{T} p\,(y_t | y < t, d) \tag{1}$$

where $y < t = y_1 ... y_{t-1}$.

In our method,the encoder which encodes natural language descriptions d into a vector representation, and the decoder then learns to generate code snippet c from open source codebase according to the vector representation. It is important to note that the more similar the natural language description are, the closer the encoding vector are. In this way, semantic similarity between two descriptions can be well expressed, so as to improve the accuracy of code generation.

B. Encoder

For a natural language description d consisting of n tokens, the encoder processes tokens one by one in a recursive manner. Let $h_t \in \mathbb{R}^n$ indicate the hidden vector at time step t, w_t represent the token that input to the model at time step t. So h_t is then computed by:

$$h_t = \text{LSTM}(h_{t-1}, w_t) \tag{2}$$

where LSTM refers to the LSTM function in Keras [20]. From the prospective of encoder, $h_t = \text{W}_q \text{e}(x_t)$ is the word embedding vector of the current input token, with $\text{W}_q \in \mathbb{R}^{n \times |V_q|}$ indicate a parameter matrix, and e(.) is the index of the corresponding token.

C. Decoder

The decoder uses an LSTM network to model the sequential generation of the code snippet we need. Once the tokens of the natural language descriptions $[d = x_1 ... x_{|n|}$ are encoded as a vector, they are delivered to the decoder to initialize the hidden states. Next, the hidden vectors of the topmost LSTM h_t in the decoder is used to predict the output token at time step t as:

$$p(y_t | y_{<t}, q) = \text{softmax}(\text{W}_o \text{h}_t)^{\text{T}} \text{e}(y_t) \tag{3}$$

where $\text{W}_o \in \mathbb{R}^{|V_a| \times n}$ denote a parameter matrix, and $\text{e}(y_t) \in \{0,1\}^{|V_a|}$ mean a one-hot vector for computing y_t's probability from the predicted distribution. Then conditional probability of generating the entire target code from input is calculated by Equation (1).

D. Validation

At test time, we predict the target code for a natural language description d by:

$$\hat{c} = \arg\max_{c'} p(c'|d) \tag{4}$$

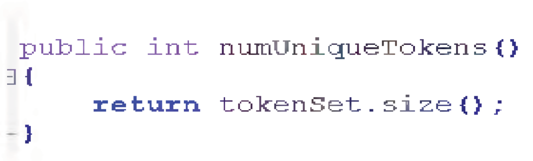
where c' present a possible output. However, it is impractical to iterate over all possible result to obtain the optimal prediction. According to Equation (1), we decompose the probability $p\,(c|d)$ so that we can use beam search to generate code snippet from open source codebase.

IV. Experiments

We train and validate our proposed method on dataset. We describe the dataset in detail below, and show our experimental settings and results.

A. Dataset

To provide annotated data, each piece of open source software code snippets needs to be labeled with a natural language description. This is very labor-intensive if done by hand. Therefore, we used an existing dataset. The dataset we used was established by Gu et al. that was comprising 18233872 commented Java methods [21]. The corpus was collected from open-source projects on GitHub. Each piece of data in the dataset consists of a code snippet and its corresponding natural language description. The dataset was extracted from all Java projects with a star on GitHub from August, 2008 to June, 2016, so it's very representative and very suitable as a training set for code generation. One piece of data in this dataset that contained natural language description and their corresponding code snippets are shown as follows.
return how many unique tokens we have:



```
public int numUniqueTokens()
{
    return tokenSet.size();
}
```

B. Settings

Evaluate our approach by experimenting in the dataset. The dataset involved 100000 pairs of data, the split of train set and test set is 0.2. The epoch of this training is 2000. After the training is completed, we got an optimal neural network model with parameters. Then we validated the model using 20 natural language descriptions that collected from undergraduate students. These descriptions are some of the problems they often encounter in daily programming. Descriptions are as present in TABLE 1.

