CSE538 Project report
Text to code generation in Java

Tianao Wang
Computer Science Department
Stony Brook University
tianao.wang@stonybrook.edu

Abstract

Code generation is a famous topic in NLP. It's a challenging task, because model has to map natural language into environment variables, library API calls and user-defined methods. In this project, I use CONCODE, a new large dataset with 100,000 examples consisting of Java classes from online code repositories, and develop a new encoder-decoder architecture that models learn to generate code by text. I also try some pretrained models and fine-tune them to get a better performance.

1 Introduction

For most people, they are more familiar with natural language. Writing code requires a long period of training and practice. Therefore,for most people, programming is difficult and requires professional knowledge. Many people have good ideas, but don't know how to implement their ideas with programs. If natural language can be automatically processed into executable code, the threshold for programmers will be greatly reduced. It's a challenging task because the desired code can vary greatly depending on the functionality the class provides. Models must (a) have a deep understanding of NL description and map the NL to environment variables, library API calls and user-defined methods in the class, and (b) decide on the structure of the resulting code.

Many methods have been tried in this field. Some people use a model for analyzing source code via its Abstract Syntax Tree(AST) try to generate the code. There is also a method which train model with a network of LSTM and MLP[1]. A method for automatic code generation that includes obtaining a parse tree[2]. In addition, a transition-based neural semantic parser that maps natural language (NL) utterances into formal meaning representations (MRs)[3].

Some of them use UML model. However, the area of UML model generation has seen no significant advancements since 2012 and the simplistic NLP techniques have remained almost the same. So, we have to try more deep learning method. Moreover, these methods can not learn the details like variables and methods well since these didn't specifically handle them.

So, in this project, I use Bi-LSTM to encode some key information and decode them in the decoder part. Also, I will try to use an attention mechanism with multiple steps, by first attending to the NL, and then to the variables and methods, thus also learning to copy variables and methods. This two step attention helps the model to match words inthe NL with representations of the identifiers in the environment.

I will try different models to test. The baseline model is that only use Bi-LSTM to encode the input sequence. Next, I will encoder more information to see whether it can get better performance. Also, I will try to add more layers to change the architecture to improve the model. In addition, I will use pretrained models to do the test and fine-tune these models.

In the original project, it provides some evaluate code. It will use Exact Match score and BLEU score to evaluate the performance of models. The dataset I use is called CONCODE which is a new dataset comprising over 100,000 (class environment, NL, code) tuples by gathering Java files containing method documentation from public Github repositories. I compare the models I talked above and show the reuslt.

Here are the main outcomes of this project.

1. I reproduce the baseline model for the task of text to code generation

2. I modify the architecture of baseline model and try pretrained model to get a better performance

3. The evaluation shows the improved model is

better than baseline model in EM score and BLEU score

4. Based on my work, I conclude that investigating variable name and types, method information might be promising avenues for future work.

2 Tasks definition

In this project, the input is a natural language query with some tag to store extra informations such as variable names, variable types, method names, method returns and so on. The output is a automatically generated code. The key challenges is to let models learn these extra informations.

The standard solution is to use LSTM model to encode and decode the information to help the model learn how to generate the code[1]. In the project I choose, they use Bi-LSTM to build an encoder-decoder model that encodes the NL together with representations based on subword units for environment identifiers and decodes the resulting code using an attention mechanism with multiple steps[4].

2.1 Baseline Model(s)

The figure of the baseline model is shown in figure1. In the encoder part, it first encodes input sequences as embeddings and use a Bi-LSTM to encode them. In the decoder part, it decodes these contextual representations to produce a sequence of valid production rules that derive syntactically valid source code. The decoder also uses a two-step attention mechanism to match words in the NL with environment components, and then uses a supervised copy mechanism to incorporate environment elements in the resulting code.

2.2 The Issues

In the baseline model, it only encode the sequence information. So, it doesn't performance well in the test part. Also, in the decoder layer, the input size and output size between layers have a large gap. I think it may lose some information in the decode part.

3 My Approach

I try to modify the baseline model and encode more information in the encode part. Also, I try some pretrained model and fine-tune them to get a improvement.

