DeepFix: Fixing Common C Language Errors by Deep Learning

Rahul Gupta, Soham Pal, Aditya Kanade, Shirish Shevade
Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India
{rahul.gupta, soham.pal, kanade, shirish}@csa.iisc.ernet.in

Abstract

The problem of automatically fixing programming errors is a very active research topic in software engineering. This is a challenging problem as fixing even a single error may require analysis of the entire program. In practice, a number of errors arise due to programmer's inexperience with the programming language or lack of attention to detail. We call these common programming errors. These are analogous to grammatical errors in natural languages. Compilers detect such errors, but their error messages are usually inaccurate. In this work, we present an end-to-end solution, called DeepFix, that can fix multiple such errors in a program without relying on any external tool to locate or fix them. At the heart of DeepFix is a multi-layered sequence-to-sequence neural network with attention which is trained to predict erroneous program locations along with the required correct statements. On a set of 6971 erroneous C programs written by students for 93 programming tasks, DeepFix could fix 1881 (27%) programs *completely* and 1338 (19%) programs partially.

Introduction

Debugging programming errors is one of the most time-consuming activities for programmers. Therefore, the problem of automatically fixing programming errors, also called *program repair*, is a very active research topic in software engineering (Monperrus 2015). Most of the program repair techniques focus on logical errors in programs. Using a specification of the program (such as a test suite or an assertion), they attempt to fix the program. Since their focus is on fixing logical errors in individual programs, they assume that the program compiles successfully.

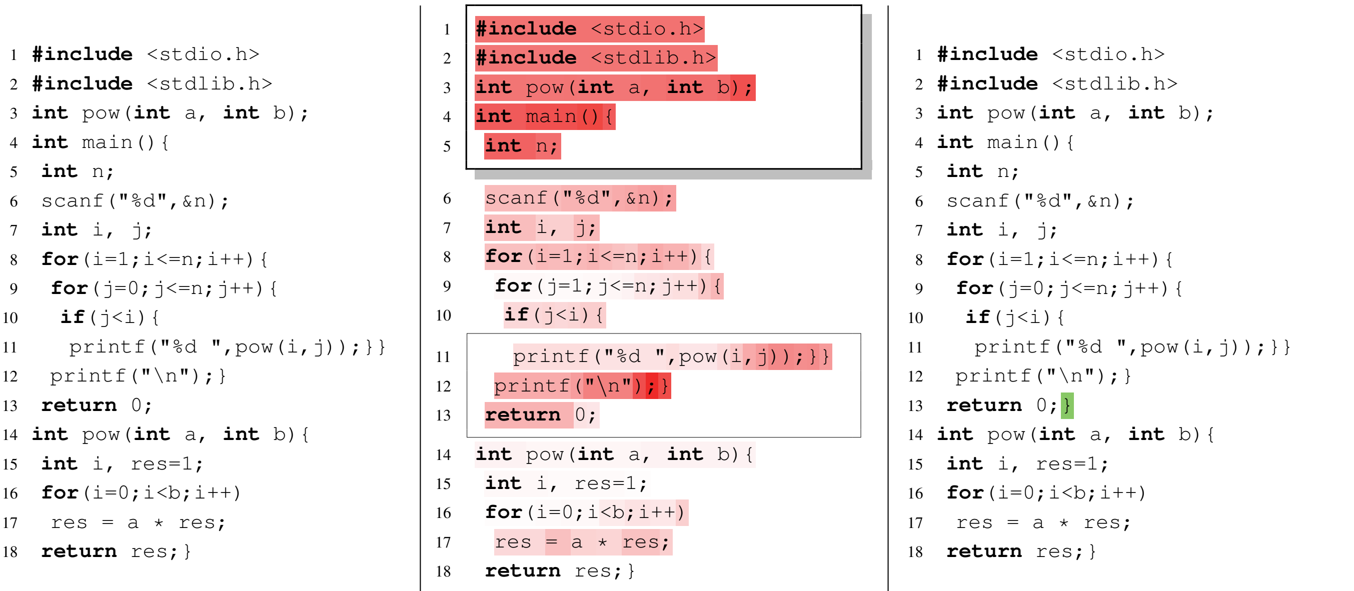
This leaves a large and frequent class of errors out of the purview of the existing techniques. These include errors due to missing scope delimiters (such as a closing brace), adding extraneous symbols, using incompatible operators or missing variable declarations. Such mistakes arise due to programmer's inexperience or lack of attention to detail, and cause compilation or build errors. Not only novice students but experienced developers also make such errors as found in a study of build errors at Google (Seo et al. 2014). We call them *common programming errors*. These are common in the sense that, unlike logical errors, they are not specific to the programming task at hand, but relate to the overall syntax and structure of the programming language. These are analogous to grammatical errors in natural languages.

