DeepDelta: Learning to Repair Compilation Errors

Ali Mesbah[∞], Andrew Rice[◇],
Edward Aftandilian[♣], Emily Johnston[♣], and Nick Glorioso[♣*]
[∞]University of British Columbia, [◇]University of Cambridge
, [♣]Google
amesbah@ece.ubc.ca,acr31@cam.ac.uk
{eaftan,epmjohnston,glorioso}@google.com

ABSTRACT

Programmers spend a substantial amount of time manually repairing code that does not compile. We observe that the repairs for any particular error class typically follow a pattern and are highly mechanical. We propose a novel approach that automatically learns these patterns with a deep neural network and suggests program repairs for the most costly classes of build-time compilation failures. We describe how we collect all build errors and the human-authored, in-progress code changes that cause those failing builds to transition to successful builds at Google. We generate an AST diff from the textual code changes and transform it into a domain-specific language called Delta that encodes the change that must be made to make the code compile. We then feed the compiler diagnostic information (as source) and the Delta changes that resolved the diagnostic (as target) into a Neural Machine Translation network for training. For the two most prevalent and costly classes of Java compilation errors, namely missing symbols and mismatched method signatures, our system called DeepDelta, generates the correct repair changes for 19,314 out of 38,788 (50%) of unseen compilation errors. The correct changes are in the top three suggested fixes 86% of the time on average.

ACM Reference Format:
Ali Mesbah[∞], Andrew Rice[◇],, Edward Aftandilian[♣], Emily Johnston[♣], and Nick Glorioso[♣]. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

One of the benefits of using a compiled programming language is that programming mistakes can emerge at compilation time rather than when the program is executed. A failed build will often prompt an edit-compile cycle in which a developer iterates between attempting to resolve diagnostic errors and rerunning the compiler.

A previous large-scale study reported that professional developers build their code 7–10 times per day on average [43]. The study found that build-time compilation errors are prevalent and cost developers substantial time and effort to resolve. Build-time compilation errors[1] emerge when a developer compiles her code through a build management system such as Bazel, Gradle, or Maven.

Our goal is to help developers to repair build errors automatically. We propose a novel approach, called DeepDelta, for automated repair of build-time compilation errors. Our insight is that there exist common patterns in the way developers change their code in response to compiler errors. Such patterns can be learned automatically by extracting Abstract Syntax Tree (AST) changes between the failed and resolved snapshots of the code and feeding these as abstracted features to a deep neural network.

This paper makes the following contributions:

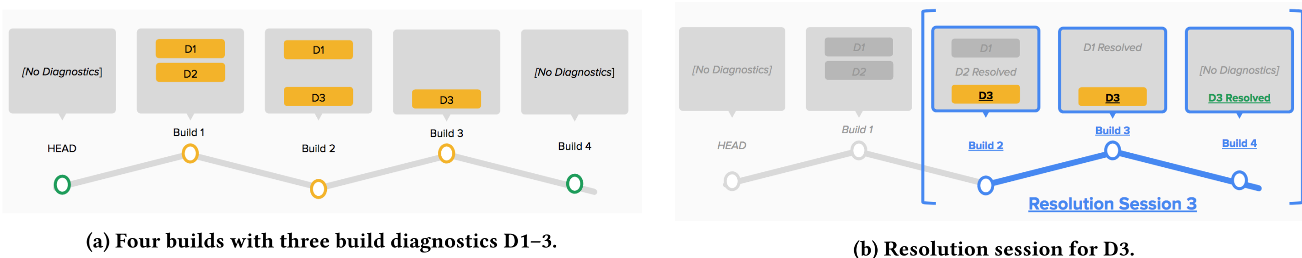
- We perform a large-scale study of compilation errors and changes that resolve them to find the most prevalent and costly error kinds in practice. Our dataset is collected from 10,000 Java projects containing 300 million LOC at Google. Our study shows that 51% of all compiler diagnostics are related to the compiler not being able to resolve a particular symbol, which is also the most costly category of compiler errors to fix.
- We formulate automated repair as a *Neural Machine Translation* (NMT) [47] problem in which the source contains information about the failure and the target is the set of AST changes, captured in a domain-specific language, called Delta, which resolves the failure.
- We present the instantiation of our approach, called DeepDelta, which automatically generates source and target features from previous developer data and learns repair patterns for the two most prevalent and costly classes of Java compilation errors in practice.
- We show that our technique is effective through a large-scale empirical evaluation on 38,788 unseen compilation errors at Google. Our results show that DeepDelta can generate the exact correct repair between 47%–50% of the time. Of these cases, the correct fixes are in the top three suggested fixes 85%–87% of the time.

Previous research in the area of automated program repair has focused on finding patches when a test failure occurs through fixed templates and search-based techniques [16, 25, 29, 30]. A recent related work [18] deploys deep learning and achieves an accuracy of 27% on fixing syntax errors in C. This accuracy is obtained on examples of students completing 93 different programming tasks, which means numerous implementations of the same program. In comparison, we learn from much more diverse and real developer

*This work took place while Ali Mesbah was a Visiting Researcher at Google.

[1]In this paper, we use build errors and compilation errors interchangeability to mean build-time compilation errors.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Four builds with three build diagnostics D1–3.

(b) Resolution session for D3.

Figure 1: Build Resolution Sessions.

code changes and achieve a 85% improvement (or 23 percent point improvement). Our work focuses on learning AST changes, rather than whole program code, which we believe is novel and gives us significantly better results as shown by our evaluation results.

2 COLLECTING COMPILATION ERRORS

The first step required for learning repair patterns is obtaining developer data on compilation errors. We collect this data as part of the regular development cycle at Google. We also use this dataset to characterize the prevalence and cost of different classes of compilation errors in practice.

2.1 Data Collection

Every build initiated by a developer is automatically logged at Google. The log contains detailed information about each build, including any compiler diagnostics, i.e., detailed error messages, along with a snapshot of the code that was built.

For this study, we collected textual build logs for a period of two months, from January 23, 2018, to March 23, 2018. The collected logs were subsequently parsed and analyzed to understand which build errors happen most frequently in practice. Although our build-diagnostics framework is language-agnostic, in this paper, we focus on build errors pertaining to Java projects.

