The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)

Deep Reinforcement Learning for Syntactic Error Repair in Student Programs

Rahul Gupta, Aditya Kanade, Shirish Shevade
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, KA 560012, India
{rahulg, kanade, shirish}@iisc.ac.in

Abstract

Novice programmers often struggle with the formal syntax of programming languages. In the traditional classroom setting, they can make progress with the help of real time feedback from their instructors which is often impossible to get in the massive open online course (MOOC) setting. Syntactic error repair techniques have huge potential to assist them at scale. Towards this, we design a novel programming language correction framework amenable to reinforcement learning. The framework allows an agent to mimic human actions for text navigation and editing. We demonstrate that the agent can be trained through self-exploration directly from the raw input, that is, program text itself, without either supervision or any prior knowledge of the formal syntax of the programming language. We evaluate our technique on a publicly available dataset containing 6975 erroneous C programs with typographic errors, written by students during an introductory programming course. Our technique fixes 1699 (24.4%) programs completely and 1310 (18.8%) program partially, outperforming DeepFix, a state-of-the-art syntactic error repair technique, which uses a fully supervised neural machine translation approach.

Introduction

Programming oriented massive open online courses have gained huge popularity in the last few years. Due to the large number of student enrollments in these courses, providing real time personalized feedback is infeasible for instructors. Addressing this issue, a number of automated feedback generation techniques for programming assignments have been developed in recent years. However, most of the existing techniques focus on fixing semantic errors and work only for programs that compile successfully (Piech et al. 2009; Singh, Gulwani, and Solar-Lezama 2013; Kaleeswaran et al. 2016). These techniques require either SAT solving or test execution, both of which are not possible for the programs with syntactic errors.

Programmers rely on compilers to get feedback for syntactic errors. However, compiler error messages do not always localize the errors accurately and are often difficult to understand (Traver 2010). This makes syntactic errors very time consuming to fix and a major learning hurdle for students of introductory programming courses.

Figure 1 illustrates an erroneous programming submission implementing a tax calculation algorithm in an introductory course on C programming. It has two syntactic errors: (1) incorrect use of semicolon within the `scanf` function call in line 4 and (2) a missing closing brace at the end of line 12. The Clang compiler generates the following error message for it:

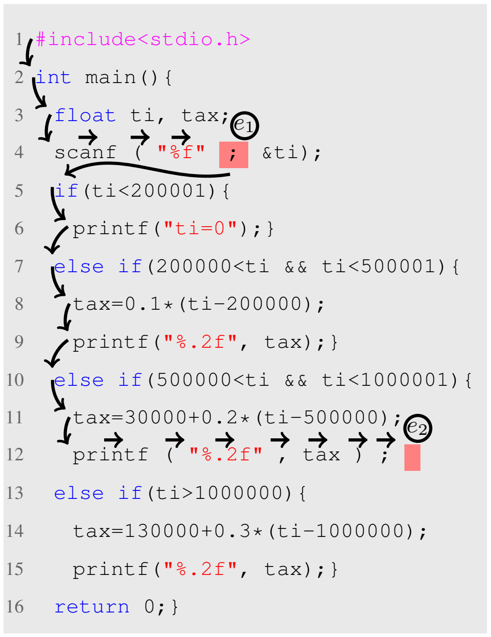
```
ex.c:4:12: error: expected ')'
scanf("%f"; &ti);
          ^

ex.c:4:7: note: to match this '('
scanf("%f"; &ti);
     ^

ex.c:4:17: error: extraneous ')' before ';'
scanf("%f"; &ti);
               ^

ex.c:13:2: error: expected expression
else if(ti>1000000){
^

ex.c:16:12: error: expected '}'
return 0;}
          ^

ex.c:2:11: note: to match this '{'
int main(){
          ^

4 errors generated.
```

Note that the above error message does not pinpoint either of the errors precisely[1]. While the error message may be sufficiently helpful for an expert programmer, a novice programmer may even end up introducing more errors in the program by following the suggested fixes.

Our aim in this work is to develop a syntactic error repair technique that can assist novice programmers by automatically correcting common syntactic errors in programs. When faced with an error, a programmer navigates through the program text to arrive at a possible location of error and then performs an edit to fix the error. Intuitively, this is like playing a game, in which an agent starts with an incorrect program as initial state and takes navigation and edit actions to reach the goal state (error-free program).

