Application of Reinforcement Learning in Code Repair

Author: Young Kim

Korea International School, Pangyo, Korea

Abstract

Novice coders very frequently come across compile errors and learning to code without syntactical errors or debugging based on the given error messages can be a challenging task. In this study, I created a machine learning model that collects compile error messages of codes created by novice students, learns them using an LSTM recurrent neural network model, and repairs them correctly. Training data were collected from an online judge system, in which functioning codes were purposely and systematically modified to become erroneous. After the tokenization preprocessing step, I used LSTM to repair the erroneous parts of the given code. It was confirmed that the machine learning model created in this study solved 43% of the errors generated by novice programmers. Specifically, relatively simple errors including missing semicolons or unmatched brackets could be fixed with high accuracies of 78% and 73%, respectively. The results of this study highlight those errors of simple syntax are easy to fix with artificial intelligence, whereas those that depend more on context and user intention are harder to repair.

Key Words: Code repair, Long Short-Term Memory (LSTM), Recurrent Neural Networks (RNN), Tokenization

Introduction

1.1 Motivation

During my time at the UPenn ESAP computer science summer program, I was exposed to students of all types of programming backgrounds in which I realized that a large portion of students very often makes small mistakes in the form of basic syntax or structure of their code. When students are given solely a compile error message, I noticed that it's very difficult to diagnose the specific error of one's code, especially if you're new to programming. I further realized that in most cases students aren't the most challenged when forming the logic behind a program, but rather its implementation. Reflecting upon when I first started programming, even the smallest syntactic error such as forgetting to add a semicolon after each line posed a barrier for me to grow as a programmer. At the same time, I was learning about natural language processing using LSTM in machine learning, and I thought that it might be possible to create a program that automatically corrects these compilation errors by learning the mistakes commonly made by novice programmers. This tool, rather than a typical program editing tool, will make it much easier and more convenient for novice students to find the location of errors and how to fix them. As a result, I decided to create a machine learning model that helps novice coders automatically correct the many syntax errors often encountered by programmers. With this tool, novice coders will find programming more welcoming and easier to learn by themselves. During the time of development of my machine learning model, I explored the various types of compile errors that programmers face and learned about the limits of machine learning when fixing these errors.

1.2 Problem

According to a case study by Google, the median resolution time of build errors was 5 and 12 minutes for C++ and Java, respectively, and can vary by an order of magnitude across error types. An even larger amount of time is dedicated to debugging episodes for novice coders. According to the George Mason University in figure 1, debugging episodes occupy a large portion of most programming sessions, displaying prevalent inefficient use of time. Furthermore, the most frequent types of errors were the use of undeclared variables and misdeclaration of variables – errors that typically require minimal lines of code to repair yet has been shown to be very difficult to diagnose as seen in figure 2.
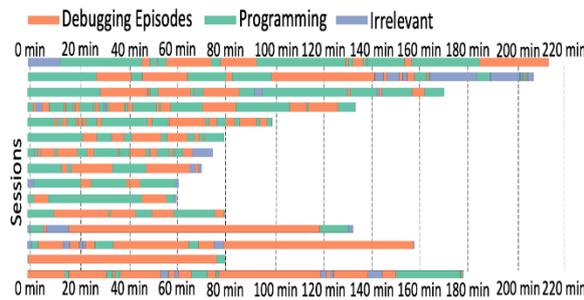
[Figure 1]. Time distribution in 15 programming sessions.
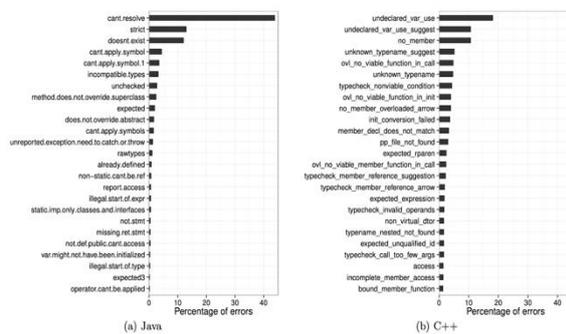
[Figure 2]. Reason for build errors in Java and C++