2.2 Diagnostic Kinds

We group compiler error messages by *diagnostic kind*. A diagnostic kind represents a class of errors that all have the same cause. Compilers have error message templates into which concrete names are interpolated. For example, javac uses the template "{0} is abstract; cannot be instantiated" for an attempt to instantiate an abstract class and it refers to it by the key `abstract.cant.be.instantiated`. We built a parser to map concrete error messages in the build log back to the message templates that produced them (see Table 2).

2.3 From Diagnostics to Resolutions

A failed build can contain numerous diagnostics; each of these diagnostics might be new or might be the same as one reported from a previous build. We therefore first set out to convert sequences of builds containing a particular diagnostic into *resolution sessions*.

Definition 1 (Resolution Session (RS)). *A resolution session (RS) for a build diagnostic D_i is a sequence of two or more consecutive builds, $B_1, B_2, \ldots, B_k$ where D_i is first introduced in B_1 and first resolved in B_k, and the time between build B_n and B_{n-1} is no more than a given time window T.*

The intention with a resolution session is to capture the period of time that the developer is actively working on resolving the diagnostic. Therefore, we define the time window T to be one hour. This window represents a "task switch window," i.e., if a developer has not performed a build within T, it is likely they have switched to some other activity; e.g., they could have gone to a meeting, lunch, or left the office.

We quantify the developer cost in terms of time needed to resolve a particular diagnostic. Consider a diagnostic D_i with a resolution session $B_1, B_2, \ldots, B_k$. Let $|D_i|$ be the number of diagnostics produced by build B_i; we call this the *active diagnostics* at B_i. Let Ts_i be the start time that build B_i was initiated, and Te_i be the end time that build B_i completed. Then we define active resolution cost as:

Definition 2 (Active Resolution Cost (ARC)). *For a diagnostic D_i, Active Resolution Cost (ARC) represents the active time the developer spends resolving D_i, excluding the cost of the builds themselves, divided by the number of diagnostics present in the intermediate builds of its resolution session:*

$$\sum_{i=1}^{k-1} \frac{Ts_{i+1} - Te_i}{|D_i|}$$

Figure 1 depicts an example of how we construct resolution sessions. As shown in fig. 1a, $D1$ and $D2$ are first introduced in build 1, and $D3$ first appears in build 2. We consider a diagnostic to be resolved when it disappears from the build (see Definition 1). For instance, $D3$ disappears in build 4, and thus its resolution session includes builds 2–4, as shown in Figure 1b.

Table 1: Dataset

Builds	4.8 million
Failed Builds	1.0 million
Compiler Diagnostics	3.3 million
Resolution Sessions	1.9 million
Green & Singular Sessions	110,219

2.4 Dataset and Findings

Table 1 summarizes our dataset. We processed a total of 4.8 million builds of which 1.0 million were failures, i.e., around 20% of all builds fail.

Table 2: Top 10 diagnostic kinds by Active Resolution Cost (ARC).

Diagnostic Kind	ARC (s)			Builds in session			Resolved diagnostic		ARC	Description
compiler.err	Avg.	Min	Max	Avg.	Min	Max	Instances	%	%	
cant.resolve	301	0	14,828	2.6	2	143	949,325	51	47	Use of undefined symbol
cant.apply.symbol	351	0	11,064	2.6	2	101	151,997	8	11	No method declaration found with matching signature ({0} {1} in {4} {5} cannot be applied to given types)
strict	218	0	11,776	2.2	2	72	109,156	6	9	Incorrectly declared dependencies
doesnt.exist	348	0	13,244	2.7	2	70	159,158	9	7	Use of undefined package
cant.apply.symbols	320	0	8,859	2.5	2	41	60,287	3	5	No method declaration found with matching signature (No suitable {0} found for {1}({2}))
expected	188	0	6,542	2.5	2	72	168,299	9	4	Syntax error
inconvertible.types	277	0	6,907	2.5	2	42	38,191	2	3	Cast between inconvertible types
unreported.exception	207	0	4,917	2.3	2	32	22,684	1	2	Code may throw checked exception, which must be handled
already.defined	206	0	6,423	2.4	2	24	12,381	1	1	Symbol already defined
does.not.override.abstract	458	0	10,015	2.8	2	43	8,089	0.4	1	No implementation for inherited abstract method

Recall that a build failure can contain multiple compilation errors, i.e., diagnostics. These build failures contained a total of 3.3 million diagnostics from which we were able to find 1.9 million resolution sessions. The remaining 1.5 million diagnostics for which we found no resolution session correspond to those changes abandoned by developers or with more than one hour between build attempts.

As a final step we identified 110,219 resolution sessions which contained only a single diagnostic (*singular*) and which ended in a successful build (*green*). We use these singular, green resolution sessions as training data since we can be sure that the change made by the developer actually resolved the diagnostic in question. Our dataset excludes automated batch builds since we are only interested in interactive activity by developers.

Table 2 presents the top-ten most frequent and costly build errors in our dataset. The table shows the diagnostic kind, active resolution cost (average, min and max), the number of subsequent builds to resolve the diagnostic (average, min and max), the number and percentage of instances of each diagnostic kind, the relative active resolution cost with respect to the total, and a textual description of the diagnostic kind.

In total, there were 1,853,417 compiler diagnostics that were later fixed within a resolution session. As the table shows, 51% (949,325) of those diagnostics are related to the compiler not being able to resolve a particular symbol, i.e., the `cant.resolve` diagnostic kind with a "cannot find symbol" message. Our results also confirm a previous study conducted in 2014, which showed 43% of build errors are caused by issues related to `cant.resolve` [43]. Compared to the findings in 2014, it seems the issues developers have with missing symbols have only been exacerbated. The next diagnostic kind in the table in terms of instances is `cant.apply.symbol` with 8% of total diagnostics. `cant.apply.symbol` happens when the compiler cannot find a method declaration with the given types.

We also calculated the relative cost of build errors by multiplying the number of build-diagnostic instances by the average active resolution cost needed to resolve the diagnostic, for each diagnostic kind. The total cost amounts to 57,215,441 seconds for two months of data. This means within two months, developers spent approximately 21 months fixing build errors. From this total, `cant.resolve` is again the most costly diagnostic kind by far, with 47% ($\approx$ 10 months) of the total active resolution cost. `cant.apply.symbol` acounts for 11% ($\approx$ 2 months) of the total active resolution cost. These top-two error classes alone account for 58% of the total cost, which is approximately a year of developer cost in our dataset.