Deep reinforcement learning has enjoyed great success in

[1]The GCC compiler precisely pinpoints both the errors in this program but fails on many others. One such example is shown in (Gupta et al. 2017).



```
1  #include<stdio.h>
2  int main(){
3    float ti, tax;
4    scanf ( "%f" ; &ti);
5    if(ti<200001){
6      printf("ti=0");}
7    else if(200000<ti && ti<500001){
8      tax=0.1*(ti-200000);
9      printf("%.2f", tax);}
10   else if(500000<ti && ti<1000001){
11     tax=30000+0.2*(ti-500000);
12     printf ( "%.2f" , tax ) ;
13   else if(ti>1000000){
14     tax=130000+0.3*(ti-1000000);
15     printf("%.2f", tax);}
16   return 0;}
```

Figure 1: An erroneous program and the sequence of actions taken by a trained agent to fix it: The error locations are highlighted in the red color. The arrows show how the agent navigates over the program text. The edit actions are marked by e_1 and e_2.

training agents to play visual and text-based games at expert levels (Mnih et al. 2015; Narasimhan, Kulkarni, and Barzilay 2015; Wu and Tian 2016). Inspired by this, we approach this problem through deep reinforcement learning. We propose a novel programming language correction framework, in which an agent can mimic these actions. While the agent can access and modify the program text, the compiler which checks syntactic validity of the program text is a black-box for the agent. As noted above, compilers usually do not pinpoint error locations precisely. Hence, we do not rely on a compiler to aid in error localization or correction. Instead, we design a reward function using only the *number* of error messages generated by the compiler. The goal of the agent is to perform edits necessary for successful compilation of the program.

To see our framework in action, consider the example C program shown in Figure 1 again. This program is given to a *trained* agent which has not seen it during training. The program is presented in a tokenized form to the agent and the cursor position of the agent is initialized to the first token in the program. The navigation actions of the agent over the program text are shown by arrows. As shown by the sequence of actions taken by the agent in Figure 1, the agent correctly localizes and fixes both the errors. First, the agent navigates to the error location at line 4 and replaces the incorrect semicolon with a comma (marked by e_1). Next, it navigates to the error location at line 12 and inserts the missing closing brace (marked by e_2). After these edits, the program compiles successfully and the agent stops. In comparison, a brute-force search will have to enumerate all possible edited versions of the program with up to two simultaneous edits and compile each of them to identify the right edits. This will be far more computationally expensive than using the agent. A video demonstration of this example is available at: https://youtu.be/kNtBT1fgJ-0

We represent the program text, augmented with the position of the cursor as a sequence of tokens which is then embedded using a long short-term memory (LSTM) network (Hochreiter and Schmidhuber 1997). The agent is allowed a set of navigation and edit actions to fix the program. It receives a small reward for every edit action which fixes some compiler error and the maximum reward is given for reaching the goal state, which is a compiler error-free version of the program. The control policy of the agent is learned using the asynchronous advantage actor-critic (A3C) algorithm (Mnih et al. 2016).

Training an agent in this setting is non-trivial. As illustrated by the example above, the agent needs to both localize the errors and make precise edits at the error locations to be able to fix a program. A wrong edit only makes the program worse by introducing more errors and consequently, makes the task even more difficult. To overcome this, we configure the environment to reject all the edits that do not reduce the number of compilation errors. This significantly prunes the state space that an agent is allowed to explore, allowing it to train in a reasonable amount of time. This also prevents an agent from performing arbitrary edits such as deleting erroneous lines. We call our technique *RLAssist*.

DeepFix (Gupta et al. 2017) is a state-of-the-art, end-to-end syntactic error repair technique. We compare RLAssist with DeepFix on the task of fixing typographic errors in 6975 C programs from a publicly available dataset. These programs were written by students during an introductory programming course and span 93 different programming problems (Gupta et al. 2017; Das et al. 2017). These programs use non-trivial constructs of the C language such as conditionals, switch statements, nested loops, multi-dimensional arrays, multiple procedures, and recursion. DeepFix uses a fully supervised neural machine translation approach.

In contrast, RLAssist is a deep reinforcement learning based technique. We demonstrate that RLAssist outperforms DeepFix, although it is trained through self-exploration using erroneous programs only, i.e., without any supervision. RLAssist fixes 24.4% programs from the test set completely and resolves 35.1% error messages. Relative to DeepFix, this is an improvement of 4.6% and 13.8% respectively. We further show that we can accelerate the training of RLAssist by a factor of 5 by leveraging expert demonstrations for only one tenth of the training dataset. The main contributions of this work are as follows:

1. We design a novel framework for programming language correction amenable to reinforcement learning and train our agent using A3C.
2. We empirically show that our technique, RLAssist, outperforms a fully supervised state-of-the-art syntactic error repair technique, DeepFix, on a publicly available dataset

of thousands of C programs.

3. We achieve more than $5\times$ training speedup for RLAssist using expert demonstrations for only one tenth of its training data.
4. We provide the source code of RLAssist online at: https://bitbucket.org/iiscseal/rlassist.

