# Improving Automated Program Repair using Two-layer Tree-based Neural Networks

Yi Li, Shaohua Wang*
SPACE Lab, New Jersey Inst. of Technology, USA
{yl622,davidsw}@njit.edu

Tien N. Nguyen
University of Texas at Dallas, USA
tien.n.nguyen@utdallas.edu

## ABSTRACT

We present DLFix, a two-layer tree-based model learning bug-fixing code changes and their surrounding code context to improve Automated Program Repair (APR). The first layer learns the surrounding code context of a fix and uses it as weights for the second layer that is used to learn the bug-fixing code transformation. Our empirical results on *Defect4J* show that DLFix can fix 30 bugs and its results are comparable and complementary to the best performing pattern-based APR tools. Furthermore, DLFix can fix **2.5 times** more bugs than the best performing deep learning baseline.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

## KEYWORDS

Deep Learning; Automated Program Repair

**ACM Reference Format:**
Yi Li, Shaohua Wang and Tien N. Nguyen. 2020. Improving Automated Program Repair using Two-layer Tree-based Neural Networks. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), October 5–11, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377812.3390896

## 1 INTRODUCTION

Automated Program Repair (APR) techniques are useful to help developers identify and fix software bugs. Recently, various APR tools perform automated repairs using distinct ways, e.g., mining and learning fixing patterns/templates [5], information retrieval [9], or machine learning [6]. Deep learning (DL) has been applied in the recent APR studies. Some DL approaches treat the APR as a neural network machine translation (NMT) problem [10]. However, they still have some major limitations: (1) have a hard time finding the correct fixing location in a statement; (2) treat code using sequence-based representations; and (3) often cannot learn the surrounding code context information (i.e., unchanged code).

To address these challenges, we introduce DLFix, a two-layer tree-based deep learning model to learn code transformations from prior bug fixes to fix a given buggy code. We treat the APR problem as code transformation learning, in which transformations corresponding to bug fixes including (un)-changed parts are encoded as the input for model training. We conducted several experiments to evaluate DLFix on three datasets:*Defects4J, Bugs.jar, and Bigfix.* BigFix contains +20k real-world bugs. Our results show that DLFix can fix 30 bugs and generate comparable and complementary results compared with the top-performing pattern-based APR tools.

We make the following contribution: **A. DL for APR:** DLFix is the first DL APR that generates comparable and complementary results comparing with powerful pattern-based tools on Defects4J. **B. Model:** DLFix has a new two-layer model to do the surrounding code context and transformation learning separately. **C. Empirical Results:** 1) We show DLFix can have comparable and complementary results comparing with the best pattern-based tools. DLFix does not require hard-coding of bug-fixing patterns as in those tools. DLFix is fully automatic and data-driven. 2) DLFix can fix at least 2.5 times more bugs than the best performing DL approach.

## 2 OUR APPROACH

Figure 1 shows our approach consisting five steps.

**Step 1.** We conduct alpha-renaming on the variable names of a project and use Word2Vec [7] to generate vector representations for code tokens in $M_b$ and $M_f$, where $M_b$ is a buggy method and $M_f$ is $M_b$'s fixed version. We generate a vector for the buggy sub-tree $T_b^{sub}$ that represents the buggy statement in $M_b$ and the other vector for the sub-tree $T_f^{sub}$ which represents the fixed statement in $M_f$. To do this, we use a DL-based code summarization model [11] to summarize a sub-tree into a vector for a node (called *a summarized node*) which be used in following steps.

**Step 2.** We build a two-layer tree-based learning model to do the automatic repairing. We use one tree-based RNN as the local **C**ontext **L**earning **L**ayer (CLL) to learn the local context of the code surrounding the bug-fixing changes (i.e., the unchanged code surrounding the changed one).

**Step 3.** The second layer is a code **T**ransformation **L**earning **L**ayer (TLL) for learning the code transformation changes. In this TTL, we used the bug fixing before and after to train the model. And also, we use the context of the transformation computed as the vector in the CLL as the weight in this layer.