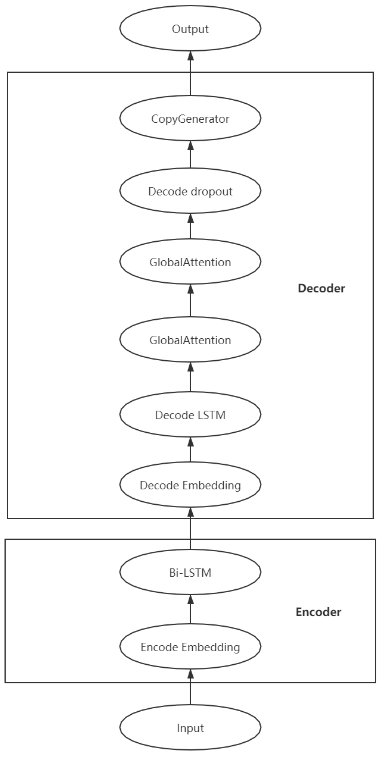


Figure 1: baseline model

3.1 Idea 1: Modify baseline model

The input size and output size between layers in baseline model have a large gap. In the deep learning field, I don't think it is a reasonable design. So, I try to add more layers to smooth the model. Moreover, in our assignment, I find that attention mechanism is easy to overfitting. To solve this problem, I add more dropout layers to avoid overfitting. The result shows that my modification has a little improvement.

3.2 Idea 2: Encode more information

In the paper, the author introduces a method that encodes variables and methods in the encode part. In the evaluation part, I find that this method can improve at least one time from the baseline model.

3.3 Idea 3:Use pre-trained models

Due to the limitation of GPU and training time, I also think that I can use pre-trained model to get a relatively good performance. In the evaluation part, I find that using pre-trained model can get the best performance in my project.

3.4 Implementation Details

Here I will talk about the details of implementation. In the encode part, the encoder creates

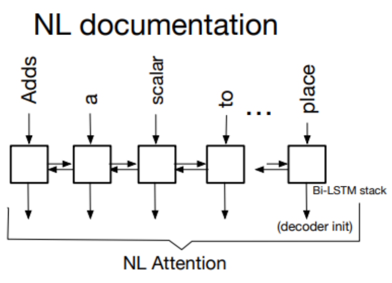


Figure 2: NL encoder

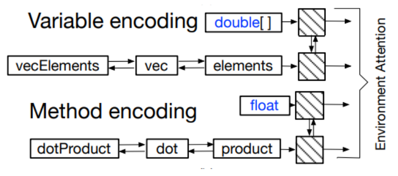


Figure 3: Variable and method encoder

contextual representations of the NL (figure2), the variables and the methods (figure3). Variable (method) names are split based on camel-casing and encoded using a BiLSTM. The variable (method) type and name are further contextualized using another BiLSTM.

The decoder first uses an n-layer LSTM. Next, the decoder attends to every token in the NL representation, using the current decoder state, to compute a set of attention weights, which are used to combine NL representation into an NL context vector. The context vector is used to attend over every type (return type) and variable (method) name in the environment, to produce attention weights that are used to combine the entire context into an environment context vector.

I use GPT-2 as a pretrained model which is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million web pages. For the fine-tune of pre-trained model, I follow the fine-tune tutorial on pytorch website and try different architecture to reach a good performance.

4 Evaluation

4.1 Dataset Details

I use a dataset called CONCODE which is on Github that contains environment information together with NL (Javadoc-style method comments) and code tuples. The data is divided into 100,000 train and 2,000 test. I use google colab to run my project. Due to the limitation of GPU memory and running time on colab, I reduce the dataset and

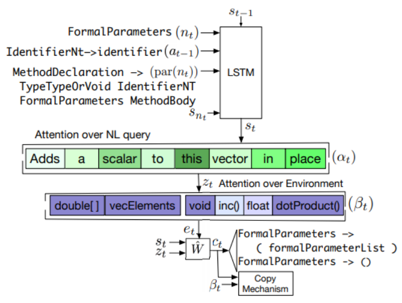


Figure 4: Decode model

limit the training time. I tried to get a relatively good performance in limited time.

4.2 Evaluation Measures

The project provides a script to evaluate predictions for this task, and report exact match and BLEU score(partical credits). In the original project, they use two NVIDIA P100 training for 22 hours. What I want to do is that getting a relatively good model in one or two hours.