Background

Reinforcement Learning

In reinforcement learning, an agent interacts with its environment over a number of discrete time steps. At each time step t, the environment presents a state $s_t \in \mathcal{S}$ to the agent. In response, the agent selects an action a_t from the set of allowed actions $\mathcal{A}$. This selection is controlled by the agent's policy $\pi(a|s) = Pr\{a_t = a|s_t = s\}$. The action is passed on to the environment and its execution may modify the internal state of the environment. The agent then receives the updated state s_{t+1} and a scalar reward r_t. This interaction, which is also called an episode, stops when the agent reaches a goal state. The objective of the agent is to maximize the expected sum of discounted future rewards G_t from each state s_t. For an episode terminating at time step T, $G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k}$, where the discount rate $\gamma \in (0, 1]$ (Sutton and Barto 1998).

Asynchronous Advantage Actor-Critic (A3C)

One of the many ways to solve an RL problem is to use policy gradient methods. These methods learn a parameterized policy $\pi(a|s;\theta)$, where θ represents the parameters of a function approximator, such as a neural network. One example of policy gradient methods is the actor-critic methods, which learn both $\pi(a|s;\theta)$, the 'actor', and the value function $V(s;w)$, the 'critic'. Here, $V(s;w)$ defines the expected reward from state s, with w being the parameters of a function approximator. The critic evaluates how advantageous is it to be in the new state reached by taking an action a_t sampled from the distribution given by $\pi(s_t)$. Based on this evaluation, the parameterized policy is updated using an appropriate optimization technique such as gradient ascent (Sutton and Barto 1998).

The A3C algorithm uses multiple asynchronous parallel actor-learner threads to update a shared model, stabilizing the learning process by reducing the correlation of an agent's experience. It has also been observed to reduce training time by a factor that is roughly linear in the number of parallel actor-learners (Mnih et al. 2016). We use A3C in this work.

Technical Details

A Framework for Programming Language Correction Tasks

When faced with an error, a programmer navigates the program text to arrive at the location of error and then performs an edit operation to fix the error. In the presence of multiple errors, the programmer can repeat these steps. We present a programming language (PL) correction framework in which an agent can mimic these actions. Figure 2a shows the block diagram of our PL correction framework. We now describe the components of this framework and their instantiation for our task of correcting syntax errors in C programs.

States A state is a pair $\langle string, cursor\rangle$, where $string$ is the program text, and $cursor \in \{1, \ldots, len(string)\}$, where $len(string)$ denotes the number of tokens in the $string$. The environment also keeps track of the number of errors in $string$. These errors can either be determined from the ground truth whenever available or estimated using the error messages generated by a compiler upon compiling the $string$. For compilation, we use the GNU C compiler.

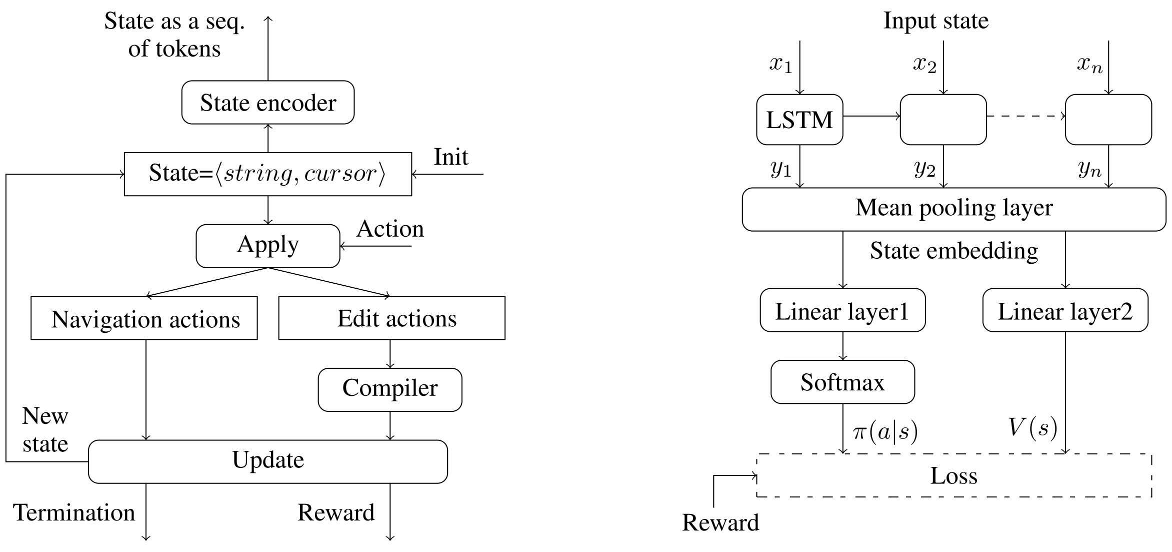
We encode the state into a sequence of tokens as follows. First, we convert the program $string$ into a sequence of lexemes. The lexemes are of different types, such as keywords, operators, types, functions, literals, and variables. In addition, we also retain line-breaks as lexemes to allow two-dimensional navigation actions over the program text. We use a special token to represent $cursor$, which is inserted in the sequence of lexemes right after the token whose index is held by $cursor$.