We propose to fix common programming errors by deep learning. Because of the inter-dependencies among different parts of a program, fixing even a single error may require analysis of the entire program. This makes it challenging to fix them automatically. The accuracy bar is also quite high for program repair. The usual notion of token-level accuracy is much too relaxed for this setting. For a fix to be correct, the repair tool must produce the entire sequence pertaining to the fix precisely.

We present an *end-to-end solution*, called DeepFix, that does not use any external tool to localize or fix errors. We use a compiler only to validate the fixes suggested by DeepFix. At the heart of DeepFix is a multi-layered sequence-to-sequence neural network with attention (Bahdanau, Cho, and Bengio 2014), comprising of an encoder recurrent neural network (RNN) to process the input and a decoder RNN with attention that generates the output. The network is trained to predict an erroneous program location along with the correct statement. DeepFix invokes it iteratively to fix multiple errors in the program one-by-one.

Figure 1(a) shows an input C program `p.c` with a missing closing brace at line 13. Figure 1(c) shows the program after the fix suggested by DeepFix is applied. DeepFix correctly predicts that line 13 has an error and generates a sequence with a closing brace inserted after "`return 0;`". To gain an insight into how our network predicts the fix correctly, we visualize the attention weights assigned by DeepFix to each token in the input program in Figure 1(b). The background color of a token is proportional to the average attention weight assigned to it by the network while predicting the sequence of tokens in the fix. We observe that the network captures a local context closer to the faulty line (enclosed by a box) and a global context (enclosed by a shadow-box) which also contains the declaration of `main` at line 4 with the *unmatched opening brace*. It is also interesting to note that attention is comparatively less for lines 6–10 and also for lines 14–18. Lines 14–18 define another function `pow` that does not have a bearing on the error being fixed.

Compilers can detect common programming errors, but they usually do not pinpoint error locations accurately (Traver 2010). For example, if we compile the program in



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int pow(int a, int b);
4  int main(){
5   int n;
6   scanf("%d",&n);
7   int i, j;
8   for(i=1;i<=n;i++){
9    for(j=0;j<=n;j++){
10    if(j<i){
11     printf("%d ",pow(i,j));}}
12   printf("\n");}
13  return 0;
14 int pow(int a, int b){
15  int i, res=1;
16  for(i=0;i<b;i++)
17   res = a * res;
18  return res;}
```

(a) Input program p.c with a missing closing brace at line 13

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int pow(int a, int b);
4  int main(){
5   int n;
6   scanf("%d",&n);
7   int i, j;
8   for(i=1;i<=n;i++){
9    for(j=1;j<=n;j++){
10    if(j<i){
11     printf("%d ",pow(i,j));}}
12   printf("\n");}
13  return 0;
14 int pow(int a, int b){
15  int i, res=1;
16  for(i=0;i<b;i++)
17   res = a * res;
18  return res;}
```

(b) Attention weights (darker the background, higher the weight)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int pow(int a, int b);
4  int main(){
5   int n;
6   scanf("%d",&n);
7   int i, j;
8   for(i=1;i<=n;i++){
9    for(j=0;j<=n;j++){
10    if(j<i){
11     printf("%d ",pow(i,j));}}
12   printf("\n");}
13  return 0;}
14 int pow(int a, int b){
15  int i, res=1;
16  for(i=0;i<b;i++)
17   res = a * res;
18  return res;}
```

(c) The program after the fix (shaded) suggested by DeepFix is applied

Figure 1: Example to illustrate the DeepFix approach.

Figure 1(a) through the GCC compiler, we get the following error message:

```
p.c: In function 'main':
p.c:18:2: error: expected declaration or
      statement at end of input
   return res;}
   ^
```

The error message is both cryptic and misleading. It does not localize the error to line 13. Instead, it wrongly implicates line 18 in the program text.