**Step 4.** We built some program analysis filters to help improve the model performance.

**Step 5.** We use a Convolutional Neural Network (CNN) [3] based classification model to re-rank all patches.

## 3 EVALUATION

### 3.1 Research Questions

**RQ1.** How well does DLFix perform in comparison with the state-of-the-art pattern-based APR approaches?
**RQ2.** How well does DLFix perform in comparison with the state-of-the-art deep learning-based APR approaches?

*Corresponding Author


*ICSE '20 Companion, October 5–11, 2020, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3390896

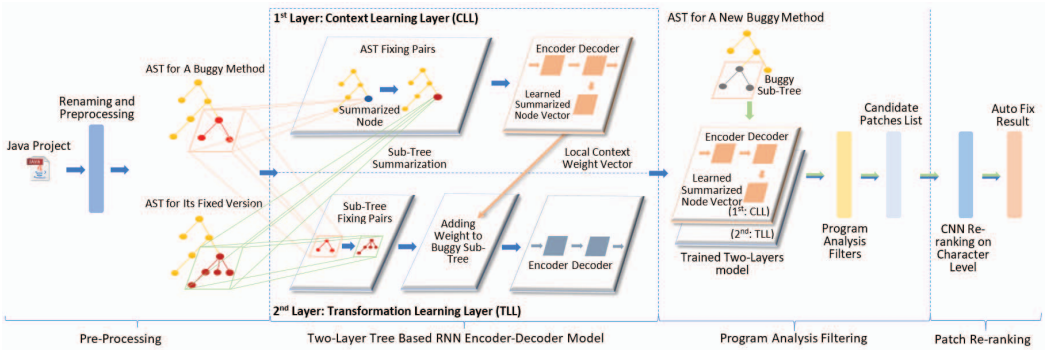


**Figure 1: DLFix: Context-based Transformation Learning for Automated Program Repair**

**Table 1: RQ1. Comparison with the Pattern-based APR Baselines on Defect4J.**

| Project | jGenProg | HDRepair | Nopol | ACS | ELIXIR | ssFix | CapGen | SketchFix | FixMiner | LSRepair | AVATAR | SimFix | TBar | DLFix |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Chart | 0/7 | 0/2 | 1/6 | 2/2 | 4/7 | 3/7 | 4/4 | 6/8 | 5/8 | 3/8 | 5/12 | 4/8 | **9/14** | 5/12 |
| Closure | 0/0 | 0/7 | 0/0 | 0/0 | 0/0 | 2/11 | 0/0 | 3/5 | 5/5 | 0/0 | **8/12** | 6/8 | **8/12** | 6/10 |
| Lang | 0/0 | 2/6 | 3/7 | 3/4 | 8/12 | 5/12 | 5/5 | 3/4 | 2/3 | 8/14 | 5/11 | **9**/13 | 5/**14** | 5/12 |
| Math | 5/18 | 4/7 | 1/21 | 12/16 | 12/19 | 10/26 | 12/16 | 7/8 | 12/14 | 7/14 | 6/13 | 14/26 | **19/36** | 12/28 |
| Mockito | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 1/1 | **2/2** | 0/0 | 1/**2** | 1/1 |
| Time | 0/2 | 0/1 | 0/1 | 1/1 | **2**/3 | 0/**4** | 0/0 | 0/1 | 1/1 | 0/0 | 1/3 | 1/1 | 1/3 | 1/2 |
| Total | 5/27 | 6/23 | 5/35 | 18/23 | 26/41 | 20/60 | 21/25 | 19/26 | 25/31 | 19/37 | 27/53 | 34/56 | **43/81** | 30/65 |
| P(%) | 18.5 | 26.1 | 14.3 | 78.3 | 63.4 | 33.3 | **84.0** | 73.1 | 80.6 | 51.4 | 50.9 | 60.7 | 53.1 | 46.2 |

Note: P is the probability of the generated plausible patches to be correct. In a cell, x/y: x is the number of correct fixes and y is the number of candidate patches that can pass all test cases.