3 RUNNING EXAMPLE

Given the high prevalence and cost of `cant.resolve` and `cant.apply.symbol` in practice, we focus on these two categories of build errors to generate repair suggestions in this work. Note, however, that our approach is generic enough to be applied to any of these diagnostic kinds with minor adaptations. We use the `cant.resolve` kind as a running example in our paper. An identifier must be known to the compiler before it can be used. An inconsistency between the definition of an identifier and its usage, including when the definition cannot be found, is the root cause of this build error. This occurs when there is, for instance, a missing dependency (e.g., on another library), a missing import, or a mistyped symbol name.

As a motivating example, consider the following code snippet:

```
import java.util.List;
class Service {
  List<String> names() {
    return ImmutableList.of("pub", "sub");
  }
}
```

When this code is built, the compiler produces the following error message:

```
Service.java:4: error: cannot find symbol
  symbol:   variable ImmutableList
```

In this case, the developer has forgotten to import the package for `ImmutableList` (from the Google Guava library) and so the compiler cannot resolve the symbol. To fix the problem, the developer determines the correct package for the missing symbol and adds an appropriate import statement:

```
import java.util.List;
+++ import com.google.common.collect.ImmutableList;
class Service {
  List<String> names() {
    return ImmutableList.of("pub", "sub");
  }
}
```

Because `ImmutableList` is defined in a different project, one must also declare the dependency to the build system. Using the Bazel build system, the fix might be to delete the existing reference to Guava's base package and add its collect package instead:

```
java_library(
    name = "Service",
    srcs = [
      "Service.java",
    ],
    deps = [
      --- "//java/com/google/common/base",
      +++ "//java/com/google/common/collect",
    ],
    ...
)
```

4 FINDING RESOLUTION CHANGES

Once build diagnostics are collected and resolution sessions are constructed, we pass the resolution sessions to the resolution change detection step of our pipeline. The goal in this step is to systematically examine how developers change their source code to resolve build errors.

4.1 Retrieving Code Snapshots

At Google, changes made to the source code in developers' IDE clients are automatically saved in the cloud as temporal snapshots. This means a complete history of all saved changes made to the code is preserved with retrievable snapshot identifiers. This feature allows us to go backward in time and retrieve a snapshot of the code in the state it was in at the time of a particular build.

For every build resolution session computed, our approach first extracts the snapshot IDs in the first and the last builds. The first ID corresponds with the code snapshot that caused the diagnostic. The second ID points to the code snapshot in which the diagnostic was resolved.

Using each snapshot ID, we then query the snapshot cloud server to obtain the code exactly as it was at that particular point in time.

4.2 AST Differencing

At this point, we have at our disposal two snapshots of the code at the broken and fixed states, for each resolution session. To understand how developers change their code to resolve build errors, we compute the differences going from the broken state to the fixed one.

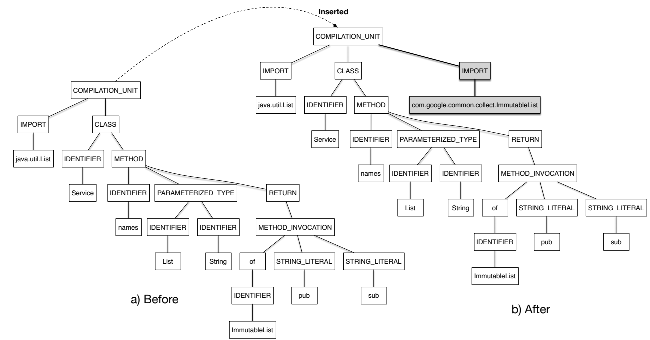


Figure 2: Java AST Changes.

The conventional method for detecting source code changes is the Unix line diff [23], which computes changes at the textual granularity of only line-level add and delete actions. This leads to a diff which is largely dependent on how the source code is formatted. While line diff is a popular method for human consumption, e.g., during code review, automatically inferring syntactic changes to the code from textual line diffs is difficult.

To analyze code changes at the syntactic level, we take a tree differencing approach [13, 15]. We parse the broken and fixed snapshots of each resolution session to generate the corresponding abstract syntax trees (ASTs). We have created parsers for Java and the Bazel BUILD language in this project, although support for other languages can easily be added.

Then the ASTs of the broken and fixed snapshots are passed into a tree differencing algorithm.

Definition 3 (AST Diff). *Given two ASTs, source ast_s and target ast_t, an AST Diff is a set of vertex change actions that transforms ast_s into ast_t.*

The AST differencing algorithm first tries to map each vertex on the source AST ast_s to a vertex on the target AST ast_t, by comparing the vertex labels. If any unmatched vertices are detected, it computes a short sequence of edit actions capable of transforming ast_s into ast_t. Finding the shortest edit action is NP-hard; therefore, heuristics are used to compute a short transformation from ast_s to ast_t deterministically [13]. The final output of the tree differencing step is composed of a set of change actions that indicate moved, updated, deleted, or inserted vertices on the source and target ASTs:

- Moved: an existing vertex (and its children) in ast_s is moved to another location in ast_t.
- Updated: the old value of a vertex in ast_s is updated to a new value in ast_t.
- Deleted: a vertex (and its children) in ast_s is removed in ast_t.
- Inserted: a vertex that is non-existent in ast_s is added in ast_t.

Figure 2 visualizes the AST of our motivating example (see Section 3) in the initial broken state (a) and the fixed AST (b) after the developer added the import in the Java code.

The changed vertices detected by the AST diff are indicated in grey in Figure 2. The AST diff for the Java code in the running example detects that there is an `IMPORT` vertex inserted into the root vertex, `COMPILATION_UNIT`. The fully-qualified package name of `ImmutableList` is also inserted as a child vertex into the new `IMPORT`

vertex. For the build file, the change action detected is an update in the dependencies (deps), namely, the common base package is updated to common collect.

4.3 Resolution Changes

Our insight is that there are recurrent patterns in the way developers resolve build errors in practice. Such patterns can be automatically inferred from resolution changes and leveraged to assist developers.

Definition 4 (Resolution Change (RC)). *A resolution change (RC) is an AST Diff between the broken and resolved snapshots of the code, in a build resolution session.*

5 REPAIRING BUILD ERRORS

Recent studies [2, 21] suggest that models originally developed for analyzing natural language, such as n-gram models, are also effective for reasoning about source code. This has come to be known as the software *naturalness hypothesis* [2], which states that large code corpora are statistically similar to natural-language text, since coding is also an act of human communication. Following this naturalness hypothesis, we believe probabilistic machine learning models that target natural language can be further exploited for helping software developers. More specifically, the idea we propose here is to formulate the task of suggesting build repairs as a Neural Machine Translation (NMT) problem. Instead of translating one natural language into another, in our case, a given *source* build-diagnostic feature is "translated" to a *target* resolution change feature that resolves the diagnostic.