Common programming errors are not specific to any particular programming task. Two recent approaches (Bhatia and Singh 2016; Pu et al. 2016) train neural networks on correct student solutions to a *specific* programming task and try to learn task-specific patterns for fixing erroneous solutions for the *same* task. In comparison, DeepFix can be used on solutions to any unseen programming task. Unlike DeepFix, the neural networks used in the above-mentioned approaches cannot localize errors on their own. Bhatia and Singh (2016) use prefixes implicated by a compiler and Pu et al. (2016) perform a brute force, enumerative search for localizing errors. Based on the compiler message above, the approach in (Bhatia and Singh 2016) would treat a prefix up to line 18 in Figure 1(a) as correct and try to predict a suffix. The error however is much earlier at line 13 and hence, cannot be fixed by it. Fixing non-trivial errors requires reasoning about long term dependencies in the program text. Pu et al. (2016) train a language model to predict an intermediate line given the statements at the previous and next lines, and thus, capture only a short local context. DeepFix uses an attention based sequence-to-sequence model to capture long term dependencies.

We apply DeepFix on C programs written by students for 93 different programming tasks in an introductory programming course. The nature of these tasks and thereby, that of the programs vary widely. Nevertheless, our network generalizes well to programs from across these tasks. Out of 6971 erroneous programs, DeepFix fixed 1881 (27%) programs *completely* and 1338 (19%) programs partially.

The main contributions of this work are as follows:

1. It introduces the issue of common programming errors and offers an end-to-end solution based on deep nets.
2. It can iteratively fix multiple errors in the same program.
3. The technique is evaluated on thousands of erroneous C programs with encouraging results.

Related Work

In recent years, deep learning algorithms have been successfully applied to a variety of tasks in natural language processing. Though there exist similarities between natural languages and programming languages, programs are characterized by rich structural information. Hindle et al. (2012) have explored regularities in program text which they refer to as "naturalness" in an analogy to natural language text. The software engineering community has successfully applied deep learning to problems such as API mining, code migration and code categorization. For more details, we refer the reader to (White et al. 2015) and the references therein.

Genetic programming has been used for discovering fixes for programs (Arcuri 2008; Debroy and Wong 2010; Le Goues et al. 2012). These techniques typically rely on redundancy present in other parts of the program to restrict the search space of mutants. Long and Rinard (2016) learn a probabilistic model using explicitly designed code features to rank repair candidates. However, the repair candidates are enumerated separately and not predicted as in our work.

The popularity of massive open online courses (MOOCs) in the recent past is evident from the large number of registrants for such courses. Piech et al. (2015) proposed a neural network based approach to find program representations and used them for automatically propagating instructor feedback to students in a massive course. Their approach is based on an encoder-decoder framework and is useful especially in the discrete gridworld-type programming problems. We consider the far more general class of C programs and generate fixes automatically.

We pose the problem of program repair as that of generating an output sequence (a fix) from an input sequence (an erroneous program). Sutskever, Vinyals, and Le (2014) introduced a neural network model for solving the sequence-to-sequence learning problem under the general framework of neural machine translation. The main drawback of this approach is the use of a fixed length vector for encoding information related to source sequences of various lengths. Bahdanau, Cho, and Bengio (2014) alleviated this problem by proposing a model with an attention mechanism. This model focuses on the most relevant information in a source sequence by appropriate search mechanism and has the capability of handling long distance relations well. Vinyals et al. (2015) achieved state-of-the-art results for the problem of syntactic constituency parsing using this architecture. Xie et al. (2016) use a character-level attention based mechanism for natural language correction. To our knowledge, ours is the first end-to-end solution which uses a deep network for localizing and fixing common programming errors.

Technical Details

In this section, we discuss the overall design of DeepFix.

Program Representation

We pose the problem of fixing a programming error as a sequence-to-sequence learning problem. This requires a program to be represented as a sequence. We now give details of our program representation.

Program text consists of different kinds of tokens such as types, keywords, special characters (e.g., semicolons), functions, literals and variables. Among these, types, keywords, special characters and library functions form a *shared vocabulary* across different programs. We retain them while representing a program. We model the other types of tokens as follows. We first define a fixed-size pool of names and then construct a separate *encoding map* for each program by randomly mapping each distinct identifier (variable or function name) in the program to a unique name in our pool. We choose a pool that is large enough to create the above mapping for any program in our dataset. This transformation does not change the semantics of the program and is reversible. The exact values of literals do not matter for our learning task. We therefore map a literal to a special token based on its type, e.g., we map all integer literals to NUM and all string literals to STR. We use a special token <eos> to denote the end of a token sequence.

