Analyzing Software using Deep Learning

– Project Description, Summer Semester 2018 –

Prof. Dr. Michael Pradel

May 29, 2018

Goal

The goal of this project is to design, implement, and evaluate a neural network-based code completion approach. Given a partial program, the approach suggests source code to be inserted at a specified location. Such a code completion technique could be used, e.g., as part of an integrated development environment (IDE). The approach consists of two phases. In the first phase, the approach learns how to complete code from a corpus of training programs. In the second phase, the approach is given a partial program and predicts the missing code based on the learned knowledge about programs. The project focuses on JavaScript code, but the approach could easily be applied to other languages.

Framework and Training Data

The project will be implemented within a given framework and should use a given set of training programs. The framework contains a minimal solution that serves as a baseline. Students should improve this solution, guided by the overall goal to improve the accuracy of the prediction of code:

$$accuracy = \frac{nb.\ of\ correct\ predictions}{total\ nb.\ of\ queries}$$

The framework is implemented in Python based on the Tensorflow and TFLearn libraries. We have tested everything with the following versions:

- Python 3.5.2
- Tensorflow 1.1 (with CPU support only)
- TFLearn 0.3.1

Students are strongly encouraged to use the same or compatible versions. During grading, any problems that result from incompatible versions are considered as an incorrect implementation.

The training data are 1,000 JavaScript files. For each file, we provide the source code (in a **.js* file), the sequence of tokens of the source code (in a **_tokens.json* file), and the abstract syntax tree of the source code (in a **_ast.json* file). During the design, implementation, and evaluation of the project, we recommend to use most of the files for training and to keep a small subset, e.g., 200 files, of the data for validating how accurate the approach is. We provide the training data in two folders that contain 800 and 200 JavaScript files, respectively. When submitting the final solution, students should train the network with all 1,000 training programs. The approach is not allowed to use any other training data beyond the provided files or any other information on how to predict code, e.g., manually written code completion rules.

The provided framework consists of two main files:

1. The *runner.py* file is the main entry point. It can be executed as a Python program, which loads the training data and validation data, trains the network with the training data, and then queries the trained network with the validation data. Finally, the file outputs the accuracy of the prediction. Students should not modify the *runner.py* file, except for explicitly marked regions of the file, and must ensure that the implementation can be executed with this file.

2. The *code_completion_baseline.py* file implements a simple feedforward neural network architecture used by *runner.py*. Students should replace this file with their own solution. We recommend the following workflow: First, understand and execute the baseline solution. Then, copy the *code_completion_baseline.py* file into, say, *code_completion_improved.py* and improve it. During the project, each student is likely to experiment with multiple solutions. Each of these can be stored in a separate file. For the final submission, clearly mark the best found solution by naming the file *code_completion_final.py*.

The framework represents a program as a sequence of tokens. Each token has two properties: type and value. For an example, see the **_tokens.json* file of any of the JavaScript files in the training data. The baseline implementation represents each token as a one-hot vector, i.e., a vector whose length is the number of unique tokens and where each element is zero except for the element that corresponds to the token represented by the vector. To train the network, the implementation creates pairs of tokens (t, t'), where t is the token just before t' in the training data. In total, the baseline implementation extracts about 1.7 million such pairs. These pairs are provided as training examples to a simple feedforward network, i.e., the network is trained to predict the next token given the previous token. To query the network with a given program that has a "hole" of to-be-predicted code, the implementation passes the last known token before the hole to the network and returns the token that the network predicts as the next token.

The code completion class used by *runner.py* must provide three APIs:

- The `train` method obtains a list of token sequences and a path where to store the parameters of the trained network. The method should train the network with the provided token sequences and then store the results. Storing the trained network is useful to avoid re-training the network unnecessarily and to enable students to submit an already trained network. The method does not have any return value.

- The `load` method also obtains a list of token sequences and a path where the parameters of an already stored network are stored. Instead of re-training the network, the method simply loads the previous state. The method does not have any return value.