5.1 Feature Extraction

We generate features from each resolution change: a build diagnostic and the edits that fix that diagnostic. We use the generated features as input into a neural network to learn patterns of transformations between build diagnostics and resolution changes.

For each resolution change in the resolution sessions, we generate a source feature and a target feature, separately. A pair of source and target features capture information about the build failure and AST changes made to resolve the failure, respectively.

Source Features. The source feature pertains to the build diagnostic kind and its textual description. These can be included in the source features for any diagnostic kind without any diagnostic-specific knowledge.

To provide more contextual information to the machine-learning algorithm, we can optionally add more diagnostic-specific information to the source feature. For instance, for `cant.resolve`, we parse the snapshot of the broken code into an AST. We locate the missing symbol on the AST using the information provided by the compiler, such as its label and location. Once the symbol is located, we traverse the tree to extract its AST path.

Definition 5 (AST Path (AP)). *The AST Path AP of a missing symbol S_m is defined as the sequence of AST vertices from the root to the parent vertex of S_m on the AST of the broken snapshot.*

Figure 3 highlights the AST path for the missing symbol, `ImmutableList`, in our running example. Our evaluations show that including the

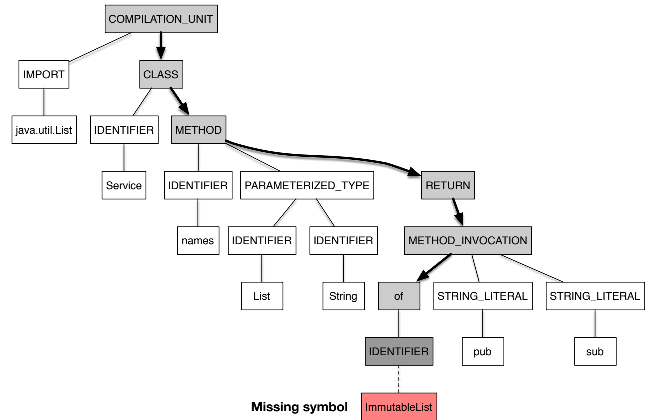


Figure 3: AST Path of `ImmutableList`.

AST path can increase the accuracy of the technique between 15-20%. The AST path provides the deep neural network with contextual information about the missing symbol, such as whether the symbol is a local variable inside a method or a class variable.

In addition to the AST path of the symbol, its tree vertex type and label, as well as its child vertices (e.g., type arguments for method calls) are added to the source feature.

For our running example, the source feature would include:

- Diagnostic kind: `compiler.err.cant.resolve`
- Diagnostic text: cannot find symbol
- AST path: `COMPILATION_UNIT CLASS METHOD RETURN METHOD_INVOCATION of`
- Symbol type: `IDENTIFIER`
- Symbol label: `ImmutableList`

For `cant.apply.symbol`, we augment the source feature with three types of data, namely, expected, found, and reason. Expected pertains to the expected types inferred by the compiler, found shows the types found in the code, and reason represents a textual description of why the compiler cannot apply the symbol.

Target Features. The target feature contains information about the resolution changes, i.e., AST changes made to the failing snapshot to resolve the build diagnostic.

To capture the resolution-change features, we define a domain-specific language (DSL) called Delta. Delta's grammar is formally specified in the Extended Backus-Naur form (EBNF) for ANTLR [38] and shown in Listing 1.

Each Delta feature starts with a file type (i.e., `JAVAFILE` or `BUILDFILE`) where the change was applied, followed by a series of change actions. Each change action contains an AST-change type (e.g., `INSERT`, `UPDATE`) and the changed AST node's type and value. For change types `INSERT`, `DELETE`, and `MOVE`, the parent node of the changed node is also included to provide more contextual information about the relative proximity of the changed node on the AST. For `UPDATE`, the before and after values of the changed nodes are captured.

For our running example, the target resolution-change features for Java and for the build file are shown in Listing 2 and Listing 3, respectively.

```
Delta.grammar

resolution_change_feature
  : file_type WS (change_action (WS)?)* EOF ;

change_action
  : change_type WS (location WS)? single_token token_seq ↩
       location WS single_token
    token_seq ;

file_type : 'BUILDFILE' | 'JAVAFILE' ;
change_type : 'INSERT' | 'DELETE' | 'UPDATE' | 'MOVE' ;
location : 'INTO' | 'FROM' | 'BEFORE' | 'AFTER' ;
single_token : TOKEN WS ;
token_seq : (TOKEN (WS)*)* ;
WS : (' ' | '\t') ;
TOKEN : ( COLON | QUOTE | COMMA | LOWERCASE | UPPERCASE | DIGIT ↩
     | UNDERSCORE ) +;

fragment UNDERSCORE : '_' ;
fragment COLON : ':' ;
fragment QUOTE : '"' ;
fragment COMMA : ',' ;
fragment LOWERCASE : [a-z ] ;
fragment UPPERCASE : [A-Z ] ;
fragment DIGIT : [0-9] ;
```

Listing 1: Delta grammar

```
fileType  JAVAFILE
change_action
        change_type     INSERT
        single_token    IMPORT
        location        INTO
        single_token    COMPILATION_UNIT
change_action
        change_type     INSERT
        single_token    com
        token_seq       google common collect ImmutableList
        location        INTO
        single_token    IMPORT
```

Listing 2: Delta Java example

```
fileType  BUILDFILE
change_action
  change_type     UPDATE
  location        BEFORE
  single_token    java
  token_seq       com google common base
  location        AFTER
  single_token    java
  token_seq       com google common collect
```

Listing 3: Delta build file example

Feature Generation. Once features are computed, all source and target features are analyzed, separately, to generate two vocabulary lists of $|V|$ frequent tokens for source and target, respectively. For instance, since the token 'COMPILATION_UNIT' is the root node of the AST path, it occurs frequently in the source features. Therefore, this token will be in the source vocabulary list. Similarly, the token 'BUILDFILE' exists in many target features and will be included in the target vocabulary list. These frequent tokens are used for embeddings during the training and inference, i.e., tokens from the vocabulary are mapped to vectors of real numbers for training and inferred vectorized representations are mapped back to the tokens in vocabulary for inference.