We treat a program as a sequence of tokens X. We may want a network to produce another sequence Y such that Y fixes errors in X. However, a typical program we consider has a few hundred tokens and predicting the target sequence of similar size accurately is difficult. To overcome this problem, we encode *line numbers* in the program representation. A statement S at a line L in a program is represented by (ℓ, s) where ℓ and s are tokenizations of L and S. A program P with k lines is represented as $(\ell_1, s_1), \ldots, (\ell_k, s_k)$ <eos> where $\ell_1, \ldots, \ell_k$ are the line numbers and $s_1, \ldots, s_k$ are token sequences for statements at the respective lines. We can now train the network to predict a single fix. A fix consists of a line number ℓ_i and an associated statement s'_i that fixes errors in the statement s_i. This output is much smaller compared to the entire sequence representing the fixed program and might be easier to predict. We discuss shortly how to reconcile the fix with the input program and how to fix multiple errors in a program.

Neural Network Architecture

Bahdanau, Cho, and Bengio (2014) introduced an attention mechanism on top of the sequence-to-sequence model of (Sutskever, Vinyals, and Le 2014). Their network consists of an encoder RNN to process the input sequence and a decoder RNN with attention to generate the output sequence. Our network is based on the multi-layered variant in (Vinyals et al. 2015). We briefly describe it below.

Both the encoder and decoder RNNs consist of N stacked gated recurrent units (GRUs) (Cho et al. 2014). The encoder maps each token in the input sequence to a real vector called the *annotation*. For an input sequence $x_1, \ldots, x_{T_x}$, the hidden unit activation at time t is computed as follows:

$$h_t^{(1)} = \text{GRU}\left(h_{t-1}^{(1)}, x_t\right)$$

$$h_t^{(n)} = \text{GRU}\left(h_{t-1}^{(n)}, h_t^{(n-1)}\right), \forall n \in \{2, \ldots, N\}$$

The hidden states of the decoder network are initialized with the final states of the encoder network and then updated as follows:

$$d_t^{(n)} = \text{GRU}\left(d_{t-1}^{(n)}, d_t^{(n-1)}\right), \forall n \in \{2, \ldots, N\}$$

$$d_t^{(1)} = \text{GRU}\left(d_{t-1}^{(1)}, z_t\right)$$

where z_t is the concatenation of the output $\hat{y}_{t-1}$ at time step $t-1$ and the context vector c_t defined as follows:

$$c_t = \sum_{j=1}^{T_x} a_{tj} h_j^{(N)}$$

$$a_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T_x} \exp(e_{tk})}$$

$$e_{tk} = \phi\left(d_{t-1}, h_k^{(N)}\right)$$

The context vector c_t is computed as a weighted sum of the input sequence annotations $h_1^{(N)}, \ldots, h_{T_x}^{(N)}$ using the normalized weights a_{tj} for $j \in \{1, \ldots, T_x\}$. The associated energy values e_{tj} are learned using a soft alignment

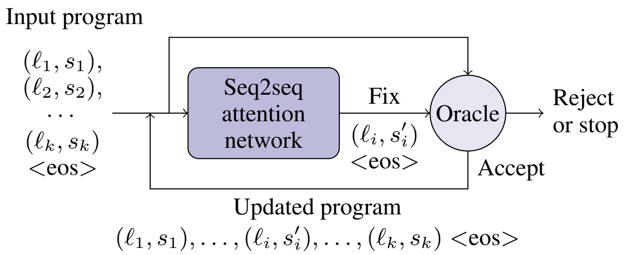


Figure 2: The iterative repair strategy of DeepFix.

model ϕ parameterized as a feed-forward network which is jointly trained with the other components of the model. Intuitively, these weights or energy values determine the importance of the annotations with respect to the hidden state d_{t-1} for computing the current decoder hidden state d_t.

Finally, the decoder output $d_t^{(N)}$ is concatenated with c_t and the result is passed through an affine transformation layer followed by a softmax layer to predict the most probable output token $\hat{y_t}$. We calculate the cross-entropy over the softmax layer outputs at each time step and sum them over the output sequence to compute the loss function.

Iterative Repair

As discussed earlier, in order to keep the prediction task simple, we decided that the network can predict only a single fix. However, a program may have multiple errors. DeepFix uses a simple yet effective iterative strategy to fix multiple errors in a program as shown in Figure 2.

