SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk and Martin Monperrus

Abstract—This paper presents a novel end-to-end approach to program repair based on sequence-to-sequence learning. We devise, implement, and evaluate a technique, called SEQUENCER, for fixing bugs based on sequence-to-sequence learning on source code. This approach uses the copy mechanism to overcome the unlimited vocabulary problem that occurs with big code. Our system is data-driven; we train it on 35,578 samples, carefully curated from commits to open-source repositories. We evaluate SEQUENCER on 4,711 independent real bug fixes, as well on the Defects4J benchmark used in program repair research. SEQUENCER is able to perfectly predict the fixed line for 950/4,711 testing samples, and find correct patches for 14 bugs in Defects4J benchmark. SEQUENCER captures a wide range of repair operators without any domain-specific top-down design.

Index Terms—program repair; machine learning.

1 INTRODUCTION

PEOPLE have long dreamed of machines capable of writing computer programs by themselves. Having machines writing a full software system is science-fiction but teaching machines to modify an existing program to fix a bug is within the reach of current software technology; this is called automated program repair [1].

Program repair research is very active and dominated by techniques based on static analysis (*e.g.,* Angelix [2]) and dynamic analysis (*e.g.,* CapGen [3]). While great progress has been achieved, the current state of automated program repair is limited to simple small fixes, mostly one line patches [3], [4]. These techniques are heavily top-down, based on intelligent design and domain-specific knowledge about bug fixing in a given language or a specific application domain. In this paper, we also focus on one line patches, but we aim at doing program repair in a language-agnostic generic manner, fully relying on machine learning to capture syntax and grammar rules and produce well-formed, compilable programs. By taking this approach, we aim to provide a foundation for connecting program repair and machine learning, allowing the program repair community to benefit from training with more complete bug datasets and continued improvements to machine learning algorithms and libraries.

As the foundation for our model, we apply sequence-to-sequence learning [5] to the problem of program repair. Sequence-to-sequence learning is a branch of statistical machine learning, mostly used for machine translation: the algorithm learns to translate text from one language (say French) to another language (say Swedish) by generalizing over large amounts of sentence pairs from French to Swedish. The training data comes from the large amount of text already translated by humans, starting with the Rosetta stone written in 196 BC [6]. The name of the technique is explicit: it is about learning to translate from one sequence of words to another sequence of words.

Now let us come back to the problem of programming: we want to learn to 'translate' from one sequence of program tokens (a buggy program) to a different sequence of program tokens (a fixed program). The training data is readily available: we have millions of commits in open-source code repositories. Yet, we still have major challenges to overcome when it comes to using sequence-to-sequence learning on code: 1) the raw (unfiltered) data is rather noisy; one must deploy significant effort to identify and curate commits that focus on a clear task; 2) contrary to natural language, misuse of rare words (identifiers, numbers, etc) is often fatal in programming languages [7]; in natural language some errors may be tolerable because of the intelligence of the human reader while in programming languages the compiler (or interpreter) is strict 3) in natural language, the dependencies are often in the same sentence ("it" refers to "dog" just before) , or within a couple of sentences, while in programming, the dependencies have a longer range: one may use a variable that has been declared dozens of lines before.

We are now at a tipping point to address those challenges. First, sequence-to-sequence learning has reached a maturity level, both conceptually and from an implementation point of view, that it can be fed with sequences whose characteristics significantly differ from natural language. Second, there has been great recent progress on using various types of language models on source code [8]. Based on this great body of work, we present our approach to using sequence-to-learning for program repair, which we created

- *Zimin Chen and Martin Monperrus are with KTH Royal Institute of Technology, 114 28 Stockholm, Sweden E-mail: {zimin, monp}@kth.se*
- *Steve Kommrusch and Louis-Noël Pouchet are with Colorado State University, Colorado 80523, USA Email: {steveko, pouchet}@cs.colostate.edu*
- *Michele Tufano and Denys Poshyvanyk are with The College of William and Mary, VA 23185, USA Email: {mtufano, denys}@cs.wm.edu*
- *Zimin Chen and Steve Kommrusch have equally contributed to the paper as first authors.*

Manuscript submitted February 11, 2019

to repair real bugs from large open-source projects written in the Java programming language.

Our end-to-end program repair approach is called SEQUENCER and it works as follows. First, we focus on one-line fixes: we predict the fixed version of a buggy programming line. For this, we create a carefully curated training and testing dataset of one-line commits. Second, we devise a sequence-to-sequence network architecture that is specifically designed to address the two main aforementioned challenges. To address the unlimited vocabulary problem, we use the copy mechanism [9]; this allows SEQUENCER to predict the fixed line, even if the fix contains a token that was too rare (*i.e.,* an API call that appears only in few cases, or a rare identifier used only in one class) to be considered in the vocabulary. This copy mechanism works even if the fixed line should contain tokens which were not in the training set. To address the dependency problem, we construct *abstract buggy context* from the buggy class, which captures the most important context around the buggy source code and reduces the complexity of the input sequence. This enables us to capture long range dependencies that are required for the fix.

