Source Code Generation from Task Description by Deep Neural Networks

Dragan Vidaković
Department of Computing and Control Engineering
Faculty of Technical Sciences, University of Novi Sad
Novi Sad, Serbia
vdragan@uns.ac.rs

Milan Keča
Department of Computing and Control Engineering
Faculty of Technical Sciences, University of Novi Sad
Novi Sad, Serbia
milan.keca@uns.ac.rs

Abstract—**Deep neural networks are achieving outstanding results in various difficult tasks, including neural machine translation, where recurrent neural networks achieved state-of-the art results. This paper envisions an end-to-end source code generation using neural machine translation approach. Users define algorithmic tasks using natural language, and encoder-decoder recurrent neural network automatically generates source code in a character-by-character fashion. We demonstrate its feasibility through a Python source code generation from description of basic algorithmic tasks. Results show that this approach represents a strong base for end-to-end program generation. However, much long-term research shall be addressed in this field in order to achieve industry-level usage.**

Keywords—Deep learning; Source code generation; Recurrent neural networks;

I. Introduction

Automatic program generation has long been the dream of software engineering, and is closely related to a variety of software engineering tasks [1]. However, traditional approaches are typically weak in terms of automation and abstraction. An end-to-end process where the machine automatically reads a human intention expressed in natural language and generates high-quality source code, well commented and documented, will fulfill the dream. More realistic process should generate flexible and (almost) correct source code: it should satisfy the syntax, and implement desired functionality. The code should be usable with a little post-editing.

Machine automatically reads a natural language sentence character-by-character to capture user intention, and then generates code in similar fashion. Therefore, one approach in addressing the problem of source code generation could be machine translation. Nowadays, as source code and code documentation have become Big Data, it’s easy to provide sufficient training data for neural machine translation approach. Neural machine translation [2][3] attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation.

Most of proposed neural machine translation models belong to family of encoder-decoders [2][4], with an encoder and a decoder for each language. An encoder neural network reads and encodes a source sentences into a fixed-length vector. A decoder then outputs a translation from the encoded vector. The whole encoder-decoder system is jointly trained to maximize the probability of a correct translation given a source sentence [5]. A potential issue with encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector. This may make it difficult for the neural network to cope with long sentences, and longer sentences than ones in the training corpus will deteriorate performance of the network. [5] Applying appropriate data preprocessing in training process could address the following problem.

In this paper, we proposed encoder-decoder approach on the task of Python source code generation from task description. In Section II, several related works are presented and discussed. Section III contains proposed neural network architecture. In Section IV the experimental process is described, and experimental results are presented and analyzed. Finally, we make a brief concluding mark and give the future work in Section V.

II. Related Work

The usage of neural networks in machine translation started with a neural probabilistic model which uses a neural network to model the conditional probability of a word given in a fixed number of preceding words [6]. However, the role of neural networks has been largely limited to simply providing a single feature to an existing statistical machine translation system or to re-rank a list of candidate relations provided by an existing system [5].

Kalchbrenner and Blunson [7] were the first to map the input sentence into a vector and then back to a sentence. They have used convolutional neural network, which lost the ordering of the words. Cho et al. [4] used recurrent neural network (RNN) architecture to map sentences into vectors and back, although the primary focus was integration of their model into existing statistical machine translation system.

Sutskever et al. [3] proposed a large deep Long-Short Term Memory (LSTM) based RNN that outperformed a standard statistical machine translation system using a limited vocabulary. However, very long sentences significantly downgraded their translation accuracy.

Mou et. al [8] proposed an end-to-end program generation scenario using RNNs to generate almost executable, functionally coherent source code. They envisioned several scenarios where such technique can be applied, but they haven't proposed metrics and automatic evaluation.

III. Model

The RNN is a natural generalization of feedforward neural network to sequences. It can easily map sequences to sequences whenever the alignment between the input and output is known ahead of time. However, it is not clear how to apply an RNN to problems whose input and the output sequences have different lengths. The simplest strategy for general sequence learning is to map the input sequence to a fixed size vector using one RNN, and then to map the vector sequence with another RNN. [3] As RNN with LSTM units already achieves close to the state-of-the art performance on the conventional phrase-based machine translation system, we adopted this strategy in our approach.