Next, we build a shared vocabulary across all programs. Except some common functions such as `printf` and `scanf`, all other function and variable identifiers are mapped to a special token ID. Similarly, all the literals are mapped to special tokens according to their type, e.g., numbers to `NUM` and strings to `STR`. All remaining tokens are retained without any modifications. This mapping reduces the size of the vocabulary seen by the agent.

Note that this encoding is only required for feeding the state to the agent. The actions predicted by the agent based on this encoding are executed by the environment on the original program $string$.

Actions and Transitions The agent actions are divided into two categories, the first which update the $cursor$ and the second which modify the $string$. We refer to the first category of actions as *navigation actions* and the latter as *edit actions*. The navigation actions allow an agent to navigate through the $string$. These actions only change the $cursor$ of a state and not the $string$. The edit actions on the other hand, are used for error correction. They only modify the $string$ and not the $cursor$. Wrong edit actions introduce more errors in the $string$ rather than fixing them. An edit action is categorized as wrong if it does not reduce the number of error messages in the program $string$. We configure the environment to reject all such edits to prune the state space from which fixing the program becomes even more difficult. This also prevents an agent from performing arbitrary edits such as deleting erroneous lines.

For our task, we allow only two navigation actions, *move_right* and *move_down*. These set the $cursor$ to the next token on the right or to the first token of the next line respectively. The *move_right* (respectively, *move_down*) action has no effect if the $cursor$ is already set to the last token of a line (respectively, any token of the last line). Note that the *move_down* action is possible because we retain the line-breaks in the state encoding. This choice of navigation actions allows an agent to systematically reach all potential error locations in the program while keeping the number of



(a) PL correction framework. (b) RLAssist agent.

Figure 2: Design of RLAssist.

navigation actions low.

Based on our study of common typographic errors that novice programmers make, we design three types of edit actions. The first is a parameterized *insert_token* action which inserts the parameter *token* immediately before the $cursor$ position. The parameter can be any token from a fixed set of tokens which we call *mutable tokens*. The second is the *delete* action, which deletes the token at the $cursor$ position. However, the token is deleted only if it is from the set of mutable tokens. We restrict the set of mutable tokens to the following five types of tokens: semicolon, parentheses, braces, period, and comma. The third edit action is a parameterized *replace token1 with token2* action which replaces *token1* at the cursor position with *token2*. We have the following four actions in this class: (1) *replace '`;`' with '`,`'*, (2) *replace '`,`' with '`;`'*, (3) *replace '`.`' with '`;`'*, and (4) *replace '`;)`' with '`);`'*. Although atomic replacement actions can be substituted with a sequence of delete and insert actions, having them prevents the cases where the constituent delete and/or insert actions can be rejected by the environment.

Episode, Termination, and Rewards An episode starts with an erroneous program text as $string$ and the $curosr$ set to its first token. The goal state is reached when the edited program compiles successfully. An agent is allowed $max\_episode\_len$ number of discrete time steps to reach the goal state in an episode after which the episode is terminated. Also, the agent is allowed only one pass over the program in an episode, i.e., once the agent navigates past the last token of a program, the episode is terminated. In each step, the agent is penalized with a small $step\_penalty$ in order to encourage it to learn to fix a program in the minimum number of steps. The agent is given $maximum\_reward$ for reaching the goal state. Also, a small $intermediate\_reward$ is given for taking an edit action that fixes at least one error.

Model

We use the A3C algorithm for the programming language correction task. Figure 2b shows the block diagram of our agent. Our model first uses a long short-term memory (LSTM) (Hochreiter and Schmidhuber 1997) network for embedding the tokenized state into a real vector. The LSTM network maps each token x_i of an input sequence $(x_1, \ldots, x_n)$ to a real vector y_i. The final state embedding is calculated by taking an element-wise mean over all the output vectors $(y_1, \ldots, y_n)$ following (Narasimhan, Kulkarni, and Barzilay 2015). Given this state embedding, we use two separate fully connected linear layers to produce the policy function $\pi(a|s;\theta)$, and the value function $V(s;w)$ outputs. Finally, before updating the network parameters, the gradients are accumulated using the following rules:

$$\begin{aligned} d\theta &\leftarrow d\theta + \nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R - V(s_t;w')) \\ &\quad + \beta \nabla_{\theta'} H(\pi(s_t;\theta')) \\ dw &\leftarrow dw + \nabla_{w'}(R - V(s_t;w'))^2 \end{aligned}$$

where $R = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};w')$, H is the entropy, and β is a hyperparameter to control the weight of the entropy regularization term; θ' and w' are the thread specific parameters corresponding to θ and w respectively (Mnih et al. 2016).