4.3 Baselines

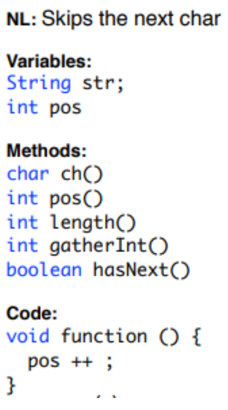
hyperparameters are shown below

- learning rate = 0.001
- enc layers = 2
- dec layers = 2
- batch size = 20
- src word vec size = 1024
- tgt word vec size = 512
- rnn size = 1024
- dropout rate = 0.5

I increase the batch size but colab is out of GPU memory. Due to the limitation of colab, I tried src word vec size in 512 and 1024, rnn size in 512 and 1024, tgt word vec size in 256 and 512. After comparing, I get the final hyperparameters.

4.4 Results

The result is shown below, it's not the best performance since the limitation of colab. If I can use better GPU for more time, the data maybe become better.



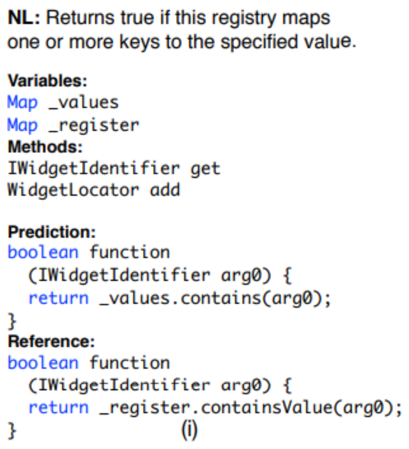
```
NL: Skips the next char

Variables:
String str;
int pos

Methods:
char ch()
int pos()
int length()
int gatherInt()
boolean hasNext()

Code:
void function () {
  pos ++ ;
}
```

Figure 5: correct example



```
NL: Returns true if this registry maps
one or more keys to the specified value.

Variables:
Map _values
Map _register
Methods:
IWidgetIdentifier get
WidgetLocator add

Prediction:
boolean function
  (IWidgetIdentifier arg0) {
  return _values.contains(arg0);
}
Reference:
boolean function
  (IWidgetIdentifier arg0) {
  return _register.containsValue(arg0);
}
```

Figure 6: wrong example

Model	EM	BLEU
baseline model	2.5	17
adv model	2.8	17.14
model with more information	4.65	15.76
gpt2	12.4	15.86
gpt2-java-adapted	16.2	17.5

The baseline model doesn't get a good EM score. The advanced model improve a little bit. The model with more variable and method information does improve the performance. Pre-trained models called gpt2-java-adapted get the best performance.

At least we can see, according to the result, more smooth layers and dropout layers can improve the model. More variable and method information can also improve the model. The pre-trained model is the easiest way to get a good performance in limited time and resources.

4.5 Analysis

After I see the predict files which generated by the models, I find that the model works well in some easy functions. However, it may fail when there are similar function names like figure6. If it has a better encoder representation, maybe can solve this problem.

In general, the exact match score isn't too high for all models. So, there are still great prospects in this field.

4.6 Code

Google Drive Link

5 Conclusions

I have learnt a lot from this project and CSE538 course in this semester. I know that we can encode sequences with more important information and use them in the decoder part. In addition, I learn that we can use attention mechanism to extract key message which we want our models to learn about. What's more, using pre-trained model is a good trick to get a relatively good result when we only have limited resources and time.

6 References

[1] R. Tiwang, T. Oladunni and W. Xu, ”A Deep Learning Model for Source Code Generation,” 2019 SoutheastCon, Huntsville, AL, USA, 2019, pp. 1-7, doi: 10.1109/SoutheastCon42311.2019.9020360.

[2]Yu J B , Fleming A M . Automatic code generation via natural language processing: US 2010.

[3] Yin P , Neubig G . TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation[J]. 2018.

[4] Iyer, Srinivasan, Konstas, Ioannis, Cheung, Alvin, Zettlemoyer, Luke,Mapping Language to Code in Programmatic Context, doi: 10.18653/v1/D18-1192, 2018.