Given the tokenized representation of an input program $(\ell_1, s_1), \ldots, (\ell_k, s_k)$ <eos>, the network predicts a fix, say (ℓ_i, s'_i) <eos>. An oracle takes this fix and the input program, and reconciles them to create the updated program. The updated program is obtained by replacing s_i at line ℓ_i with s'_i. The job of the oracle is to decide whether to accept the fix or not, by checking whether the updated program is better than the input program. In our case, we use a compiler and accept a fix if the updated program does not result in more error messages than the input program. We also use some heuristics to prevent arbitrary changes to the input program. For example, the oracle rejects a fix s'_i if it does not preserve the identifiers and keywords present in the original statement s_i. Once a fix is accepted, DeepFix presents the updated program again to the network as shown in Figure 2. This iterative strategy stops when either 1) the oracle determines that the updated program does not have any errors left to be fixed, or 2) the network deems the input program to be correct and emits a special token “fixed”, or 3) the oracle rejects the fix, or 4) a predefined upper bound on the number of iterations is reached.

Apart from deciding to replace a statement at a line, the network may determine that a new line is to be inserted before (or after) a line l_i. In that case, it emits l_i^- (or l_i^+) instead of l_i. To delete a line l_i, it emits l_i with an empty string ϵ. The network operates on a tokenized representation of a program. Since we must ultimately fix actual programs, the oracle reconstructs the program text from the token sequence obtained after applying the fix. It uses the program-specific encoding map constructed during tokenization to back-substitute the original identifiers. It uses the line number in the fix and replaces the special tokens like NUM and STR by literals at the corresponding line in the input program. If the oracle cannot reconstruct the program text then it rejects the fix.

Dataset statistics				Results	
Dataset	Erroneous programs	Avg. tokens	Error msgs.	Completely fixed programs	Msgs. resolved
Raw	6971	203	16743	1881 (27%)	5366 (32%)
Seeded	9230	206	31783	5185 (56%)	19962 (63%)

Table 1: Summary of the datasets and results.

The proposed repair strategy has several advantages. 1) The program is presented in its entirety to the network. Identifying and fixing programming errors typically requires global analysis capable of inferring long term dependencies. The network architecture, being capable of attending selectively to any part of the program, can reason about the structural and syntactic constraints to predict an erroneous location and required fix. 2) The inclusion of line numbers in both input and output reduces the granularity and hence, the complexity of the prediction task. 3) DeepFix can iteratively fix multiple errors in a program. 4) The oracle is used to track progress and prevent unhelpful or arbitrary changes. 5) The repair strategy of DeepFix is quite general. For example, if we are to attempt fixes for logical errors, we can use a test engine with a test suite as an oracle. A fix would be accepted if it results in a program that passes more tests.

Experiments

Experimental Setup

For training and evaluation, we used C programs written by students for 93 different programming tasks in an introductory programming course. The programs were captured through a web-based tutoring system (Das et al. 2016). We train the neural networks on an Intel(R) Xeon(R) E5-2640 v3 16-core machine clocked at 2.60GHz with 125GB of RAM and equipped with an NVIDIA Tesla K40 GPU accelerator. To work within this machine configuration, we selected programs whose token length ranged from 100 to 400.

Dataset We have two classes of programs in our dataset – programs which compile (henceforth, *correct programs*), and programs which do not compile (henceforth, *erroneous programs*). A student may submit several erroneous programs. We randomly select only one erroneous program per student for each programming task in order to avoid biasing the test results.

In order to give an accurate evaluation of our technique, we do a 5-fold cross validation by holding out roughly 1/5th of the programming tasks for each fold. The erroneous programs belonging to the held out tasks are used for generating the *raw* dataset (Table 1). The correct programs from the rest of the tasks are used for generating training examples. We studied the erroneous programs in our dataset to design mutations required for seeding common programming errors into the correct programs.

We mutate up to 5 statements from each correct program to introduce errors. The mutated program is paired with the fix for the first erroneous line (in the order of increasing line numbers) and this pair constitutes a single *training example*. We then apply this fix to the first erroneous line and continue to recursively generate mutant-fix pairs until all the mutated lines have been fixed. Additionally, we also train the network to identify correct programs by training it to emit a special token "fixed" in response to unedited correct programs. Thus, if the network identifies a program as being correct, it emits this special token to stop the DeepFix iteration loop. To avoid biasing the network to any of the tasks, we restrict the training data generation process to consider only 500 correct programs for each task.