Experiments

Dataset

We use the publicly available dataset originally developed in the DeepFix (Gupta et al. 2017) work. The programs in the dataset span 93 programming problems in an introductory programming course and make use of non-trivial C language constructs. The program lengths range from 75 to 450 tokens. DeepFix uses five fold cross validation for its experiments by partitioning the 93 programming problems. In each

Dataset statistics			Technique	Results		
Erroneous programs	Error msgs.	Avg. tokens		Completely fixed programs	Partially fixed programs	Error messages resolved
6975	16766	203	DeepFix	1625 (23.3%)	1129 (16.2%)	5156 (30.8%)
			RLAssist	1699 (24.4%)	1310 (18.8%)	5884 (35.1%)
			RLAssist+Demo	1854 (26.6%)	1426 (20.4%)	6652 (39.7%)

Table 1: Summary of the test dataset and performance comparison of DeepFix, RLAssist, and RLAssist+Demo on it for typographic errors. Note that we take the most recent version of the dataset and improved results for DeepFix from its webpage: https://bitbucket.org/iiscseal/deepfix.

fold, the correct programs for the 4/5th of the programming problems are used to generate about 165K training examples. Each correct program x is mutated to generate up to five training examples of the form $(x\prime, y\prime)$, where $x\prime$ is the mutated program and $y\prime$ is the required correction to it. The erroneous programs for the remaining 1/5th of the programming problems are used for testing without any modification. Summed across all the folds, the test set contains 6975 erroneous programs with 16766 compilation error messages, as shown in Table 1. This experimental setup requires a learning algorithm to learn the syntactic validity as per the language syntax so that it can generalize to unseen programming problems.

For our experiments, we use the same experimental setup as DeepFix. However, we use only the incorrect programs ($x\prime$) from the training dataset and discard the corresponding fixes ($y\prime$).

Experiment Configuration and Training

We implement our technique in Tensorflow (Abadi et al. 2016). We find a suitable configuration of the PL correction framework and the learning model for our task through experimentation. In particular, the LSTM encoder in our model has two recurrent layers with 128 cells each. Our vocabulary has 91 tokens, which are embedded into 24-dimensional vectors. We set the discounting factor $\gamma = 0.99$, the maximum number of exploration steps before a neural network parameter update is made $t_{max} = 24$, and entropy regularization factor $\beta = 0.01$. The LSTM encoder for state embedding and the policy network are trained together in an end-to-end manner. We use 32 parallel agents, and a learning rate of 0.0001 for optimizing our model using the ADAM optimizer (Kingma and Ba 2014). We also use gradient clipping to prevent the gradients from exploding (Pascanu, Mikolov, and Bengio 2013). We configure the PL correction framework for our task by setting $max\_episode\_len = 100$, $step\_penalty = -0.01$, $maximum\_reward = 1$, and $intermediate\_reward = 0.02$. We train RLAssist for 30 epochs (1 epoch $\approx$ 165K episodes), which takes 3 weeks on an Intel(R) Xeon(R) E5-2630 v4 machine, clocked at 2.20GHz with 32GB of RAM.

Evaluation

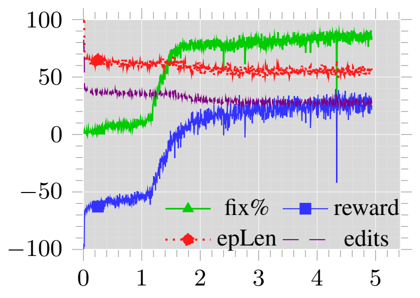
In this section, we first discuss the training performance of RLAssist. Next, we discuss how we can accelerate the training of RLAssist by leveraging only a small amount of expert demonstrations. Later, we compare it with DeepFix on the test dataset described earlier.

Training Performance of RLAssist In order to evaluate the training performance of RLAssist, we use the following metrics: (1) the percentage of error messages resolved, (2) the average episode length, (3) the average number of edit actions, and (4) the average reward obtained by the agent as the training progresses. We report the training performance on one of the folds. For ease of plotting, the average reward shown in the figures is scaled by a factor of 100. Figure 3a illustrates the training performance of RLAssist.