We evaluate SEQUENCER in two ways. First, we compute accuracy over 4,711 real one-line commits, curated from three open-source projects. The accuracy is measured by the ability of the system to predict the fixed line exactly as originally crafted by the developer, given as input the buggy file and the buggy line number. Our golden configuration is able to perfectly predict the fix for 950/4,711 (20%) of the testing samples. This sets up a baseline for future research in the field. Second, we apply SEQUENCER to the mainstream evaluation benchmark for program repair, Defects4J. Of the 395 total bugs in Defects4J, 75 have one-line replacement repairs; SEQUENCER generates patches which pass the test suite for 19 bugs and patches which are semantically equivalent to the human-generated patch for 14 bugs. To our knowledge, this is the first report ever on using sequence-to-sequence learning for end-to-end program repair, including validation with test cases.

Overall, the novelty of this work is as follows. First, we create and share a unique dataset for evaluating learning techniques on one-line program repair. Second, we report on using the copy mechanism on seq-to-seq learning on source code. Third, on the same buggy input dataset, SEQUENCER is able to produce the correct patch for 119% more samples than the closest related work [10].

To sum up:

- Our key contribution is an approach for fixing bugs based on sequence-to-sequence learning on token sequences. This approach uses the copy mechanism to overcome the unlimited vocabulary problem in source code.
- We present the construction of an *abstract buggy context* that leverages code context for patch generation. The input program token sequences are at the level of full classes and capture long-range dependencies in the fix to be written. We implement our approach in a publicly-available program repair tool called SEQUENCER.
- We evaluate our approach on 4,711 real bug fixing tasks. Contrary to the closest related work [10], we do not assume bugs to be in small methods only. Our golden trained model is able to perfectly fix 950/4,711 testing samples. To the best-of-our knowledge, this is the best result reported on such a task at the time of writing this paper [10][11][12].
- We evaluate our approach on the 75 one-line bugs of Defects4J, which is the most widely used benchmark for evaluating programming repair contributions. SEQUENCER is able to find 2,321 patches for these bugs, 761 compile successfully, 61 are plausible (they pass the full test suite) and 18 are semantically equivalent to the patch written by the human developer.
- We provide a qualitative analysis of 8 interesting repair operators captured by sequence-to-sequence learning on the considered training dataset.

2 BACKGROUND ON NEURAL MACHINE TRANSLATION WITH SEQUENCE-TO-SEQUENCE LEARNING

SEQUENCER is based on the idea of receiving buggy code as input and producing fixed code as output. The concept is similar to neural machine translation where the input is a sequence of words in one language and the output is a sequence in another language. In this section, we provide a brief introduction to neural machine translation (NMT).

In neural machine translation, the dominant technique is called "sequence-to-sequence learning", where "sequence" refers to the sequence of words in a sentence. An early example of a sequence-to-sequence network [5] used a recurrent neural network to read in tokens and to generate an output sequence, as shown in Figure 1. Let us consider that the input tokens are denoted x_t, and after receiving all of the input tokens a special <EOS> token is used. The output tokens are denoted y_t, and at training time the output tokens are fed into the network to learn proper generation of the next token. In the following equations, h_t is the hidden state of a recurrent neural network, W^{hx} is the weight matrix that computes how the input x_t affects the hidden state, W^{hh} is the weight matrix related to recurrence (*i.e.,* how the previous hidden state affects the current hidden state), and W^{yh} is the weight matrix used to predict which token should be output given the hidden state. All weights are learned with supervised learning and back-propagation:

$$h_t = \sigma(W^{hx}x_t + W^{hh}h_{t-1})$$
$$y_t = W^{yh}h_t$$

A softmax function is then used to turn the y_t values into probabilities to choose the most likely token from a learned vocabulary. In this example, one can see how the weight matrices capture the learning of common patterns; after processing the input sequence, the hidden state $h_{<eos>}$ encodes the most likely initial token to begin the output and each subsequent h_t uses the W matrices to predict the most likely next token given the input as well as preceding tokens just produced in the output. The W matrices thus learn the long range dependencies in the full input.

A problem with the sequence generation described above is that only tokens which are in the training set are available for output as y_t. In the case of natural human language, words such as proper names (*e.g.,* Chicago, Stockholm) may be so rare that they do not appear in the training vocabulary, but those words may be necessary for

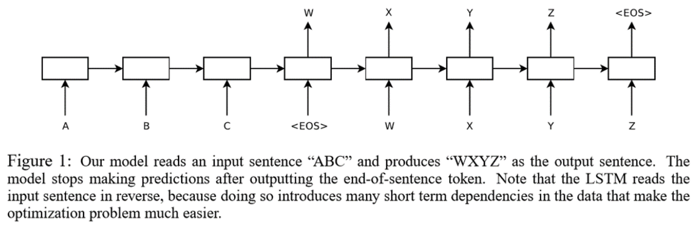

Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

Fig. 1: Figure from Sutskever, *et al.* [5] showing example of early sequence-to-sequence model

proper output. One successful approach to overcome the vocabulary problem is to use a copy mechanism [9]. The basic intuition behind this approach is that rare words not available in the vocabulary (*i.e.,* unknown words, referred as `<unk>`), may be directly copied from the input sentence over to the output translated sentence. This relatively simple idea can be successful in many cases - especially when translating sentences containing proper names - where these tokens can be easily copied over.