Training We use the attention based sequence-to-sequence architecture implemented in Tensorflow (Abadi et al. 2015). Both the encoder and the decoder in our network have 4 stacked GRU layers with 300 cells in each layer. We use dropout (Srivastava et al. 2014) at a rate of 0.2 on the non-recurrent connections (Pham et al. 2014). The initial weights are drawn from the distribution $\mathcal{U}(-0.07, 0.07)$ and biases are initialized to 1.0. Our vocabulary has 129 unique tokens, each of which is embedded into a 50-dimensional vector. The tokenized representation of an input program is fed to the network in the reverse direction, as in (Sutskever, Vinyals, and Le 2014; Vinyals et al. 2015). The network is trained using the Adam optimizer (Kingma and Ba 2015) with the learning and the decay rates set to their default values and a mini-batch size of 128. We clip the gradients to keep them within the range $[-1, 1]$ and train the network for up to 20 epochs. Finally, we select the trained model with peak validation performance. We train two networks, one for fixing undeclared variables and another to fix all other errors. Since the exact identifier is not relevant for the latter, they are represented by a special token 'ID'. This optimization reduces the complexity of the fix prediction task. In our experiments, we found that it improves the network validation accuracy by up to 5%. We run the two networks simultaneously and for each iteration, show the combined results of both. We provide the source code of the tool online at http://iisc-seal.net/deepfix.

Results

We evaluate DeepFix on the raw dataset as follows. For each fold, we consider all erroneous student programs belonging to the held-out tasks in that particular fold and evaluate the trained model on them. We give the summarized results for all programs in the raw dataset below.

Successful fixes DeepFix *completely* fixes 1881 (27%) out of the 6971 erroneous programs in a way that the fixed programs compile without any errors. It also fixes an additional 1338 (19%) programs partially. Originally, all the programs put together had generated 16743 error messages when compiled using GCC. As shown in Table 1, DeepFix resolves 5366 (32%) error messages from these. DeepFix is also efficient. On an average, it takes only a few tens of milliseconds to fix errors in a test program.

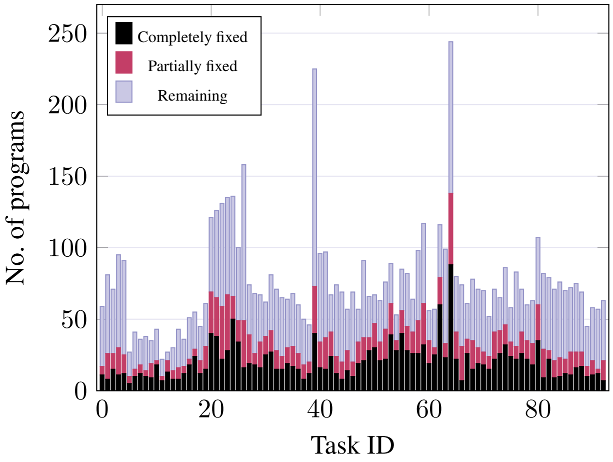


Figure 3: Task-wise reduction in erroneous programs for the raw dataset.

	Error types	Representative causes of errors
e_1	expected declaration or statement	missing statement delimiters
		missing block delimiters
e_2	expected identifier	duplicate block delimiters
e_3	undeclared	extraneous closing parentheses
e_4	expected (misc. tokens)	extraneous commas
		undeclared variables

Table 2: Summary of error types and their causes.

Distribution of fixes by programming tasks The problem statements of the tasks from the programming course are quite diverse, ranging from simple integer arithmetic to sorting and dynamic programming. The programs utilize many language constructs ranging from scalar and array variables to conditionals, nested loops, recursion and functions. Figure 3 shows the distribution of programs before and after fixes for each of the 93 programming tasks. It can be seen that our network generalizes well to programs from all these programming tasks. This gives credence to our claim that *many programming errors are common and not task-specific, and DeepFix can fix them.*

Distribution of fixes by error types Across the entire raw dataset, we observed 71 unique types of compiler error messages. This indicates that errors were of diverse nature. In Figure 4, we show the Top-4 types of error messages and their numbers before and after the fixes for the raw dataset. These form a majority of errors in the dataset. Table 2 describes the top-4 error types and some representative causes for the errors. As can be seen, DeepFix can handle diversity in errors and successfully resolves errors leading to each of these messages.

