Code Prediction

Remy Bubulka, Coleman Gibson, Kieran Groble

14 November 2017

Introduction

We are using neural networks to predict the behavior of code segments. We are working with a minimal subset of Scheme for the code we are analyzing, as it has a simple and consistent structure. Given a program, we would like to predict whether or not it will error. For programs which do run, we would also like to predict what value is returned.

Motivation

Humans write imperfect code, and a substantial amount of effort has been put into creating tooling for developers to handle this fact of programming. However, these tools have significant room for improvement. The Economic Impacts of Inadequate Infrastructure for Software Testing Report, written in May 2002, suggests that nearly $60 billion dollars are lost every year to software bugs in the economy of the United States, and that there is room for over $20 billion in improvement. A large body of research has been developed in removing this deficit. Facebook has developed the Infer tool for static analysis, NASA has worked on creating Java PathFinder, and tools like Valgrind and JUnit are frequently used to track down bugs.

We believe that these tools have the potential to be improved by deep learning. In particular, although introducing neural networks would make these tools unsound, we believe that deep learning could be used to remove false negatives and false positives in static analysis tools. This would remove noise for developers, allowing them to focus more of their time on bugs that truly exist.

Deep learning could also be used to improve the speed of an analysis. Some analyses of large code bases can take well over a day, using a common approach of iterating over the syntax tree until a fix-point is reached in the analysis. A single pass by a neural network would require no complex logic and would not necessarily need to be iterated, allowing for reduced time to bug discovery. Our tool represents a step in the direction of incorporating deep learning into the world of code analysis.

Data

We generate all of our data. Our programs contain the arithmetic operators addition, subtraction, multiplication and division; literal values -2, -1, 0, 1, and 2; and lambda functions. We train on short, "well-behaved" programs – those in which all sub-calls return numerical values (rather than closures).

Below is a Backus-Naur Form grammar describing valid programs in our language:

⟨*exp*⟩ ::= ⟨*lambda-exp*⟩
| ⟨*app-exp*⟩
| ⟨*var-exp*⟩
| ⟨*lit-exp*⟩

⟨*lambda-exp*⟩ ::= (λ (⟨*formal*⟩) ⟨*exp*⟩)

⟨*formal*⟩ ::= x | y

⟨*app-exp*⟩ ::= (⟨*lambda-exp*⟩ ⟨*exp*⟩)
| (⟨*prim-proc*⟩ ⟨*exp*⟩ ⟨*exp*⟩)

⟨*prim-proc*⟩ ::= + | - | * | /

⟨*var-exp*⟩ ::= x | y

⟨*lit-exp*⟩ ::= -2 | -1 | 0 | 1 | 2

Data is pre-processed via tokenization – a single program is represented as an $m * n$ matrix, where m is the length of the longest program in the dataset and n is the number of tokens. A column is a one-hot encoded representation of the token at that position. There are a total of 14 tokens – 0, 1, 2, λ, +, -, /, *, (,), x, y, a token for space, and a null token to pad short programs.

For error detection, we used dataset of 235881 programs with depth of at most three nested procedure applications. For output prediction, we used a dataset of 111011 programs with depth of at most two. In both cases, 20% of the data were reserved for test data, and within the remaining 80% there was a 70% – 30% split between training and validation d235881ata.

Network

The best set of hyperparameters we have discovered so far form the following network: A 256-node LSTM input layer, a dropout layer with a factor of 0.2, and an output layer (2-node with softmax and cross-entropy for error classification, 1-node with MSE for output prediction). We use a batch size of 800 and enable early stopping with a tolerance of 6. Adding dense layers after the LSTM layer negatively impacted prediction accuracy; adding additional LSTM layers had no impact on accuracy but slowed down training.

Results

Error Classification

We have achieved a test accuracy of 99.19% with this network. Table 1 provides some selected test cases, whether or not they actually error, and how likely our network thought they were to error.

Table 1: Results for Tests on Error Evaluation

Code Snippet	Should Error	Confidence of Error
(/ 1 1)	No	0.0001%
(/ 2 0)	Yes	99.99%
(/ 0 1)	No	0.0001%
(/ 1 (- 1 1))	Yes	34.47%
(+ 1 0)	No	0.000099%
(/ 1 ((λ (x) 1) 0))	No	0.0001%
(/ 1 ((λ (x) 0) 1))	Yes	98.82%
(+ 0 (* 0 (/ 0 1)))	No	0.0001%
((λ (x) (/ (- 0 0) (- 0 1))) 0)	No	83.34%

Output Prediction

Our output prediction network achieves a Mean Squared Error of 0.0096, down from a variance of 0.425; and a Mean Absolute Error of 0.027, down from a mean absolute distance from the median of the dataset of 1.65.

Table 2 contains some selected test cases, expected output, and predicted output. Predictably, our network tends to perform better on simpler programs with fewer nested procedure applications. The exception to this rule is programs consisting of a single literal expression, which our network almost never encountered without additional context during training.

Table 2: Predicted Output for Valid Code Samples

Code Snippet	Expected Result	Predicted Output
(+ 2 1)	3	3.02568173
(+ 1 2)	3	3.05547428
((λ (x) (+ x 1)) 1)	2	1.39392674
(/ 1 2)	0.5	0.44873938
(* 2 (* 2 2))	8	9.61662197
(* 2 2)	4	4.25841475
(- 1 2)	-1	-0.90237552
(* 1 (+ 1 1))	2	1.83598292
(* 2 (+ 1 1))	4	3.6835146
1	1	0.72898144
2	2	-0.44422317

Table 3 contains intentionally malformed test cases and the output the network predicted. Some of these cases are valid Python code (but syntactically invalid in our grammar); others contain unresolved variable references.

Table 3: Predicted Outputs for Bad Code Samples

Code Snippet	Predicted Output
(2 + 1)	2.87946272
2 + 1	2.93325639
1 + 2	2.00386
(2 * (1 + 1))	2.85971975
(+ x y)	-0.4209832
x	-0.88294739
y	-0.89534634