CS230

Automatic Code Generation Project Proposal

Lizi Ottens
Department of Computer Science
Stanford University
lottens@stanford.edu

Luis Perez
Department of Computer Science
Stanford University
luis0@stanford.edu

Sudharshan Viswanathan
Department of Computer Science
Stanford University
viswans@stanford.edu

Abstract

We propose developing a machine learning system for code generation.

1 Problem Description

Automating even small parts of software development is an interesting research area [1] of machine learning that can save time for countless software engineers and resources for technology companies to invest into solving other more challenging problems. Automating code generation can take on many forms, from auto-completing lines of source code to generating lines of source code from comments, generating source code from UI images, or generating unit tests from source code. In this project, we aim to take incomplete lines of code as input and generate code to complete it. We hope that a byproduct of our endeavor will be the ability to understand and visualize [2] how deep learning architectures perceive and process code.

Subsequent work would be to ensure that the generated function code can compile, applying reinforcement learning to incorporate this signal in training. We also hope to extend our work to applications such as generating unit tests given a source function, or generating code given documentation.

2 Challenges

A primary challenge in code generation is that it's still an active area of research, with many possible solutions and ongoing investigation [3]. State of the art solutions have not yet come close to automating basic tasks software engineers perform on a daily basis.

Other challenges include restricted language domains, complexity in evaluation results, etc. Specifically, while pre-trained models are trained on free-form language data, programming languages often utilize non-natural variable names, function names, and syntax with more structure [3]. Work in this area has focused on creating more structured models that take advantage of specific architectures [4]. Other approaches involve restricting the output of the model to a CFG or DSL [5]. A code generation model's output must adhere to a very specific form in order to be syntactically correct.

Compilers/interpreters could be leveraged to provide a better form of feedback and hence can be used to create labelled data, but it is left to be seen how effectively we can leverage this feedback to aid the learning process.

3 Datasets

We propose two approaches. The first would involve scraping open source code from GitHub. The benefits of this approach is that it would provide us with ample data to utilize, from a diverse set of open source projects. The drawback is that the data would require significant pre-processing, cleaning, and labeling.

As an alternative initial approach, we will begin with the CodeSearchNet dataset [6]. The dataset consists of 2 million (comment, code) pairs from open source libraries, ranging in languages from Python, Javascript, and PHP. Median code-length consists of 60-100 text tokens, with 95% code-length of up to 350 tokens. Median documentation length consists of 10 text tokens. There also exist other curated datasets such as those used in [7] and [8].

4 Learning Method

Developments in the area of NLP such as GPT-2 [9] and BERT [10] have significantly improved the "quality" of text generation systems. State of the art results from these models can be conditioned on a given context, encouraging the models to generate text from interesting distributions. This yields itself as a good starting point for code generation since the act of writing/reading code can be viewed as an extension of natural language. Though, such an attempt also has the additional bar of ensuring that generated code compiles, or runs on an interpreter and also produces intended outputs.

In our project, we aim to start from pre-trained language models, and utilize that to generate code by conditioning the model and fine-tuning it to our specific task. The input to our model will be a text sequence corresponding to an incomplete function call, and the output of the model will be the completed function.

4.1 Code Generation

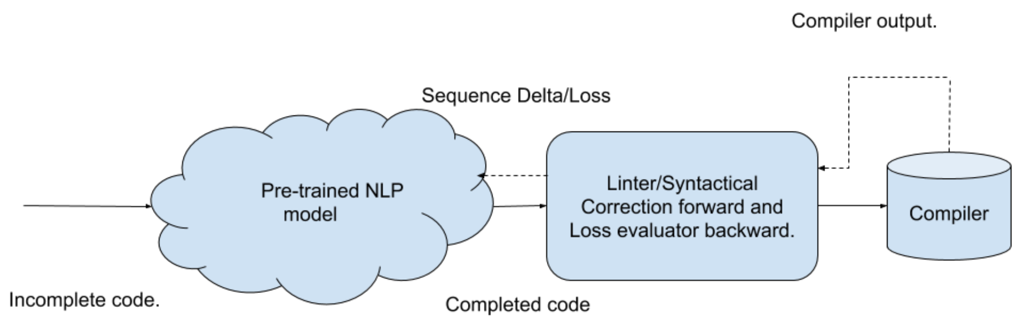

Figure 1: Overview of Proposed Architecture for Automatic Code Generation with Compiler Feedback.

We start with a pre-trained NLP model that would generate more text that fits the context of the input. In this case, the input text is going to be code and the output would be the completed code. We would have to ensure that the generated code fits the syntax of the language and hence feed it through another network. Furthermore, the second block becomes instrumental in interpreting the compiler output and output a delta that could then be used as a loss function for the NLP model to re-evaluate its parameters.

4.2 Loss Computation / Adapting Compiler Feedback for Gradient Descent

Typical software engineers use the compiler output in order to decide their next steps. And those steps would be incremental in nature. The delta change in code would serve as a very good loss function for the NLP model to train on, while the forward path of the same network could double as a linter and perform simple syntactic correction. A good way to arrive at this model would be to train different Seq2Seq models (RNN/LSTM/GRU) on known good code and randomly modify, record the delta and codify the compiler output. This would enable us to flip the model and use it to feed compiler output back into the model and generate the program delta which would serve as the loss for the first network.

5 Evaluation

For evaluation, we plan to hold out a fraction of the input dataset and then attempt code generation/completion on the hold out set and evaluate whether the output generated compiles in the first place, and if it does how far it deviates from the actual code that we started from. We will define a metric, compilation success rate, and report this on our generated programs. There are also other NLP-centric metrics such as the BLEU score [11] which we can use to measure how well our model outputs match the expected function.

Qualitatively, we also hope to be able to tool the networks with visualization [2] and extract information about what the different units encode when it comes to parsing and generating code.

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[2] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[3] Yasir Hussain, Zhiqiu Huang, Senzhang Wang, and Yu Zhou. Codegru: Context-aware deep learning with gated recurrent unit for source code modeling. *CoRR*, abs/1903.00884, 2019.

[4] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation, 2019.

[5] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. A grammar-based structural CNN decoder for code generation. *CoRR*, abs/1811.06837, 2018.

[6] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019.

[7] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W. Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *CoRR*, abs/1603.06744, 2016.

[8] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, 2015.

[9] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *None*, 2019.

[10] Kenton Lee Kristina Toutanova Jacob Devlin, Ming-Wei Chang. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[11] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. Retrieval-based neural code generation. *CoRR*, abs/1808.10025, 2018.