Finally, the features dataset is randomly partitioned into three chunks of 70%, 15%, and 15% for training, online evaluation of the model during training, and offline evaluation of the model against unseen features (see Section 6), respectively.

5.2 Learning Resolution Change Patterns

Recent deep NMT models have been quite successful at translating natural language text from a source language to a target language [47]. NMT achieves this by modelling and learning the conditional probability $p(y|x)$ of translating a source feature x into a target feature y through an encoder-decoder [6] setting, also known as seq2seq [45]. The encoder is responsible for computing a representation for each source feature x, without making any predictions. The decoder's task is to generate a translation y based on that source representation, by predicting the next tokens in the sequence.

Our deep neural network is built on top of TensorFlow [1]. It is composed of deep Long Short-Term Memory (LSTM) [22] Recurrent Neural Networks (RNNs) of 1024 units with 4 encoder and 4 decoder layers. As encoder type we use the Google Neural Machine Translation (GNMT) encoder [47], which is composed of 1 bi-directional layer and 3 uni-directional layers.

As optimizer, we employ the Stochastic Gradient Descent (SGD) algorithm [9] with a learning rate of 1.0. To mitigate over-fitting the model, we set a dropout value of 0.2. The idea is to randomly ignore units from the neural network during training, which prevents co-adaptations on the training data [44].

LSTMs perform well for short to medium input sequences but fail on large sequences. Attention mechanisms [3, 31] solve this limitation to a large extent by extending the attention span of the network. Since our resolution-change feature sequences could potentially be long, in our network, we adopt the normed Bahdanau attention [3].

5.3 Inferring Repair Suggestions

The whole process of generating resolution sessions, resolution changes, and features, as well as training the model is pipelined using sequential dependencies, which makes our whole learning process automated and repeatable.

Once the model is trained, it is uploaded to a server where we can query it for repair inference. The model can produce various translations for any given input. In our NMT setting, the translation is carried out using beam search [47], a heuristic search algorithm that makes a trade-off between translation time and accuracy. The input to the model is a source feature x representing a compilation failure. The inferred suggestions are returned as n sequences $\{y_1, y_2, \ldots, y_n\}$. Each y_i represents a distinct repair suggestion for x and is composed of a series of resolution change tokens to be applied to the failing program.

6 EVALUATION

We conducted an empirical evaluation to assess the efficacy of DeepDelta for the two most prevalent and costly compilation errors, namely `cant.resolve` and `cant.apply.symbol`.

Table 3: Generated features

Diagnostic Kind	Features	Train	Val	Test	Vocab
`cant.resolve`	265,456	186,992	42,363	36,101	30,000
`cant.apply.symbol`	25,201	19,407	3,107	2,687	30,000

6.1 Data Generation

The dataset we use for training and evaluation is described in Section 2.4, which is composed of developer build data collected over a two months period at Google.

AST diffs between the failing and resolved snapshots were computed for all green, singular resolution sessions in our dataset. We constrain the number of AST changes to be larger than zero and fewer than six, as higher numbers of changes often include refactorings, and in previous studies fixes have been shown to contain fewer than six modifications [34]. We computed 110,219 green, singular resolution sessions in total, over 37,867 distinct Java and 7,011 distinct build files.

Since we are dealing with a large industrial codebase, we set $|V|$, the maximum number of frequent tokens in the source and target vocabulary lists, to 30,000. The output of the feature-generation step is two vocabulary files for source and target, each containing a maximum of 30,000 unique tokens. In total, 265,456 and 25,201 source/-target features were generated for `cant.resolve` and `cant.apply.symbol`, respectively. These feature sets are randomly shuffled and partitioned in three separate categories, namely, Train (for training), Val (for online evaluation during training) and Test (for offline evaluation of the trained model) as presented in Table 3. Each category contains source and target feature pairs.

6.2 Training

We generate features and train two models separately for `cant.resolve` and `cant.apply.symbol` to compare the applicability of DeepDelta on different diagnostic kinds. In addition to the deep neural network setup described in Section 5.2, we configure the network as follows. The maximum sequence length for both source and target is set to 100. The batch size is 128, and the number of training steps is 100,000. We configure the inference to generate 10 suggestions (See Section 5.3).

To train a model, we feed the source and target features of the train dataset as well as the vocabulary lists to the network. The model starts by creating the source and target embeddings for all token representations. To that end, a vocabulary is provided for the source and target for tokens that are meant to be treated uniquely. We also feed the Val dataset to the network for online evaluation during training. Our models are trained on Google's Tensor Processing Units (TPUs) [24].

6.3 Evaluation Method

We use the Test datasets, containing the source and target features, for our evaluation. These are datasets that our trained models have not seen before. We evaluate each diagnostic kind separately. For each item in the test dataset, we retrieve the source feature and send it to the inference server to obtain repair suggestions. The target feature of the item in the test dataset, which is the fix the developer performed to repair the build error, is used as a baseline to assess the 10 suggestions generated by DeepDelta.

We employ different metrics for evaluating the repair suggestions.

Perplexity. Perplexity [4] measures how well a model predicts samples. Low (e.g., single digit) perplexity values indicate the model is good at predicting a given sequence.

BLEU. The next metric we use to assess the generated output of the model is BLEU [37]. BLEU is a well-known and popular metric for automatically evaluating the quality of machine-translated sentences. It has been shown to correlate well with human judgments [7, 17]. BLEU calculates how well a given sequence is matched with an expected sequence in terms of the actual tokens and their ordering using an n-gram model. The output of the BLEU metric is a number between 1–100. For natural language translations, BLEU scores of 25–40 are considered high scores [47].

Syntactic Validation. For validating the suggestions for syntactical correctness, we generate a lexer and parser from our Delta grammar through ANTLR4. We pass each inferred suggestion through the Delta lexer/parser. This way, we assess whether the model generates suggestions that conform to the grammar of the expected resolution changes. The output is binary, i.e., either the suggestion is valid or invalid.

Correctness of Suggestions. A source build diagnostic is considered correctly repaired if at least one of the 10 suggested repairs is valid and exactly matches the fix the developer performed, i.e., the target feature (baseline) in the test dataset. We use textual string equality for comparing each suggestion with the baseline.

