# Learning to Fix Build Errors with Graph2Diff Neural Networks

**ACM Reference Format:**
. 2019. Learning to Fix Build Errors with Graph2Diff Neural Networks. In . ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/1122445.1122456

## 1 INTRODUCTION

Professional software developers spend a significant amount of time fixing builds; for example, one large-scale study found that developers build their code 7–10 times per day [12], with a significant number of builds being unsuccessful. Build errors include simple errors such as syntax errors, but for professional developers these are a small minority of errors; instead, the majority are linking errors such as unresolved symbols, type errors, and incorrect build dependencies [12]. A recent paper by Google reports roughly 10 developer-months of effort are spent every month fixing small build errors [10]. Therefore, automatically repairing build errors is a research problem that has potential to ease a frequent pain point in the developer workflow. Happily, there are good reasons to think that automatic build repair is feasible: fixes are often short, and we can test a proposed fix before showing it to a developer simply by rebuilding the project. Build repair is thus a potential "sweet spot" for automatic program repair, hard enough to require new research ideas, but still just within reach.

However, there is only a small literature on repairing build errors. Previous work has successfully repaired syntax errors like missing delimiters and parentheses [4, 5], but in our corpus of professional build errors (Table 1), fixes are more subtle, often requiring detailed information about the project APIs and dependencies. Recently, the DeepDelta system [10] aimed to repair build errors by applying neural machine translation (NMT), translating the text of the diagnostic message to a description of the repair. Although this work is promising, the use of an off-the-shelf NMT system severely limits the types of build errors that it can handle.

We introduce a new deep learning architecture, called *Graph2Diff networks*, specifically for the problem of predicting edits to source code, as a replacement for the celebrated sequence-to-sequence model used for machine translation. Graph2Diff networks map a *graph* representation of the broken code to a *diff*[1] in a domain-specific language that describes the repair. The diff can contain not only tokens, but also pointers into the input graph (such as "insert token HERE") and copy instructions (i.e. "copy a token from HERE in the input graph"). Thus, Graph2Diff networks combine, extend, and generalize a number of recent ideas from neural network models for source code [1, 2, 14]. We have obtained promising results on applying Graph2Diff networks for automatically repairing build errors, and more broadly, there is exciting potential that a large number of tasks across software engineering can be profitably cast as Graph2Diff networks.

[1]We slightly abuse terminology here and use "diff" to mean a sequence of edit operations that can be applied to the broken AST to obtain the fixed AST.


*ICSE '20, May 23-29, 2020, Seoul, South Korea*

ACM ISBN 0000…$15.00
https://doi.org/10.1145/1122445.1122456

## 2 METHOD

We formulate the problem of resolving a set of build diagnostics as a supervised machine learning problem. The input is the state of the source tree at the time of a broken build and a list of diagnostics returned by the compiler. The target output is a diff that can be applied to the code to resolve all of the diagnostics. For our purposes, a diff is a sequence of transformations to apply to the original source code to generate the repaired version. Compiler diagnostics do not always identify the source of the fault that needs to be repaired, so we require the models to predict the locations that need changing in addition to the repairs. This combines the well-studied problems of automated fault localization and automated program repair.

We collected a large dataset of 500k fixes by taking advantage of a highly-instrumented development process similar to that at Google, extending previous work by [10].[2] Examples from the dataset appear in Figure 1. An example edit script that Graph2Diff networks are trained to output appears in Figure 2.

At a high level, Graph2Diff networks learn to map the broken version of the code, represented as a graph, to an edit script that represents the fix. Graph2Diff is based on three key architectural ideas from deep learning: First, *graph neural networks* [8, 11] can explicitly encode syntactic structure, semantic information, and even information from program analysis in a form that neural networks can understand, allowing the network to learn to change one part of the code based on its relationship to another part of the code. For us, the graph represents the input to the build repair task, including the broken abstract syntax tree (AST) and the build diagnostics. Second, *pointer models* [13] can generate locations in the initial AST to be edited, which leads to a compact way of generating changes to large files (as diffs). Much work on program repair divides the problem into two separate steps of fault localization and generating the repair; unfortunately, fault localization is a difficult problem [9]. Using pointers, the machine learning component can predict both where and how to fix. Finally, the *copy mechanism* addresses the well-known *out-of-vocabulary* problem of source code [3, 6, 7]: source code projects often include project-specific identifiers that do not occur in the training set of a model. A copy mechanism can learn to copy any needed identifiers from the broken source code, even if they do not occur in the training set.