2. Background Knowledge

2.1 Reinforcement Learning

There are three (3) main machine learning training methods: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, input values and their respective labels are already given, meaning that the model's weights can be adjusted to best fit the input data and their respective outcomes. On the contrary, in unsupervised learning, data is unlabeled meaning that other methods including pattern detection, clustering, and PCA/anomaly detection must be used. Reinforcement learning (RL) is quite different from the other learning methods. In RL, the machine learning agent is faced with a game-like environment, where complex and uncertain decisions must be made. As seen in figure 1, after each iteration, the artificial intelligence either receives a "reward" or a "punishment" for the agent's decisions.
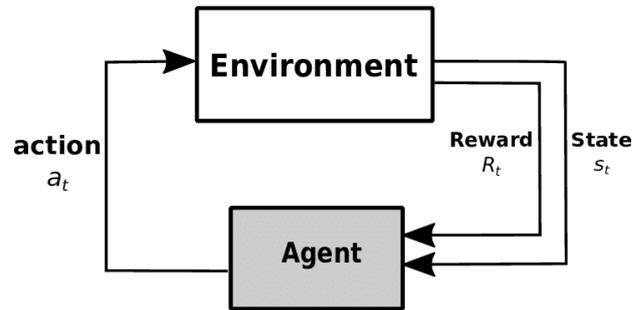
[Figure 3]. Reinforcement Learning Model

The only information that the programmer provides to the machine learning model is the reward policy. From here, the artificial intelligence will run a large series of decisions with a plethora of failures and occasional successes to learn sophisticated tactics and approaches. Reinforcement learning isn't seen exclusively in computer science. As seen in figure 2, the fundamental idea of dog training aligns with that of reinforcement learning. No dog instinctively knows that running to get a stick is the correct action, but once the trainer provides a reward once the task is succeeded, the dog learns that running to get the stick will result in a reward.

[Figure 4]. Reinforcement learning in dog training

2.2 Recurrent Neural Networks (RNN)

As you read this sentence, your understanding of each word is based on the understanding of the words that come before it. Naturally, you don't simply perceive a word by its individual meaning, but rather the role it plays within a sentence. However, this task is very difficult for traditional neural networks – the base of deep learning algorithms – as they don't have access to previous points of the data set. This is where recurrent neural networks (RNN) serve a huge role. RNN is a specific type of network that permits the persistence of information.

However, there is one downside to RNNs. RNNs are extremely good at analyzing recent information to perform a certain task. This means that when correcting code, as long as

the error lies within reach of the RNN model's code analysis, the error can be fixed. For example, given the example below, RNNs can isolate the error to line 7 where "a = b" should be "a == b."

[Figure 5]. Case where RNN is effective
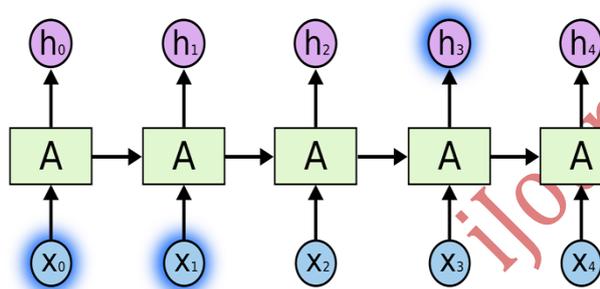
[Figure 6]. Portrayal of short-term dependencies in RNNs

However, as soon as the compile error grows more complex with a wider "radius of error", RNNs fail to gather all the context required to fix the error. For example, for the example below, it becomes very difficult for RNNs to fix the compile error given only recent information. RNNs would think that the error lies within line 9 of the code, whereas the actual error originates starting from line 6. Syntactic errors regarding variable types, therefore, become very difficult to diagnose and fix, as the sources of error lie on several different lines.

[Figure 6]. Case where RNN is ineffective

2.3 Long Short-Term Memory Networks (LSTM)

Fortunately, there exists a specific type of RNN that is capable of learning long-term dependencies: Long Short-Term Memory networks (LSTM). LSTMs are specifically designed to store information for long periods of time. Similar to traditional RNNs, LSTMs have a repeating chain-like structure. However, unlike regular RNNs that only have a single neural network layer, LSTMs are divided into four: The forget gate layer, input gate layer, update layer, and output layer.
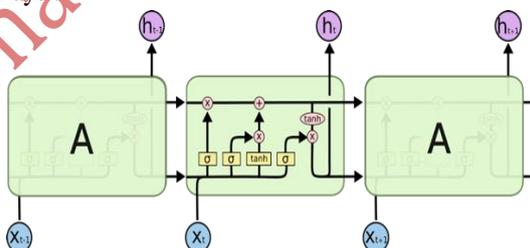
[Figure 7]. Repeating module in an LSTM containing four interacting layers