C. Result

After the train is complete, we get a code generation model from open source codebase with natural language description as input. For each natural language description, we generate a piece of code associated with it. After the code is generated, we compare the generate code with the reference code to determine if they have relationship. The statistical results are shown in Table 1:

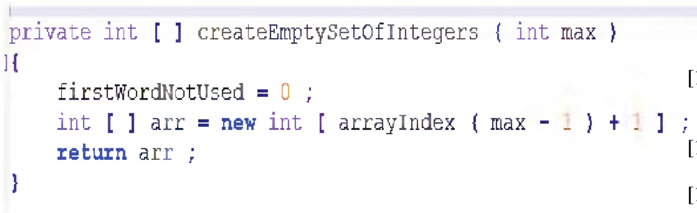
In the status column in table 1, Y represent the code snippet generated from the open source codebase is related to the natural language description of the input. N means that the generated code is independent of the description of the input. As we can see from the table 1, 9 of the 20 natural language descriptions can generate code snippets

TABLE I
STATISTICAL RESULTS OF CODE GENERATION

NO	Natural language description	status
1	Create a new empty set of integers	Y
2	Add an image to a workshop	N
3	Sort the numbers using the bubble algorithm	N
4	Get the package for this class	Y
5	Find a class using the specified class loader	Y
6	Send a message to the console window	Y
7	Set the error flag	N
8	Output an array in reverse order	N
9	Connect this socket to the server	Y
10	Load files from local	N
11	Calculate the factorial of n	N
12	Check if this header contains uncompressed strip	N
13	Create a new empty file in the specified directory, using the given prefix and suffix strings to generate its name	N
14	Print path from source	Y
15	Check if the char sequence contains only whitespace	N
16	Read the next line	N
17	Check if a filename is safe	Y
18	Generate a hash code for multiple values	N
19	Set the format of the data	Y
20	Convert principal attributes to person attributes	Y

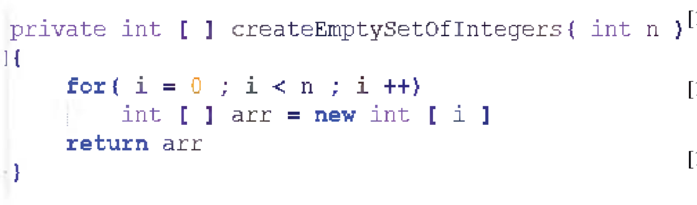
associated with them from the open source codebase. The experimental results show that the idea of extracting code fragments from the open source software library for code generation is feasible. Our approach provides a new way to address the problem of code generation. An example of code generation are as follows:
Natural language description: Create a new empty set of integers

Generated code:



```
private int [ ] createEmptySetOfIntegers ( int max )
{
    firstWordNotUsed = 0 ;
    int [ ] arr = new int [ arrayIndex ( max - 1 ) + 1 ] ;
    return arr ;
}
```

Reference code:



```
private int [ ] createEmptySetOfIntegers( int n )
{
    for( i = 0 ; i < n ; i ++)
        int [ ] arr = new int [ i ]
    return arr
}
```

V. Conclusion

We introduced a deep learning method for code generation from the open source codebase with natural language description as input. Through this model, we first convert natural language descriptions into variable length vector through an encoder, then convert the vector to the target code that we want to generate through the decoder. Validate our model in a java dataset that collected from open source software. Our experimental results prove is feasible that code generation from the open source software codebase. This is a small step toward broad code generation from open source software. It provides a new direction for the further research of code generation.

References

[1] P. Naur, "Software engineering-report on a conference sponsored by the nato science committee garimisch, germany," *http://homepages. cs. ncl. ac. uk/brian. randell/NATO/nato1968. PDF*, 1968.

[2] C. Quirk, R. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 878–888.

[3] T. Lei, F. Long, R. Barzilay, and M. Rinard, "From natural language specifications to program input parsers," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2013, pp. 1294–1303.

[4] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[5] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," *arXiv preprint arXiv:1704.07535*, 2017.

[6] M. Hong and L. Zhang, "Can big data bring a breakthrough for software automation?" *Science China Information Sciences*, vol. 61, p. 056101, 2018.

[7] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[8] 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.

[9] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 2013, pp. 1700–1709.

[10] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2011.

[11] D. Cireşan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *arXiv preprint arXiv:1202.2745*, 2012.

[12] S. Reed and N. De Freitas, "Neural programmer-interpreters," *arXiv preprint arXiv:1511.06279*, 2015.

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

[14] L. Dong and M. Lapata, "Language to logical form with neural attention," *arXiv preprint arXiv:1601.01280*, 2016.

[15] Y. Bengio, P. Simard, P. Frasconi *et al.*, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[16] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," *arXiv preprint arXiv:1811.06837*, 2018.

[17] 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.

[18] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] https://github.com/keras-team/keras/.

[21] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.