- The `query` method obtains two sequences of tokens, which represent the code around the code to predict. The first parameter, `prefix`, contains all tokens before the missing part; the second parameter, `suffix`, contains all tokens after the missing part. The method returns the sequence of tokens predicted by the network.

These APIs are implemented in the baseline implementation, and the solution submitted by students also needs to implement them, to enable us to run the implementation using *runner.py*.

Tasks

The overall goal is to improve the accuracy of the predicted code completions. We suggest several steps to achieve this goal:

1. Install all requirements and get the framework to run. Executing the baseline implementation by simply running the unmodified *runner.py* should run for at most a few minutes and then give an accuracy between 18% and 25%. The number may vary because queries are constructed by randomly removing parts of the given code.

2. Create a copy of *code_completion_baseline.py* to implement your modifications. As an alternative, students may also implement their own code completion class from scratch. The only requirement is to provide the three APIs called by *runner.py*.

3. Implement and experiment with several suggested steps that may improve the accuracy of predictions (in no particular order):

    - Try to use more hidden layers.
    - Try other neural network architectures, e.g., architectures designed for sequences of inputs and/or outputs.

- The baseline implementation uses a one-hot encoding. Try to vary the representation of tokens.
- Try to vary the hyperparameters of the network, e.g., the size of hidden layers, the number of epochs, and the batch size.
- The baseline implementation predicts sequences of exactly one token, i.e., it cannot predict the correct code completion if more than one token is missing. Instead, try to predict more than one token. The missing parts of the program may consist of up to `max_hole_size` tokens. For experimenting with different sizes of missing parts, students can configure this parameter in *runner.py*.
- To predict tokens, the baseline implementation considers only the code before the missing tokens, i.e., the prefix. Instead, try to consider both the prefix and the suffix of the missing code.

The above list is not exhaustive. Any other changes that lead to improved accuracy are welcome, and students are encouraged to come up with their own ideas on how to improve the approach.

4. Optional tasks for students that have significantly improved the accuracy of predictions and would like to obtain extra points: (The tasks are roughly sorted by increasing difficulty.)

   - The baseline implementation combines the two properties of tokens, type and value, into a single string. Instead, try to predict both properties individually, e.g., by training two neural networks.
   - In its default configuration, the framework abstracts the values of some tokens to limit the total number of unique tokens. For example, the framework replaces all identifier names with "ID" and all string literals with "STR". Instead, try to also predict these properties. To configure the framework to not ignore these properties, modify the `simplify_tokens` flag in *runner.py*.
   - The current framework represents programs as sequences of tokens. Instead, try to represent programs as ASTs. The training data contains an AST for each JavaScript file.

Deliverables and Grading

Grading will be based on three deliverables. Each deliverable has equal weight for the final grade of the project.

- *Project report.* Each student submits a report of 2-3 pages that briefly describes the implemented approach and the empirical results. Specifically, the report should contain:
  - A description of the neural network architecture, along with a discussion of why this architecture has been implemented (and not another one).
  - A description of experimental results obtained with the approach, in particular, the accuracy of the prediction. This part of the report should clearly describe the experimental setup (E.g., how many training examples and how many validation examples were used?). Moreover, the report should describe results obtained with alternative approaches (E.g., how much worse does the accuracy get with fewer hidden layers?).

  A good report typically uses examples for illustration. For example, the report should contain examples of successful predictions that show the advantage of a particular design decision, but also examples of incorrect predictions to illustrate the limitations of the approach.

- *Implementation.* Each student submits the implementation, i.e., the *code_completion_final.py* file and any other helper files required by it. The implementation must be executable with the *runner.py* provided in the original framework.

- *Trained network.* Each student submits the saved state of the trained network. The path where to save the state can be configured using the `model_file` variable of *runner.py*.

For grading purpose, we will execute the implementation with the trained network provided by the student, but on other validation data than what is provided to students. We may also execute the submitted solutions with other training data.

The deadline for submitting all three deliverables is July 9, 2018. This deadline is firm. Details on how to upload the deliverables will be provided.