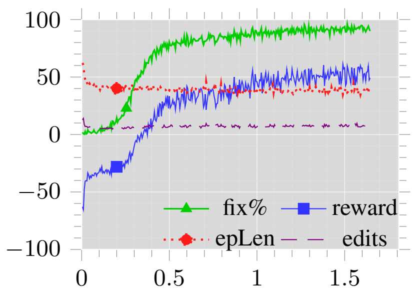
RLAssist learns to solve the task very well and is able to resolve about 90% of the error messages after training for about four million episodes. In the last epoch of training, the agent reaches the goal state for 79% of the episodes. Furthermore, it manages to resolve 36% of the error messages for the programs corresponding to the remaining 21% episodes. At the same time, the average scaled reward also reaches the maximum of about 30 from -100, the scaled reward obtained at the beginning of the training. The maximum scaled reward is almost always less than 100 because of the penalty that the agent incurs for navigating to the error location. The average length of an episode comes down to 54 from the maximum allowed episode length of 100 consisting about 27 navigation and 28 edit actions. This is much smaller than what a systematic brute-force search would require in a combinatorial search space. For the 79% of the programs that RLAssist fixes in the last epoch, the average number of rejected edit actions per episode is only about 5. This shows that RLAssist not only learns to fix a program but it also learns to do so by taking reasonably precise navigation and edit actions.

Accelerating Training with Expert Demonstrations Reinforcement learning tends to be slow for the tasks with large state spaces as the time required for gathering information by state exploration increases. One way to mitigate this problem is to guide the agent with expert demonstrations (Argall et al. 2009). Motivated by this, we further design a scheme to enable the agent to take advantage of expert demonstrations.

We configure an agent to use expert demonstrations as follows. For the episodes for which a demonstration is available, the agent follows the predetermined sequence of ac-



(a) RLAssist



(b) RLAssist+Demo

Figure 3: Training performance of RLAssist and RLAssist+Demo. The X-axis shows the number of training episodes in millions. *epLen* and *fix%* stand for episode length and the percentage of error messages resolved, respectively.

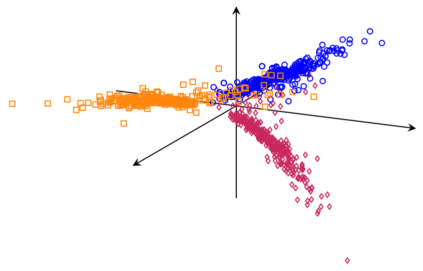


Figure 4: PCA projection of embeddings of 350 programs in three different states. The first, the second, and the third states (defined in the text) are shown by diamonds, squares, and circles, respectively.

tions provided, instead of sampling driven by the policy. The updates to the policy network parameters are made as if the predetermined action was sampled. For the rest of the episodes, the agent takes the actions sampled using the policy, following the standard A3C algorithm. We derive demonstrations for *one tenth* of training programs from their fixes. Additionally, we add a small $edit\_penalty$ to discourage the agent from taking unnecessary edit actions. We refer to this configuration as *RLAssist+Demo*.

As shown in Figure 3b, RLAssist+Demo benefits immensely from the demonstrations and starts making visible progress after training for only 0.2 million episodes compared to 1.2 million episodes for RLAssist. It resolves about 90% of the error messages after training for about a million episodes, 3 millions episodes earlier than RLAssist. The additional $edit\_penalty$ also helps in reducing the number of average edit actions per episode by a factor of 3. Note that RLAssist fails to train with $edit\_penalty$ as it stops exploring edit actions very early in the training due to more negative reward.

This shows that while RLAssist is able to train with erroneous programs alone, it can benefit when even a small amount of labeled data (i.e., erroneous programs and their fixes) is available. We train RLAssist+Demo for 10 epochs, which takes about 4 days on the same machine used for

	Completely fixed programs	Partially fixed programs	Completely or partially fixed programs
DeepFix*	264	247	398
RLAssist*	335	428	650

Table 2: Comparison of DeepFix and RLAssist on the number of programs fixed ***exclusively** by each technique.

RLAssist, achieving more than 5 times speedup in training time. In our experiments, we observed that using more demonstrations did not result in significant speedup in training while reducing the demonstrations slowed it down.

Comparison with DeepFix In Table 1, we show the comparison of RLAssist and RLAssist+Demo with DeepFix (Gupta et al. 2017) on the test dataset. For this comparison, we use the number of error messages resolved, and completely and partially fixed programs; the same metrics as reported in (Gupta et al. 2017). Further in Table 2, we also report the number of programs which are fixed exclusively by DeepFix or RLAssist.