Effectiveness of iterative repair A student may make multiple errors in a program. The iterative strategy of DeepFix attempts to fix them one-by-one. Figure 5 shows the original errors for the raw dataset marked as iteration 0 and

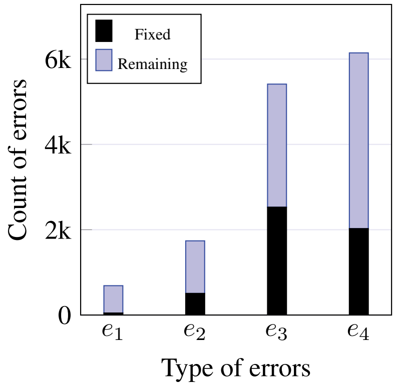


Figure 4: Top-4 types of error messages and their numbers before and after the fixes for the raw dataset.

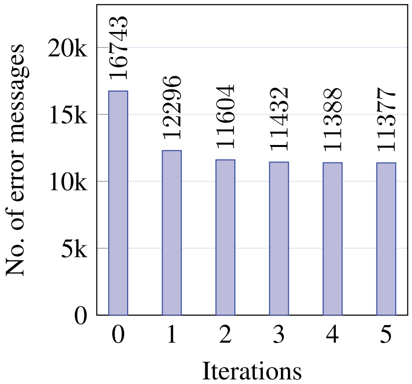


Figure 5: Number of error messages after each iteration on the raw dataset.

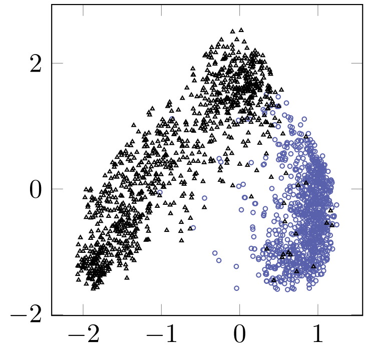


Figure 6: PCA projection of vector-representations of correct (circles) and incorrect (triangle) programs.

then shows the number of error messages remaining at the end of each iteration. We used up to 5 iterations. The first iteration resolves 4447 error messages, while the subsequent iterations resolve an additional 919 messages. If the network fails to produce a fix for a program in an iteration, the subsequent iterations are not executed for that program as applying the same network again will not change the outcome. Thus, the number of programs processed in each iteration reduces and consequently, the number of errors that get fixed also reduces. Nevertheless, many programs get increasingly more correct with each iteration.

Detailed Analysis

We randomly select up to 100 correct submissions from each of the 93 programming tasks and mutate up to 5 statements from each of them to generate a *seeded dataset*. Table 1 gives the results for the seeded dataset. Of the 9230 programs, DeepFix fixes 5185 (56%) programs *completely* and 1534 (17%) programs partially. Out of the 31783 compilation error messages, it resolves 19962 (63%) error messages.

Unlike the raw dataset, we know the exact fixes expected in the seeded dataset. We use this to better understand the performance of DeepFix. In the remaining part of this subsection, we consider the fixes generated by DeepFix on the seeded dataset in the first iteration.

Error localization As discussed before, compilers do not localize errors accurately. Ours is an end-to-end solution in which the network performs error localization on its own. We evaluated the localization accuracy of our network on the seeded dataset. In the first iteration, it is expected to localize the first erroneous line in each program. Of the 9230 erroneous lines, it localizes 7262 (78.68%) successfully. Software error localization tools usually report a ranked list of potentially erroneous lines. With beam search, we can obtain a ranked list. Out of the 9230 erroneous lines, 8077 lines (87.50%) appeared in the top-5 erroneous lines predicted by the network. While we do aim at predicting the required fixes, in practice, even reporting the erroneous lines can itself be of substantial help to programmers.

Fix length (tokens)	Number of fixes	Token-level accuracy	Fix accuracy
< 10	3105	90.01%	72.43%
$10 - 15$	3994	88.21%	66.82%
> 15	2131	80.28%	50.63%

Table 3: Fix length vs fix accuracy for the seeded dataset.

Token-level accuracy versus fix accuracy Program repair is a challenging prediction problem as a proposed fix is correct only if it correctly predicts the required token at every position in the output sequence. Our network had 86.98% token-level accuracy but only 64.97% fix accuracy. The former counts the number of tokens from the output sequence predicted correctly and the latter counts the number of output sequences predicted correctly.

Fix length versus fix accuracy In Table 3, we partition the fixes generated by DeepFix in the first iteration by the length of the fix in number of tokens. As the length of a fix to be predicted increases, both the token-level accuracy and fix accuracy decrease. Nevertheless, even for fixes requiring more than 15 tokens, it has fix accuracy of 50.63%.