We used two different LSTMs: one for the input sequence and another for the output sequence. This increases the number model parameters at negligible computational cost and makes it natural to train the LSTM easier [3]. As shown in Figure 1, our encoder contains one LSTM layer, and our decoder four LSTM layers.

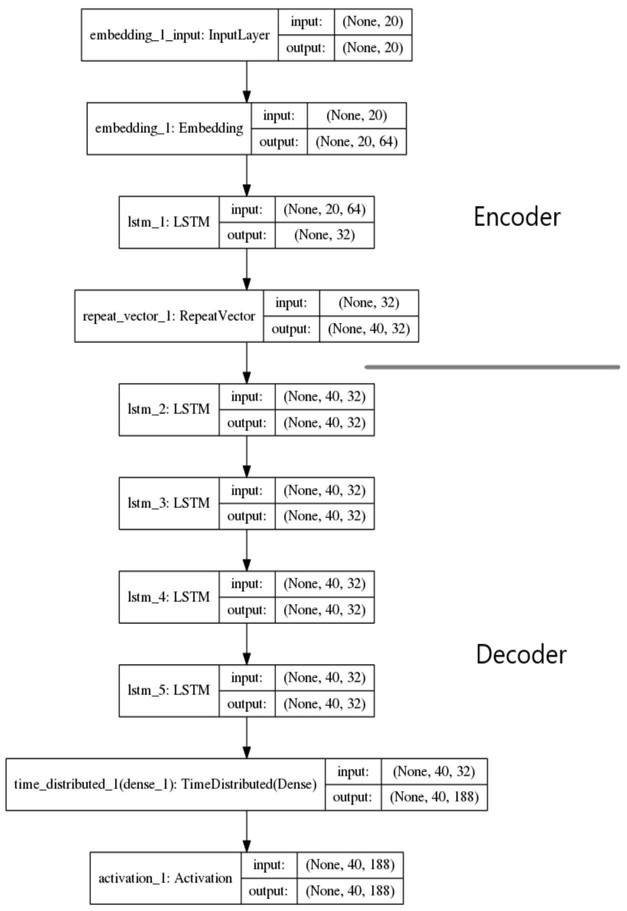
embedding_1_input: InputLayer | input: (None, 20) | output: (None, 20)
embedding_1: Embedding | input: (None, 20) | output: (None, 20, 64)
lstm_1: LSTM | input: (None, 20, 64) | output: (None, 32)
Encoder
repeat_vector_1: RepeatVector | input: (None, 32) | output: (None, 40, 32)
lstm_2: LSTM | input: (None, 40, 32) | output: (None, 40, 32)
lstm_3: LSTM | input: (None, 40, 32) | output: (None, 40, 32)
lstm_4: LSTM | input: (None, 40, 32) | output: (None, 40, 32)
Decoder
lstm_5: LSTM | input: (None, 40, 32) | output: (None, 40, 32)
time_distributed_1(dense_1): TimeDistributed(Dense) | input: (None, 40, 32) | output: (None, 40, 188)
activation_1: Activation | input: (None, 40, 188) | output: (None, 40, 188)

Figure 1. Model architecture

IV. Experiments

We applied our method to the Python source code generation based on description of simple algorithmic tasks. In this section we describe experimental data, train process and accuracy of presented model. Also, we present simple source code that has been generated.

A. Dataset

Dataset was manually collected and includes 4285 Python source code files with descriptions and solutions to simple algorithmic tasks including conditions, loops, lists, strings and logic. Tasks were downloaded from CodingBat[1], where the accuracy of solutions was confirmed. In order to generate more training data, we transformed initial generic task to use the test cases data. After this transformation, every task produced 5 new tasks, which are easier to solve. Table 1 contains example of this step. Dataset contains 166176 characters of task descriptions and 114065 characters of source codes.

Original task	Given an array of ints, return True if the sequence of numbers 1, 2, 3 appears in the array somewhere.
Generated task 1	Print True if the sequence 1, 2, 3 appears in the given array [1].
Generated task 2	Print True if the sequence 1, 2, 3 appears in the array [1, 1, 2, 1, 2, 3].