As shown in Table 1, the test dataset has 16766 error messages from 6975 erroneous programs out of which DeepFix resolves 5156 error messages, fixing 1625 programs completely and 1129 partially. RLAssist resolves 5884 error messages, fixing 1699 programs completely and 1310 partially. Thus RLAssist outperforms DeepFix by a relative margin of 13.8% and 4.6% in terms of error messages resolved and completely fixed programs, respectively. Relative to DeepFix, the percentage of programs fixed partially by RLAssist is 15.2% higher. At test time, both RLAssist and DeepFix take less than a second to fix a program. RLAssist+Demo performs even better and resolves 6652 error messages, fixing 1854 programs completely and 1426 partially. It outperforms DeepFix by a relative margin of 29% and 14.1% in terms of error messages resolved and completely fixed programs, respectively. Relative to DeepFix, it fixes 26.3% more programs partially.

One reason for better performance of RLAssist is that it works at a finer token-level granularity compared to the coarser line-level granularity of DeepFix. RLAssist can edit an incorrect line in place, whereas DeepFix has to produce a complete replacement of the erroneous line. This requires it to copy the correct token subsequences from the input while simultaneously rectifying the erroneous tokens. Another reason is that DeepFix halts when it cannot fix a line, i.e., if it fails to fix an erroneous line, it cannot fix the subsequent erroneous lines. This limitation arises because of the iterative nature of DeepFix. If a fix suggested by DeepFix is accepted by the compiler, the fix is applied and the updated program is shown to DeepFix to identify the next fix. However, if it is rejected, the iterative procedure stops. RLAssist, on the other hand, does not have this limitation. If the action taken by the agent is rejected by the environment, it takes the next highest probability action and continues to attempt other fixes.

Embedding Visualization We select 350 test programs containing only one error per program. Next, we get embeddings corresponding to the following three states of each of these programs: (1) when the cursor is set to the first token of the line preceding the erroneous line, (2) when the cursor is set to the first token of the erroneous line, and (3) when the cursor is set to the error location and the program has been fixed. Figure 4 shows the first three principal components of these embeddings. It can be seen that the three states form three distinct clusters with almost no overlap. This shows that RLAssist's encoder learns to capture not only the syntactic validity of the programs but also the location of the errors in them.

Related Work

Natural language correction is a well researched problem. Some of the earlier works in this area have focused on identifying and correcting specific types of grammatical errors such as misuse of verb forms, articles, and prepositions (Han, Chodorow, and Leacock 2006; Chodorow, Tetreault, and Han 2007; Rozovskaya and Roth 2010). The more recent works consider a broader range of error classes, often relying on language models, and machine translation (Ng et al. 2014; Rozovskaya et al. 2014). Although natural languages and programming languages are similar to some extent, the latter have procedural interpretation and richer structure.

Due to the huge popularity of programming oriented massive open online courses (MOOCs), recent years have seen increasing interest in developing automated techniques for syntactic error repair in student programs. *sk_p* (Pu et al. 2016) and *SynFix* (Bhatia, Kohli, and Singh 2018) learn syntactic error repair limited to a specific programming problem by training neural networks on the correct submissions of the same problem. Therefore, unlike RLAssist, these techniques cannot be used for real time feedback generation for novel programming problems as they need to train their model for each problem separately. TRACER (Ahmed et al. 2018) uses both hand designed abstractions and supervised learning techniques for learning error correction models on abstract statements. It takes only an incorrect statement as its input, and consequently fails to handle many frequent errors requiring global context such as opening/closing a missing brace. RLAssist, on the other hand, takes complete program as input and does not suffer from this limitation. While SynFix and TRACER use compiler based heuristics, sk_p performs a brute force, enumerative search for localizing errors.

DeepFix (Gupta et al. 2017) is a state-of-the-art, end-to-end, supervised syntactic error repair technique. It is more general than the previous three techniques as it learns to predict both the error locations and the fixes without relying on any heuristics. We propose a reinforcement learning framework in which an agent can learn syntactic error repair directly from raw program text through self-exploration, i.e. without any supervision.

Syntactic error repair techniques for student programs are useful for not only students but also instructors. A recent study (Yi et al. 2017) noted that automated program repair techniques can be used by human graders to decrease the amount of grading time. *GradeIT* (Parihar et al. 2017) uses simple rewrite rule based syntactic error repair to grade submissions that do not compile. Integrating our technique can further improve the performance of such systems.

Through deep reinforcement learning, agents have been trained to play visual and text-based games at expert levels (Mnih et al. 2015; Narasimhan, Kulkarni, and Barzilay 2015; Wu and Tian 2016). At a high level, our problem is similar to text-based games in which both the state space and cues for completing the quest are given in a textual form (Narasimhan, Kulkarni, and Barzilay 2015). However, the state space in these games is manually created by their developers and is of small size. Similarly, the transitions between states is also constrained. In contrast, the state space for our problem is combinatorial in the size of a given program. Training an agent is challenging in this large state space, which we overcome by constraining the agent to explore only those states which take it closer to the goal state.