Ranking of Correct Repairs. The ranking of the correct suggestion in the list of the suggestions is an indication of how well the model can generate the correct repair. The higher its ranking, the sooner the repair can be applied. For each failure that DeepDelta generates a correct suggestion, we note its position on the list of 10 suggestions.

6.4 Results

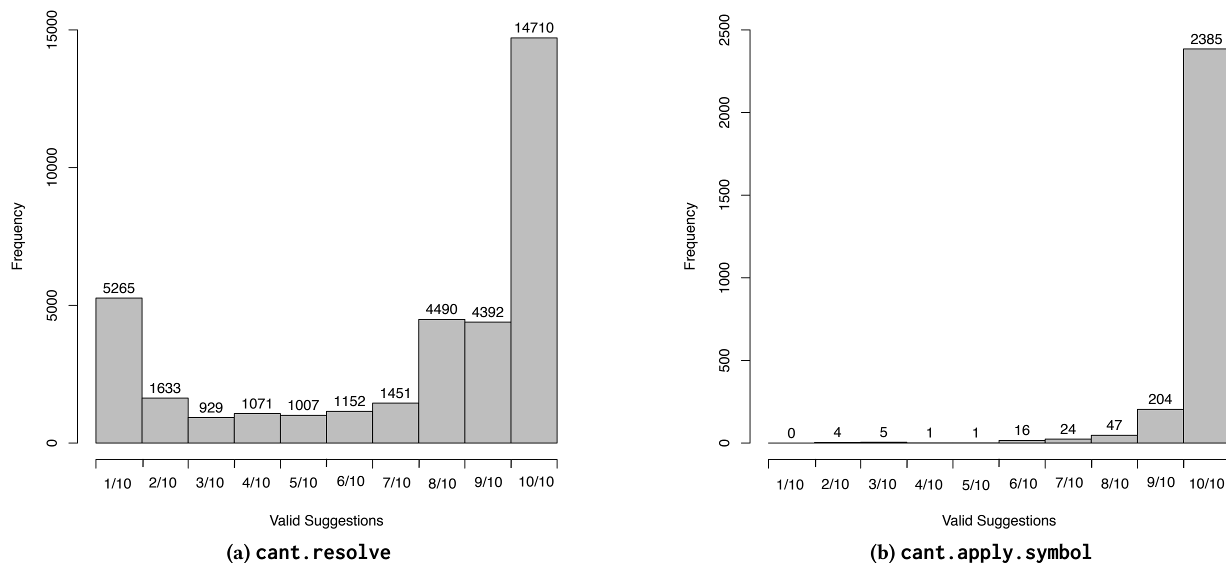
Table 4 presents our results for the two diagnostic kinds we evaluated. For each diagnostic kind, the table shows the achieved perplexity and BLEU scores, the number of compilation failures in the Test dataset evaluated against the trained models, the number of suggestions generated (i.e., 10 suggestions per failure), the average percentage of valid suggestions per failure, the percentage of correct suggestions overall, and the percentage of correct repairs that are ranked in the top 3 suggestions.

Perplexity and BLEU Scores. Recall that low perplexity (i.e., single digits) and high BLEU scores (i.e., 25–40) are desired for the models. As Table 4 shows, our models reached low perplexity values of 1.8 and 8.5 for `cant.resolve` and `cant.apply.symbol`, respectively. Also, the BLEU scores achieved were high, namely 42 for `cant.resolve` and 43 for `cant.apply.symbol`.

Validation. Figure 4 shows the distribution of valid suggestions over the 10 generated suggestions per failure as histograms. Our validation against the Delta grammar reported that on average

Table 4: Results

Diagnostic Kind	Perplexity	BLEU	Failures	Suggestions	Valid Suggestions	Correct Failure Repair	Ranked top 3
`cant.resolve`	1.8	42	36,101	361,010	71%	(18,051) 50%	85%
`cant.apply.symbol`	8.5	43	2,687	26,870	98%	(1,263) 47%	87%



(a) `cant.resolve`

(b) `cant.apply.symbol`

Figure 4: Distribution of valid suggestions over the 10 generated suggestions per failure.

71% of the generated suggestions are valid per failure for `cant.-resolve`. For `cant.apply.symbol`, the percentage of valid suggestions is higher at 98% since the code changes are syntactically simpler in nature (e.g., method argument changes). As it can be seen, the majority of the generated sequences are valid. We discuss the main reasons for invalid suggestions in Section 7.

Correctness. Table 4 shows that 18,051 out of the 36,101 (50%) failures in our Test dataset received a correct repair suggestion, i.e., one of the 10 suggestions for that failure was an exact match with the actual developer fix, for `cant.resolve`. For `cant.apply.symbol`, DeepDelta achieves a similar rate of correct suggestions, namely, 1,263 out of 2,687 (47%).

Ranking. For the failures with correct suggestions, we evaluate the position of the correct suggestion within the list of suggested resolutions. Figure 5 depicts the distribution of the position of the correct fixes in the 10 generated suggestions. Our data shows that the majority of the correct suggestions are on the first position. For `cant.resolve`, 85% of the correct fixes are in the top three positions. Similarly, for `cant.apply.symbol`, 87% of the correct fixes are in the top three positions.

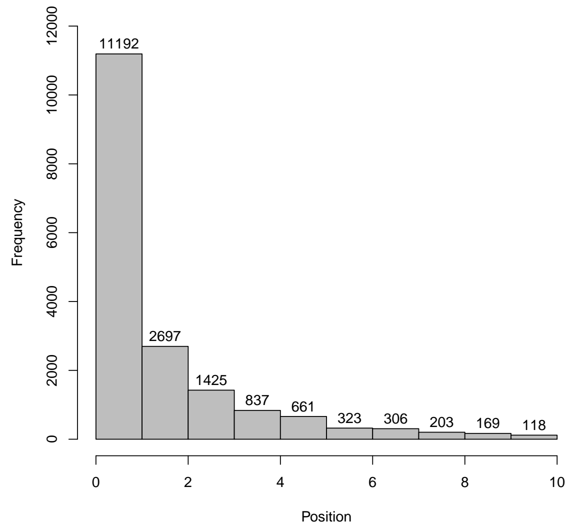
7 DISCUSSION AND THREATS TO VALIDITY

Invalid Sequences. We investigated the main reasons behind the invalid suggestions, which can be attributed to:

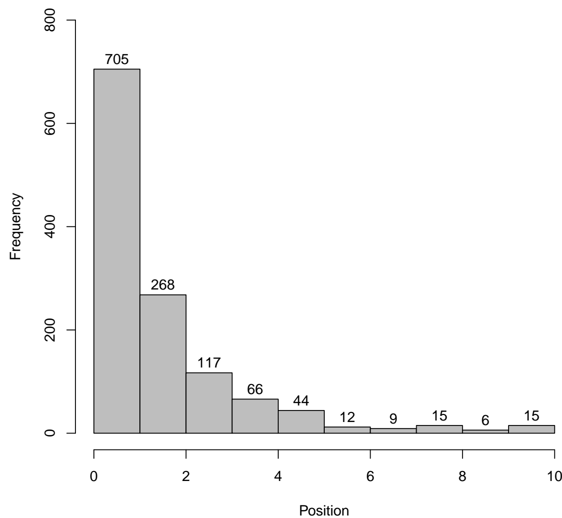
(1) Unknown tokens: the occurrence of the <unk> token in the inferred target sequences. The model predicts a vectorized embedding that the decoder then tries to translate into a token; sometimes that embedding falls far away from the valid tokens in the given vocabulary and cannot be matched to one of the frequent tokens. This can happen for failures that do not have a fix change pattern the model has learnt or when our vocabulary does not contain the token because it is not a frequent token.
(2) Incomplete sequences: the sequence generated misses parts of the expected Delta grammar sequence. This happens especially for longer sequences (i.e., more than 90 tokens) and missing tokens at the end part of sequences.

Correctness. DeepDelta is able to suggest an exact correct suggestion for around half of the build failures in our evaluation. Note that a correct suggestion is not a simple binary output. It is composed of a complex sequence of AST changes to be applied to the failing code. Compared to the state-of-the-art repair rate of 27% for syntax errors [18], we believe our 50% correctness rate is substantial in practice. Our relatively high accuracy rate can potentially be attributed to the following properties of our dataset and approach:

- There are indeed recurrent and common patterns in how developers resolve build failures, which DeepDelta is able to extract from the resolution changes,
- Our target language, Delta, is highly structured, allowing the model to learn the patterns in a systematic manner.



(a) `cant.resolve`



(b) `cant.apply.symbol`

Figure 5: Position of the correct suggestion.

Generalizibility. It is possible that our results and techniques would not generalize to other development environments. For example, environments where IDE use is more prevalent may have a different dominant diagnostic kind. However, although our dataset is from one company, nearly all projects are included in a single large source-code repository at Google. Thus, our dataset

- Includes more than 10,000 Java projects containing 300 million LOC. Of these projects, many are open-source, though we only have data on those developed primarily at Google.
- Represents changes made by tens of thousands of professional developers worldwide, from various backgrounds, using around five different editors/IDEs.
- Contains 110,219 singular build-time compilation errors and fixes, over 37,867 distinct Java and 7,011 distinct build files, collected over a two-month period.

Thus we believe it seems reasonable that our program repair technique would generalize to any other coherent development environment, where there are patterns in how developers address a particular diagnostic kind. While the particular fixes our system learns and suggests would not work in other repositories and build systems, the technique itself is general and should apply to any other source code repository and build system. For example, consider the case of adding a dependency on another project to a Bazel build file. This concept exists in other widely used build systems. As long as a parser were provided to parse other types of build configuration files (e.g., POM files for Maven, build.gradle for Gradle), our system should be able to learn how to add dependencies in the appropriate way.

Reproducibility. Our dataset contains AST of proprietary code, which unfortunately cannot be published. Our dataset is unique because it contains snapshots of in-progress code, not just code committed to source control. This granularity is key to making our approach work. We are not aware of any open-source datasets that provide this level of error/fix detail. Our findings can be reproduced in other contexts by collecting fine-grained developer histories, and using our specification of the Delta language and the open-source TensorFlow NMT library.

Diagnostic kinds. Our work addresses diagnostics only for Java projects. However, our technique is general and relies only on AST differencing; the only language-specific portion is the parsers used to build the ASTs. We should be able to support other languages simply by implementing parsers for them.

We focused on two error types (`cant.resolve` and `cant.apply.symbol`) in this work for two reasons: First, these cover the majority of Java compilation errors developers make at Google: 59% of all instances, 58% by cost (Table-II). Our data show that, whether or not they are easy to fix (R1), in practice developers spend a huge amount of time manually fixing them. Automating these repairs would free developers to focus on other issues. Second, adding a new error type requires collecting relevant developer data and re-training the model, which is time-consuming.

Our results show that DeepDelta works well for two distinct error types with different fix patterns. We expect to perform as well on other build-error types and plan to extend support to other types. We are not aware of any tools that repair Java compilation errors to compare directly against in our evaluation. The most related work we found [18] repairs syntax errors (a single error type only) in C, with a far lower success rate.

Parsable AST. For our AST-diff-based approach to work, the broken code must be parsable into an AST. With the missing symbol errors we examined, the code is always parseable resulting in a

unique AST, but this will not be true for other diagnostic kinds. Parsers can be designed to recover from syntax errors instead of failing fast [5]. We may need to switch to such a parser to handle incomplete ASTs.

8 FUTURE WORK

Putting the Program Repair Tool Into Production. We intend to integrate DeepDelta into the development environment of developers at our company. To that end, we will need to verify suggestions before applying them to user code. We intend to build a system to speculatively apply suggested fixes to a code snapshot and attempt to build them. We can then discard suggested fixes that do not compile, and only present the ones that do to the user. There is also interesting user interface work to do here to ensure that the tool is actually useful to developers.

Repairing Programs that Fail Tests. At Google we also capture output from failing and successful test executions and the code changes that caused them to begin passing. We plan to experiment with applying our technique to repair programs that fail test cases. We targeted build failures first because the repairs seemed more regular and thus more likely that a machine learning algorithm would be able to find patterns in the data. We intend to explore whether there are enough patterns in fixes for failing tests for our technique to work well.

9 RELATED WORK

We survey four lines of related work: extracting code changes, synthesizing transformations from examples, automated program repair, and machine learning for program repair.