Due to this preservation of information unique to LSTM, it is often used in natural language processing and image/video classification where data is sequential. One example of the application of LSTMs is next-frame video predictions. One specific application of LSTM is in language translation. A certain word or phrase can have several possible meanings, and the appropriate definition within the sentence depends on the context around it. For example, the word "crane" can either mean the construction machine or the bird. Depending on what comes before and after the word, its definition can change drastically. The importance of context is important in identifying code errors as well. As seen in figure 6, it is impossible to diagnose the specific reason for the error just by looking at line 9 where the error was identified. Although there are some cases where errors can be isolated to
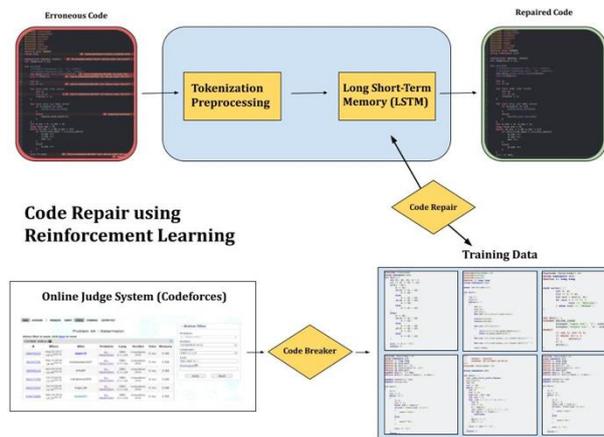
a single line in the code, in most cases the error accumulates starting from earlier points of the code.
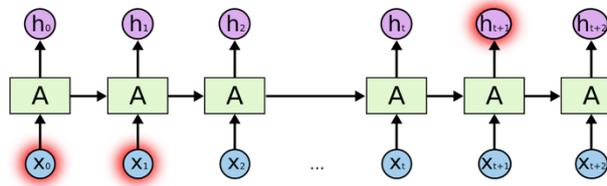
[Figure 8]. Portrayal of long-term dependencies in LSTMs

3. Research Methods

3.1 Overall System Structure

[Figure 9]. Code Repair System Structure Flow Chart

Figure 9 shows the main components of this project's structure. The user's erroneous code first goes through a tokenization preprocessing step, in which each word in the code is scanned and categorized. This information is then sent to the Long Short-Term Memory (LSTM) network. This network is trained using training data gathered from an online judge system. The LSTM model then corrects the input code and provides the user with a repaired version.

3.2 Data Preparation
3.2.1 Data Gathering Process

[Figure 10]. Codeforces Problem 4A – Watermelon

A large amount of erroneous and correct codes are needed to train the artificial intelligence model. In order to do this, I used an online judge system called CodeForces. Here, I chose two highly popular problems called "4A - Watermelon" and "231A – Team" to gather data. "Watermelon" was specifically chosen as the problem-solution required the use of an if statement. "Team" was specifically chosen as for and while loops were required in the problem solution.

As this study is targeted at code repair for C++, only submissions written in C++ were collected. In "Watermelon", there was a total of 268,598 total attempts shown in the submission history, with a total of 26,472 compilation error codes. "Team" had a total of 158,955 submissions and 5,322 compilation error codes.

3.2.2 Code Breaking

At first, I tried collecting erroneous code and planned on making a corrected version myself. However, this required manual correction of each code, which would be very time-consuming due to the magnitude of erroneous codes. Instead, I came up with the idea to intentionally break corrected codes using in a systematic way. Referring to previous works, I was able to learn that novice coders make 5 main types of errors: 1) Missing Semicolon 2) Unmatched Brackets 3) Misplaced Symbol/Operator 4) Calling of a function 5) Undefined Variable. From here, I made a "code Breaker" that creates errors for each type. 200 codes were made for each type, with a total of 1,000 codes included in the training data.

[Figure 11]. Example of bracket and semicolon code breaking

3.3 Tokenization

Accuracy is greatly reduced when code is given directly to a machine learning model in Natural Language Processing (NLP). Therefore, it is imperative to implement preprocessing in advance. Usually, the input code goes through tokenization, cleaning, and normalization processes. In our study, "Tokenization" refers to the process of dividing a given text

into several units called "tokens". When a program code is tokenized, each word is divided into three main categories: 1) keywords including "if" and "for" 2) variable names such as "a" or "b" 3) function names such as "printf" and "scanf." During this process, inputs such as gaps, indents, and variable names that do not give meaning and context to the code are ignored, leaving only grammatical elements. This process helps the LSTM comprehend the input code in subsequent steps. As the programming language doesn't require cleaning and normalization, effective tokenization is important for later processes to work faultlessly. The C++ coding language already has several premade tokenizers. In this study, we will be using a publicly available GitHub open-source tokenizer.


```
main.cpp ×
1    #include <iostream>
2    using namespace std;
3    int main() {
4        string s = "Hello World";
5        cout << s;
6    }
```