Visualizing intermediate representations We feed 2000 programs sampled from the raw dataset into a trained DeepFix encoder and extract the final annotation vectors. We show the first two principal components of these vectors in Figure 6. As can be seen, the correct and incorrect programs form two distinct clusters with very little overlap. This indicates that the DeepFix encoder learns to capture the syntactic validity of programs.

Discussion

Our experiments show that DeepFix fixes many programs and error types successfully. The programs in the raw dataset are written by students in an actual programming course. One major reason limiting the applicability of DeepFix is that fixes are sometimes too complicated for the network to generate. For example, we observed errors such as trying to assign one array variable to another which is not allowed in C. Fixing this will require predicting a loop with element-wise assignments between the arrays, which is beyond the capabilities of the network. We have used a synthetic training dataset. DeepFix fixes substantially more programs in the seeded dataset than the raw dataset (e.g., 56% completely fixed programs for the seeded dataset against 27% for the raw dataset). This indicates that our training dataset does not reflect all possible errors from the raw dataset. We hope that using a better training dataset may increase the performance of DeepFix further. In some cases, there could be multiple options to fix an error; but not all of them may be acceptable. An ad-hoc fix may delete an erroneous line entirely. We use the oracle to prevent DeepFix from accepting such changes. While fix length has an adverse effect on the fix accuracy, we observed that many fixes involve only minor edits to a line. We plan to explore a suitable mechanism in the future to exploit this fact.

Conclusions

We introduce the issue of common programming errors and present an end-to-end solution based on deep learning. Our solution, DeepFix, fixes multiple errors by iteratively invoking a trained neural network. We evaluated DeepFix on 6971 erroneous C programs written by students. DeepFix could fix 27% programs completely and 19% programs partially. Although the evaluation is done only on C programs, our technique is programming language-agnostic and should generalize to other programming languages as well.

In the future, we plan to apply deep learning to fix more challenging programming errors. We also plan to explore ways to improve the performance of DeepFix. In particular, we want to develop neural network architectures that can effectively handle longer sequences.

Acknowledgments

We thank Amey Karkare and Umair Z. Ahmed for providing student programs for experimentation. We also thank the anonymous reviewers for their helpful feedback.

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Arcuri, A. 2008. On the automation of fixing software bugs. In *ICSE Companion*, 1003–1006.

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bhatia, S., and Singh, R. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*.

Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 1724–1734.

Das, R.; Ahmed, U. Z.; Karkare, A.; and Gulwani, S. 2016. Prutor: A system for tutoring CS1 and collecting student programs for analysis. *CoRR* abs/1608.03828.

Debroy, V., and Wong, W. E. 2010. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, 65–74.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *ICSE*, 837–847.

Kingma, D., and Ba, J. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Le Goues, C.; N., T.; Forrest, S.; and Weimer, W. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.* 54 –72.

Long, F., and Rinard, M. 2016. Automatic patch generation by learning correct code. In *POPL*, 298–312.

Monperrus, M. 2015. Automatic software repair: a bibliography. Technical Report #hal-01206501, University of Lille.

Pham, V.; Bluche, T.; Kermorvant, C.; and Louradour, J. 2014. Dropout improves recurrent neural networks for handwriting recognition. In *ICFHR*, 285–290.

Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. J. 2015. Learning program embeddings to propagate feedback on student code. In *ICML*, 1093–1102.

Pu, Y.; Narasimhan, K.; Solar-Lezama, A.; and Barzilay, R. 2016. sk_p: a neural program corrector for moocs. *arXiv preprint arXiv:1607.02902*.

Seo, H.; Sadowski, C.; Elbaum, S.; Aftandilian, E.; and Bowdidge, R. 2014. Programmers' build errors: a case study (at Google). In *ICSE*, 724–734.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15(1):1929–1958.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*, 3104–3112.

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

Vinyals, O.; Kaiser, Ł.; Koo, T.; Petrov, S.; Sutskever, I.; and Hinton, G. 2015. Grammar as a foreign language. In *NIPS*, 2773–2781.

White, M.; Vendome, C.; Linares-Vásquez, M.; and Poshyvanyk, D. 2015. Toward deep learning software repositories. In *MSR*, 334–345.

Xie, Z.; Avati, A.; Arivazhagan, N.; Jurafsky, D.; and Ng, A. Y. 2016. Neural language correction with character-based attention. *arXiv preprint arXiv:1603.09727*.