Extracting Change Patterns. Extracting change patterns from code has received some attention in the literature. Most existing techniques, however, require pre-defined rules or human intervention to extract patterns. Fluri and Gall [14] define 41 basic change types for Java, which they use to discover more complex changes [15] through hierarchical clustering. Pan [36] use the Unix diff to create a database of basic changes and use Datalog to manually specify rules for more complex change types. Liveshits and Zimmerman [27] propose a technique for discovering API usage patterns through association rule mining of code histories of two Java projects. They use the patterns to detect violations. Kim et al. [26] manually inspect human-written patches and extract six common fix patterns in Java. They subsequently use these patterns for automated program repair. Hanam et al. [19] provide a semi-automatic approach for bug fix pattern detection. They extract feature vectors of language construct AST changes that resolve runtime bugs and use clustering to group them into ranked clusters of bug patterns. The clusters are then manually inspected to extract bug fix patterns. Our approach requires no manual intervention and learns *unknown* resolution change patterns.

Synthesizing Transformations by Example. Another related area of work is code transformation techniques, such as LASE [33], Genesis [28], NoFAQ [8], and REFAZER [40], which can extract and synthesize syntactic transformations from given examples. These techniques could potentially be used for program repair. Unlike our work, they operate either on a single example or a small set of examples; it is unclear how well they would perform on extracting patterns from hundreds of thousands of examples, and how to apply the synthesized transformations in the setting of program repairs.

Automated Program Repair. Our work falls in the realm of automated program repair, which pertains to the act of fixing bugs through automated techniques. Program repair has been applied to different domains such as data structures [11, 12], user interfaces [48], and source code of different programming languages such as C [16, 25, 30], Java [10, 26], JavaScript [35], and PHP [41].

Patch search techniques have a number of shortcomings in practice. First, they often require pre-defined templates of bug patterns and cannot learn new patterns. Second, the patch generation process needs to search the vast program space, which can be costly as thousands of patches need to be generated to find one that resolves the failure. Finally, they have been shown [29] to produce many false positives, i.e., they often fix the test failure, but not the actual fault.

Machine Learning for Program Repair. Allamanis et al. [2] provide a comprehensive survey of recent advancements in techniques that adopt machine learning for source-code analysis. Wang et al. [46] propose a technique for fault prediction. They feed abstract semantic features of the code to a neural network for classification. Raychev et al. [39] use neural networks for code completion of missing API calls. Seidel et al. [42] target type error localization through a supervised classification approach.

Gupta et al. [18] propose a seq2seq machine learning approach for repairing syntax errors in C. One main difference with our work is that they feed the whole source code of the buggy and fixed versions to the network and achieve a repair rate of 27%. We, however, focus only on learning the features of the failure and the accompanying AST changes that resolve it, which allows us to achieve a much higher accuracy rate (50%). They target syntax errors in C, while we target build-time compilation errors in Java. In addition, they evaluate their work on small student assignments, while our evaluation is on a large corpus of real developer data.

Our paper is the first to learn AST change patterns for fixing compilation errors. Related work that targets build errors [20, 32] is different from ours. They focus only on build-files while we target both Java and build files; they use pre-defined repair-templates, while we learn repair patterns. And, we do it at orders of magnitude larger scale; [32] and [20] use 37 and 175 failures, respectively, while we evaluate DeepDelta on 38,788 failures.

10 CONCLUSION

In this paper, we studied build diagnostics and how developers change the source code to resolve them in practice. We showed that patterns exist in such code changes. We proposed a generic technique to learn patterns of code changes, extracted from AST diffs between failure and resolution pairs. We formulated automated program repair as a machine translation problem. Using a deep neural network, our technique, DeepDelta, is capable of suggesting AST changes when given a build diagnostic as input. Our evaluation on a large corpus of real developer data at Google shows that DeepDelta generates correct fixes between 47%–50% of the time for two compiler diagnostic kinds, with the correct fix ranked in the top three 85%–87% of the time.

Our current system suggests fixes for the two most costly diagnostic kinds, `cant.resolve` and `cant.apply.symbol`. We believe repairing other compilation errors can be supported through DeepDelta and we intend to expand support to other diagnostic kinds.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182* (2017).

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

[4] Peter F Brown, Vincent J Della Pietra, Robert L Mercer, Stephen A Della Pietra, and Jennifer C Lai. 1992. An estimate of an upper bound for the entropy of English. *Computational Linguistics* 18, 1 (1992), 31–40.

[5] Michael G. Burke and Gerald A. Fisher. 1987. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Trans. Program. Lang. Syst.* 9, 2 (March 1987), 164–197. https://doi.org/10.1145/22719.22720

[6] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[7] Deborah Coughlin. 2003. Correlating automated and human assessments of machine translation quality. In *Proceedings of MT summit IX*. 63–70.

[8] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs from Examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 582–592. https://doi.org/10.1145/3106237.3106241

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[10] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*. ACM, 30–39.

[11] Brian Demsky and Martin Rinard. 2005. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th international conference on Software engineering*. ACM, 176–185.

[12] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. 2007. Assertion-based repair of complex data structures. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 64–73.

[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering (ASE)*. 313–324. https://doi.org/10.1145/2642937.2642982

[14] Beat Fluri and Harald C Gall. 2006. Classifying change types for qualifying change couplings. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 35–45.

[15] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007).

[16] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[17] Yvette Graham and Timothy Baldwin. 2014. Testing for significance of increased correlation with human judgment. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 172–176.

[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*. 1345–1351.

[19] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 144–156.

[20] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1078–1089.

[21] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] James Wayne Hunt and M Douglas McIlroy. 1976. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill.

[24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the Annual International Symposium on Computer Architecture*. ACM, 1–12.

[25] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 295–306. https://doi.org/10.1109/ASE.2015.60

[26] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 802–811.

[27] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 296–305.

[28] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 727–739.

[29] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[30] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 702–713. https://doi.org/10.1145/2884781.2884872

[31] Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. 1412–1421.

[32] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 106–117.

[33] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. IEEE Press.

[34] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 180–190.

[35] Frolin Ocariza, Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: Suggesting Fixes for JavaScript Faults. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 837–847.

[36] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.

[37] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 311–318. https://doi.org/10.3115/1073083.1073135

[38] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[39] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[40] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 404–415.

[41] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 277–287.

[42] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-driven Diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 60 (Oct. 2017), 27 pages. https://doi.org/10.1145/3138818

[43] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the International Conference on Software Engineering (ICSE) (ICSE 2014)*. ACM, New York, NY, USA, 724–734. https://doi.org/10.1145/2568225.2568255

[44] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

[45] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[46] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 297–308.
[47] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ?ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). http://arxiv.org/abs/1609.08144
[48] Sai Zhang, Hao Lü, and Michael D Ernst. 2013. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 45–55.