Learning from demonstration (LfD) approaches train an agent using expert demonstrations. Behavioral cloning (Ross, Gordon, and Bagnell 2011) is one particular class of LfD techniques to make the agent mimic expert demonstrations using supervised learning. We complement the self-exploration with expert demonstrations to accelerate the training. Inverse reinforcement learning is another class of LfD techniques. These use demonstrations to first infer a reward function which is then used to learn the policy (Abbeel and Ng 2004). These methods have been used for the tasks where there is no obvious reward function, e.g., autonomous driving (Abbeel and Ng 2004) and acrobatic helicopter maneuvers (Abbeel et al. 2007). For our task, the rewards are well defined and can be calculated easily.

Conclusions and Future Work

We address the problem of syntactic error repair in student programs and present a novel deep reinforcement learning based solution, called RLAssist, for it. We compare RLAssist with a state-of-the-art technique, DeepFix, on the task of correcting typographic errors in 6975 student-written erroneous C programs. Our experiments show that RLAssist outperforms DeepFix without using any labeled data for training. Moreover, we show that RLAssist trains much faster and converges to a better policy when expert demonstrations are available for as little as one tenth of its training data.

RLAssist is programming language agnostic and has been evaluated on C programs. In future, we will experiment with other programming languages as well. We plan to extend RLAssist to target more classes of errors, and devise RL algorithms that can learn and exploit deeper syntactic and semantic properties of programs.

Acknowledgments

We thank Sonata Software Ltd. for partially funding this work. We also thank the anonymous reviewers for their helpful feedback.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, volume 16, 265–283.

Abbeel, P., and Ng, A. Y. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, 1.

Abbeel, P.; Coates, A.; Quigley, M.; and Ng, A. Y. 2007. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems*, 1–8.

Ahmed, U. Z.; Kumar, P.; Karkare, A.; Kar, P.; and Gulwani, S. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 78–87.

Argall, B.; Chernova, S.; Veloso, M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Robotics and Autonomous Systems* 67:469–483.

Bhatia, S.; Kohli, P.; and Singh, R. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, 60–70.

Chodorow, M.; Tetreault, J. R.; and Han, N.-R. 2007. Detection of grammatical errors involving prepositions. In *Proceedings of the 4th ACL-SIGSEM workshop on prepositions*, 25–30.

Das, R.; Ahmed, U. Z.; Karkare, A.; and Gulwani, S. 2017. Prutor: A system for tutoring cs1 and collecting student programs for analysis. https://www.cse.iitk.ac.in/users/karkare/prutor/.

Gupta, R.; Pal, S.; Kanade, A.; and Shevade, S. 2017. DeepFix: Fixing common c language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, 1345–1351.

Han, N.-R.; Chodorow, M.; and Leacock, C. 2006. Detecting errors in English article usage by non-native speakers. *Natural Language Engineering* 12(2):115–129.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Kaleeswaran, S.; Santhiar, A.; Kanade, A.; and Gulwani, S. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 739–750.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, 1928–1937.

Narasimhan, K.; Kulkarni, T. D.; and Barzilay, R. 2015. Language understanding for text based games using deep reinforcement learning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 1–11.

Ng, H. T.; Wu, S. M.; Briscoe, T.; Hadiwinoto, C.; Susanto, R. H.; and Bryant, C. 2014. The CoNLL-2014 shared task on grammatical error correction. In *CoNLL Shared Task*, 1–14.

Parihar, S.; Dadachanji, Z.; Singh, P. K.; Das, R.; Karkare, A.; and Bhattacharya, A. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education*, 92–97.

Pascanu, R.; Mikolov, T.; and Bengio, Y. 2013. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 1310–1318.

Piech, C.; Huang, J.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. 2009. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 961–968.

Pu, Y.; Narasimhan, K.; Solar-Lezama, A.; and Barzilay, R. 2016. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 39–40.

Ross, S.; Gordon, G. J.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 627–635.

Rozovskaya, A., and Roth, D. 2010. Generating confusion sets for context-sensitive error correction. In *Proceedings of the conference on empirical methods in natural language processing*, 961–970.

Rozovskaya, A.; Chang, K.-W.; Sammons, M.; Roth, D.; and Habash, N. 2014. The illinois-columbia system in the CoNLL-2014 shared task. In *CoNLL Shared Task*, 34–42.

Singh, R.; Gulwani, S.; and Solar-Lezama, A. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48(6):15–26.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Traver, V. J. 2010. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction* 2010:3:1–3:26.

Wu, Y., and Tian, Y. 2016. Training agent for first-person shooter game with actor-critic curriculum learning.

Yi, J.; Ahmed, U. Z.; Karkare, A.; Tan, S. H.; and Roychoudhury, A. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 740–751.