[Figure 12]. Example code for tokenization
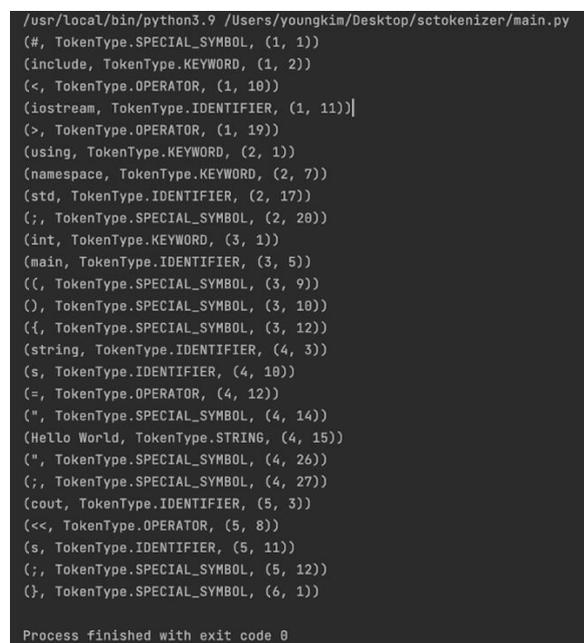

```
/usr/local/bin/python3.9 /Users/youngkim/Desktop/sctokenizer/main.py
(#, TokenType.SPECIAL_SYMBOL, (1, 1))
(include, TokenType.KEYWORD, (1, 2))
(<, TokenType.OPERATOR, (1, 10))
(iostream, TokenType.IDENTIFIER, (1, 11))
(>, TokenType.OPERATOR, (1, 19))
(using, TokenType.KEYWORD, (2, 1))
(namespace, TokenType.KEYWORD, (2, 7))
(std, TokenType.IDENTIFIER, (2, 17))
(;, TokenType.SPECIAL_SYMBOL, (2, 20))
(int, TokenType.KEYWORD, (3, 1))
(main, TokenType.IDENTIFIER, (3, 5))
((, TokenType.SPECIAL_SYMBOL, (3, 9))
(), TokenType.SPECIAL_SYMBOL, (3, 10))
({, TokenType.SPECIAL_SYMBOL, (3, 12))
(string, TokenType.IDENTIFIER, (4, 3))
(s, TokenType.IDENTIFIER, (4, 10))
(=, TokenType.OPERATOR, (4, 12))
(", TokenType.SPECIAL_SYMBOL, (4, 14))
(Hello World, TokenType.STRING, (4, 15))
(", TokenType.SPECIAL_SYMBOL, (4, 26))
(;, TokenType.SPECIAL_SYMBOL, (4, 27))
(cout, TokenType.IDENTIFIER, (5, 3))
(<<, TokenType.OPERATOR, (5, 8))
(s, TokenType.IDENTIFIER, (5, 11))
(;, TokenType.SPECIAL_SYMBOL, (5, 12))
(}, TokenType.SPECIAL_SYMBOL, (6, 1))

Process finished with exit code 0
```

[Figure 13]. Tokenization of code shown in Figure 9

As seen in figure 10, each character and string is scanned through in the tokenization preprocessing step. The aforementioned token categories (keywords, variable names, function names) are shown.

3.4 Machine Learning Model, LSTM Structure

A very specific type of machine learning model is required to accomplish code repair. Compile error messages are very important hints when a human attempts to fix compile errors. Therefore, our machine learning model first gathers the compile error message once an erroneous code is given. However, it is very common that the line indicated in the compile error message isn't the exact line the actual error originates from. Instead, when we scan ± 1 line from the indicated line number, the number of cases for which an error is diagnosed significantly increases. Therefore, our program will identify which line the error message diagnoses the error to be, then scans the previous, current, and subsequent lines for the cause of error.

To eliminate any insignificant portions of the input (such as variable names), tokenization was applied to the 3 aforementioned lines. The artificial intelligence model scans through each token so far, and makes a prediction of what the subsequent token will be. If the actual token from the input code is different from the prediction, the model infers that this is the cause of the error. Refer to figure 13 for a code example.

[Figure 14]. Sample Code

As seen in figure 13, the compile error message indicates that an error is generated on line 8 with "printf(%d,%d, a, b);." However, one can quickly notice that the actual cause of the error is from line 7 (previous line) where integer "b" was not properly declared. Our program is structured in a way that scanning starts from the declaration of integer b, and the two subsequent lines. Looking at the token sequence, our model will be able to discover that int, variable_name must be followed by a semicolon, which is not the case. Therefore, our machine learning code will add a semicolon and finish the repair job.