### 3.2 Experimental Methodology

**Dataset.** We evaluate approaches on *Defects4J*, *Bugs.jar*, and *BigFix* [4]. For *BigFix*, we use 90% data for training models and 10% for testing them, while we train all approaches on *BigFix* and test them on *Defects4J* and *Bugs.jar*.

**Analysis Approaches.** In **RQ1**, we use the same fault localization tool Ochiai as SimFix and the test case validation to help find the right patches. In **RQ2**, we assume the buggy locations are known as input and do no test case validation.

### 3.3 Experimental Results

**RQ1.** DLFix can auto-fix 30 Defects4J bugs while the best performed pattern baselines: SimFix and Tbar can fix 34 and 43 bugs. DLFix can fix 11 and 7 unique bugs compared with SimFix and Tbar, respectively. DLFix can have comparable and complementary results with top pattern based approaches.

**Table 2: RQ2. Accuracy Comparison with DL-based APR approaches on three Datasets.**

| Approach | Defect4J | | | Bugs.jar | | | BigFix | | |
|---|---|---|---|---|---|---|---|---|---|
| | Top1 | Top5 | Top10 | Top1 | Top5 | Top10 | Top1 | Top5 | Top10 |
| **Ratchet** | 2.0% | 4.0% | 6.9% | 2.4% | 4.4% | 6.8% | 3.0% | 4.1% | 6.9% |
| **Tufano ('18)** | 6.9% | 9.9% | 11.9% | 8.4% | 11.1% | 12.9% | 7.9% | 10.6% | 12.1% |
| **CODIT** | 8.9% | 13.9% | 15.8% | 7.0% | 11.8% | 14.8% | 6.9% | 13.7% | 18.3% |
| **Tufano ('19)** | 15.8% | 20.8% | 23.8% | 13.5% | 18.7% | 23.1% | 15.4% | 17.3% | 21.4% |
| **DLFix** | **39.6%** | **43.6%** | **48.5%** | **34.2%** | **36.4%** | **37.9%** | **29.4%** | **31.1%** | **33.4%** |

Note: Top K is the number of times that a correct patch is in the ranked list of top K candidate patches over the total number of bugs.

**RQ2**.DLFix can fix 39.6%, 34.2%, and 29.4% bugs in *Defects4J, Bugs.jar, and BigFix* using only top-1 ranked candidates. DLFix can fix at least 2.5 times bugs than the best performing baseline.

## 4 RELATED STUDY

The earlier APR aims to derive similar fixes for similar source code, e.g. [8]. A large group of APR approaches has explored search-based software engineering to tackle more general types of bugs [2]. The fixing patterns or templates could be automatically or semi-automatically mined [5]. Recently, DL also has been applied to APR for directly generating patches [1, 10].

## 5 CONCLUSION

In this poster, we show DLFix, a two-layer DL based automated program repair (APR) approach, to improve and complement the existing state-of-the-art APR approaches. Our results show that DLFix can have comparable and complementary results compared with pattern based approaches and can fix at least 2.5 times more bugs than other DL based approaches.

## ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

## REFERENCES

[1] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. CODIT: Code Editing with Tree-Based Neural Machine Translation. *arXiv preprint arXiv:1810.00314* (2018).
[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. In *TSE 38, 1.* 54–72.
[3] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).
[4] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 162 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360588
[5] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 31–42.
[6] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *FSE.* New York, NY, USA, 727–739.
[7] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546
[8] Baishakhi Ray and Miryung Kim. 2012. A Case Study of Cross-System Porting in Forked Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE).* Association for Computing Machinery, New York, NY, USA, Article Article 53, 11 pages.
[9] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *ASE.* 648–659.
[10] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes Via Neural Machine Translation. In *ICSE.* 25–36.
[11] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *ASE (ASE 2018).* 397–407.