Table 1. Example of a task transformation

B. Training

Dataset was split in 80% - 20% ratio, following by total characters count. In order to obtain best possible results, test set contains tasks from different categories, mentioned in previous section.

First step was to split task descriptions and source codes into list of words. It was important to treat special characters as words in output text, since they are important for Python syntax (e.g. '[', '+', ':'). Word order in the input sentences was reversed to enhance encoder's performance [3]. Since input and output sentences had to have the same length, we fixed input and output length.

Next step was to create input and output vocabularies. Input vocabulary would hold all words from natural language, and output vocabulary would hold all words from Python codes. Each vocabulary was expanded with two special words: 'ZERO' is a neutral word used for padding, and 'UNK' is a word for replacing all words not in the vocabulary. All input and output sentences were padded with 'ZERO' and had unknown words replaced with 'UNK'. One hot encoding was used for vectorization of input and output, where 1's in vectors were the indices of the word in vocabulary.

[1] http://codingbat.com/python

LSTM in encoder has 32 units and word embedding with output dimension of 64. Decoder LSTMs has 32 units. We used Adam optimizer [9], and our batch size was 32. Network contains 67196 trainable parameters.

C. *Results*

In order to evaluate the quality of generated source code, we measured if generated source code compiles or not. Because of neural machine translation approach, we used the BLEU score [10] as another metric.

As already mentioned, neural network of this type has issues with longer sentences [5]. Therefore, we performed experiments with longer and shorter sentences. As shown in Table 2, our model confirms this cognition.

Input sentence size	**BLEU**	**Compilation**
10	0.42	0.8
20	0.20	0.1

Table 2. Experimental results

Our experimental result shows that code can be compiled, but translation score must be improved. As we can see in Table 3, neural network sucessfully generates one-line solutions for simple problems, but faces difficulties with complex ones.

print (6 - 7)
print (2 + 22)
print (' Hi ' * 2)
print print (18 * 2)
if = abs ' - -))) < = 10 or or abs 200 - -) < = 10)
if = 1 mod - -))) = = = or ((200 + + = = =)
print = ' ' ' ' print ' ((' ' + + + +))

Table 3. Generated code examples

V. Conclusion

In this paper, we proposed a neural machine translation approach to a problem of source code generation for simple algorithmic tasks. We trained a LSTM-based encoder-decoder RNN to generate Python source code from natural language descriptions. Experiments, both on shorter and longer inputs, revealed that proposed architecture can generate executable code that can solve simple algorithmic problems. However, it faces significant difficulties while solving more complex problems, mostly due to small dataset.

Our future work includes investigation on a dataset extension and longer sentence translation problem. That is necessary for complex algorithmic tasks solution generation. Also, we would like to extend proposed model to solve algorithmic problems in other programming languages.

References

[1] S. Gulwani, “Dimensions in program synthesis”. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming.*2010.

[2] K. Cho, B. Van Merrienboer, D. Bahdanau and Y. Bengio, “On the properties of neural machine translation: Encoder-Decoder approaches”. In *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation.*2014.

[3] I. Sutskever, O. Vinyals and Q. Le, “Sequence to sequence learning with neural networks”. In *Advances in Neural Information Processing Systems (NIPS 2014).*2014.

[4] K. Cho, B. Van Merrienboer, C. Gulchere, F. Bougares, H. Schwenk and Y. Bengio, “Learning phase representation using RNN encoder-decoder for statistical machine translation”. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014).*2014.

[5] D. Bahdanau, K. Cho and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate”. *arXiv:1409.0473.*2016.

[6] Y. Bengio, R. Ducharme, P. Vincent and C. Janvin (2003), “A neural probabilistic language model”. *J. Mach. Learn. Res., 1137-1155.* 2003.

[7] N. Kalchbrenner and P. Blunsom, “Recurrent continuous translation models”. In *EMNLP.*2013.

[8] 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:1510.07211.*2015.

[9] D.P. Kingma and J. Ba, “Adam: A Method for Stohastic Optimization”. *arXiv:1412.6980.*2017.

[10] K. Papineni, S. Roukos, T. Ward and W. J. Zhu, “BLEU: a method for automatic evaluation of machine translation”. In *ACL*2002.