Furthermore, this model allows the repair of several lines of error if the aforementioned job is repeated iteratively. The first identified error will be fixed, and when the compiling yet again fails, the same job will be repeated to identify and repair the next error within the input code. However, when the error

is not fixed by its 5th iteration of repair, the code will end and conclude that the repair was unsuccessful.

4. Experiments

4.1 Experimental Procedure

In order to calculate the accuracy of our repair code, 40 correct answer codes were randomly selected among 410,000 actual error cases collected from Codeforces. Using the "code breaker" discussed in 3.2.2, five erroneous versions of each correct code was made, representing each type of error. In total, there were 1,000 codes in the final data set, which was further divided into training (70%), validation (10%), and test (20%) sets. Validation and test codes were randomly picked from the data set. The models were trained by minimizing the class negative likelihood loss with an open-source Torch implementation of a standard sequence-to-sequence model. All learning models were created using Python, and the process of training was conducted using a CentOS Linux Machine with an Intel Zeon E5-2680 V3 (@2.65GHz) Dual-core CPU, paired with 265GB of memory.

4.2 Experimental Results

Figure 14 shows a bar graph representing the accuracy of the output of our model to the actual correct code. The results show that our model was able to accurately repair 43% of errors made by novice coders. Specifically, syntax-related errors such as missing semicolons or unmatched brackets had high accuracies of 78% and 73% respectively. However, relatively more complex errors involving functions and variables had a low accuracy rate of 20%.
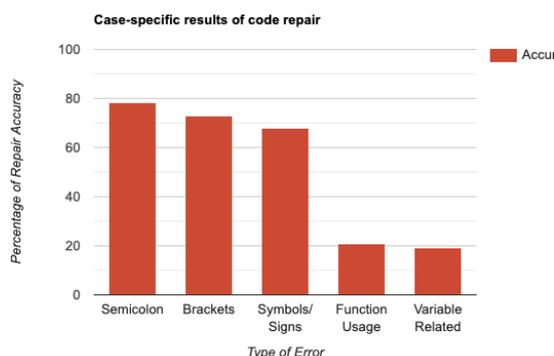
[Figure 15]. Error-specific results for accuracy of code repair

[Figure 16]. Demonstration of iterative code repair

One interesting result is that one can witness each error within a code being repaired iteratively. In the example shown in figure 15, there are two main errors: 1) missing semicolon in line four 2) missing "=" in line 7. As seen, each error is fixed chronologically.

[Figure 17]. Demonstration of iterative code repair

An example of a failed attempt of repair is shown above. The left image shows how the variable "num" is not declared. However, our model goes through a tokenization preprocessing phase which is why there is no information on any variable names. This makes it difficult to appropriately declare the "num" variable. As expected, the "num" variable isn't declared as shown in the right image, and the printf function is misused.

5. Conclusion

This study is significant mainly due to the growing number of coders and demand for software in the future. As we approach a technology-driven world, more and more individuals will need to learn how to code, and compile error messages in the status quo are misleading and often unhelpful. This repair tool will hopefully help novice coders genuinely enjoy writing code without the frustration of extended debugging episodes.

Through the experimental results, we can confirm that a functioning repair software was made that is capable of fixing erroneous code for novice students. However, it fails to effectively fix highly complicated codes as it either fails to diagnose the error within the input code or creates a solution that differs from the user's original intent for the code, which makes future areas of study.

6. References

[1] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilia, Robert Bowdidge, "Programmers' Build Errors: A Case Study (at Google)" ICSE 2014: Proceedings of the 36th International Conference on Software Engineering, 724-734, 2014

[2] Abdulaziz Alaboudi, Thomas LaToza, "An Exploratory Study of Debugging Episodes", Available From: https://www.researchgate.net/publication/351354680_An_Exploratory_Study_of_Debugging_Episodes (accessed Aug, 11, 2021)

[3] Kaelbling, Leslie P.; Littman, Michael L.; Moore, Andrew W. "Reinforcement Learning: A Survey". Journal of Artificial Intelligence Research. 4: 237-285, 1996

[4] Dupond, Samuel, "A thorough review on the current advance of neural network structures". Annual Reviews in Control. 14: 200–230, 2019

[5] Sepp Hochreiter; Jürgen Schmidhuber, "Long short-term memory". Neural Computation. 9 (8): 1735–1780, 1997

[6] Trim, Craig (Jan 23, 2013). "The Art of Tokenization". Developer Works. IBM. Available From: https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en (accessed June, 20, 2021)

[7] guillaumenkln, OpenNMT/Tokenizer, Available From: https://github.com/OpenNMT/Tokenizer (accessed July, 03, 2021)