## 3 RESULTS

We follow standard machine learning experimental details, using train, validation and test splits and grid search for choosing hyperparameters. Our main metric is sequence-level accuracy, which

[2]Further details omitted for anonymity.

| Row | Diagnostics | Fix |
|---|---|---|
| A | cannot find symbol 'widgetSet()' | `- widgetsForX.widgetSet().stream().forEach(`<br>`+ widgetCounts.widgetSet().stream().forEach(` |
| B | cannot find symbol 'of' | `- Framework.createWidget(new FooModule.of(this)).add(this);`<br>`+ Framework.createWidget(FooModule.of(this)).add(this);` |
| C | (line 10) cannot find symbol 'longName'<br>(line 15) cannot find symbol 'longName' | `- String longname = "reallylongstringabcdefghijklmnopqrstuvw..."`<br>`+ String longName = "reallylongstringabcdefghijklmnopqrstuvw..."`<br>`x = f(longName) // (line 10) Diagnostic pointed here`<br>`// ...`<br>`y = g(a, b, c, longName) // (line 15) Diagnostic pointed here` |
| D | incompatible types: ParamsBuilder cannot be converted to Optional<Params> | `- return new ParamsBuilder(args);`<br>`+ return new ParamsBuilder(args).build();` |
| E | incompatible types: RpcFuture<LongNameResponse> cannot be converted to LongNameResponse | `- LongNameResponse produceLongNameResponseFromX(`<br>`+ ListenableFuture<LongNameResponse> produceLongNameResponseFromX(` |
| F | incompatible types: WidgetUnit cannot be converted to Long | `- public Widget setDefaultWidgetUnit(WidgetUnit defaultUnit) {`<br>`+ public Widget setDefaultWidgetUnit(Long defaultUnit) {`<br>`this.defaultUnit = defaultUnit; // Diagnostic pointed here` |
| G | cannot find symbol 'of(Widget,Widget)' | `- import com.google.common.collect.ImmutableCollection;`<br>`+ import com.google.common.collect.ImmutableSet;`<br>`// ...`<br>`- ImmutableCollection.of(`<br>`+ ImmutableSet.of(` |

**Figure 1: Example diagnostics and fixes from the dataset.**

```
- LongNameResponse produceLongNameResponseFromX(
+ ListenableFuture<LongNameResponse> produceLongNameResponseFromX(
```

```
0: MethodDef
1:   Type
2:     LongNameResponse
3:   Id
4:     produceLongNameResponseFormX
5:   Args
```

```
0:INSERT 1:INPUT_POINTER(1) 2:INPUT_POINTER(2) 3:PARAMETERIZED_TYPE 4:TYPEAPPLY
5:INSERT 6:OUTPUT_POINTER(0) 7:FIRST_CHILD 8:IDENTIFIER 9:ListenableFuture
10:INSERT 11:OUTPUT_POINTER(0) 12:OUTPUT_POINTER(5) 13:IDENTIFIER 14:LongNameResponse
15:DELETE 16:INPUT_POINTER(2)
17:DONE
```

**Figure 2: An example edit script that makes the change specified in Figure 1E. (Top) a textual diff of the change, (Left) A subset of the original AST annotated with input ids. (Right) An edit script implementing the change, annotated with output ids.**

is how often the model predicts the full sequence correctly. This is a strict metric that only gives credit for exactly matching the developer's change.

We compare to the most closely related work, which is DeepDelta [10]. Even though DeepDelta can only be evaluated on a less-stringent task than exact developer fix match (because it has no pointer mechanism to precisely specify locations), we find that Graph2Diff networks achieve 26% accuracy versus DeepDelta achieving 10% accuracy when predicting less precise diffs. We interpret this result as showing promise about the potential of Graph2Diff networks for repairing broken builds, and also for code-editing tasks including other forms of program repair and beyond.

## REFERENCES


[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *ICLR*.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[3] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 207–216.

[4] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In *Working Conference on Mining Software Repositories (MSR)*. 252–261.

[5] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[6] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 763–773. https://doi.org/10.1145/3106237.3106290

[7] Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. *CoRR* abs/1903.05734 (2019). arXiv:1903.05734 http://arxiv.org/abs/1903.05734

[8] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. In *ICLR*.

[9] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 102–113.

[10] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning To Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA.

[11] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* (2009).

[12] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at Google). In *International Conference on Software Engineering*. ACM, 724–734.

[13] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 2692–2700. http://papers.nips.cc/paper/5866-pointer-networks.pdf

[14] Rui Zhao, David Bieber, Kevin Swersky, and Daniel Tarlow. 2019. Neural Networks for Modeling Source Code Edits. *arXiv preprint arXiv:1904.02818* (2019).