For example, let's consider the task of translating the following English sentence *"The car is in Chicago"* to French. Let's also assume that all the tokens in the sentence are in the vocabulary, except *"Chicago"*. An NMT model might output the following sentence: *"La voiture est à* `<unk>`*"*. With a copy mechanism, the model would be able to automatically replace the unknown token with one of the tokens from the input sentence, in this case, *"Chicago"*.

The copy mechanism can be particularly relevant for source code, where the size of the vocabulary can be several times the size of a natural language corpus [13]. This results from the fact that developers are not constrained by any vocabulary (*e.g.,* English dictionary) when defining names for variables or methods. This leads to an extremely large vocabulary containing many rare tokens, used infrequently only in specific contexts. Thus, the copy mechanism applied to source code allows a system to generate rare out-of-vocabulary identifier names and numeric values as long as they are somewhere in the input. Furthermore, in natural language, a human recipient may be able to use context to cope with one missing word in an automatically translated sentence. In a programming language, the compiler does not make any semantic inference, and the generation has to be complete. For example, if the code to predict is "if (i < num_cars)", then generating "if (i < int)" is not going to work at all. We discuss the mathematics of the copy mechanism in the context of SEQUENCER in Section 3.3.1. Readers interested in more detail are referred to the work by See et al. [9].

Tufano *et al.* [10] proposed using NMT with the goal of learning bug-fixing patches by translating the entire buggy method into the corresponding fixed method. Before the translation, the authors perform a code abstraction process which transforms the source code into an abstracted version, which contains: (i) Java keywords and identifiers; (ii) frequent identifiers and literals (a selection of 300 idioms); (iii) typified IDs (*e.g.,* `METHOD_1`, `VAR_2`) that replace identifiers and literals in the code. In Section 6 we highlight differences and improvements introduced in SEQUENCER.

Another approach to addressing the vocabulary size problem in code is to use byte pair encoding (BPE), which has been widely used in NLP and also applied to source code [14]. For SEQUENCER, we did preliminary experiments with BPE to solve the unlimited vocabulary problem, but our early results showed that it is less effective than the copy mechanism.

3 Approach to Using Seq-to-Seq Learning for Repair

SEQUENCER is a sequence-to-sequence deep learning model that aims at automatically fixing bugs by generating one-line patches (*i.e.,* the bug can be fixed by replacing a single buggy line with a single fixed line). We do not consider line deletion because: 1) it does not require a method for token generation (and is thus less interesting to our research) and 2) if desired, SEQUENCER could be combined with the lightweight Kali [11] to include line deletion. We do not consider line addition because spectrum based fault localization, used in most of the related work, is not effective for line addition patches [15]. We note that in 64% of all 395 bugs in Defects4J are fixed by replacing existing source code [16]. Given a Software System with a faulty behavior (*i.e.,* failing test case), state-of-the-art fault localization techniques are used to identify the buggy method and the suspicious buggy lines. Such techniques have been shown to predict the correct buggy line as one of the top 10 candidates in 44% of the time [15]. SEQUENCER then performs a novel ***Buggy Context Abstraction*** (Section 3.2) process which intelligently organizes the fault localization data (*i.e.,* buggy classes, methods, and lines) into a representation that is concise and suitable for the deep learning model yet able to preserve valuable information regarding the context of the bug, which will be used to predict the fix. The representation is then fed to a trained sequence-to-sequence model (Section 3.3.1) which performs ***Patch Inference*** (Section 3.4) and is capable of generating multiple single-lines of code that represent the potential one-line patches for the bug. Finally, SEQUENCER in the ***Patch Preparation*** (Section 3.5) step generates the concrete patches by formatting the code and replacing the suspicious line with the proposed lines. Figure 2 shows the aforementioned steps both for the training phase (left) and inference phase (right). In the remainder of this section we will discuss the common steps as well as those specific for training and inference.

3.1 Problem Definition

Given a buggy system b^s, and test suite t, we assume a fault localization technique, FL, which identifies an ordered set of potential bug locations $l = \{l_1, l_2, ...\}$, where each location l_i consists of the buggy class b_i^c, buggy method b_i^m, and the buggy line b_i^l:

$$l = \{loc \mid loc \in FL(b^s, t)\}$$
$$\forall l_i \in l, l_i = \{b_i^c, b_i^m, b_i^l\} \quad \text{and} \quad b_i^l \subset b_i^m \subset b_i^c$$

The problem is to predict (*i.e.,* generate) a fixed line f_i^l, where l_i is the true bug location, such that by replacing b_i^l with f_i^l in b_i^m, the resulting system f^s passes the test suite and the bug is considered fixed. SEQUENCER tackles this

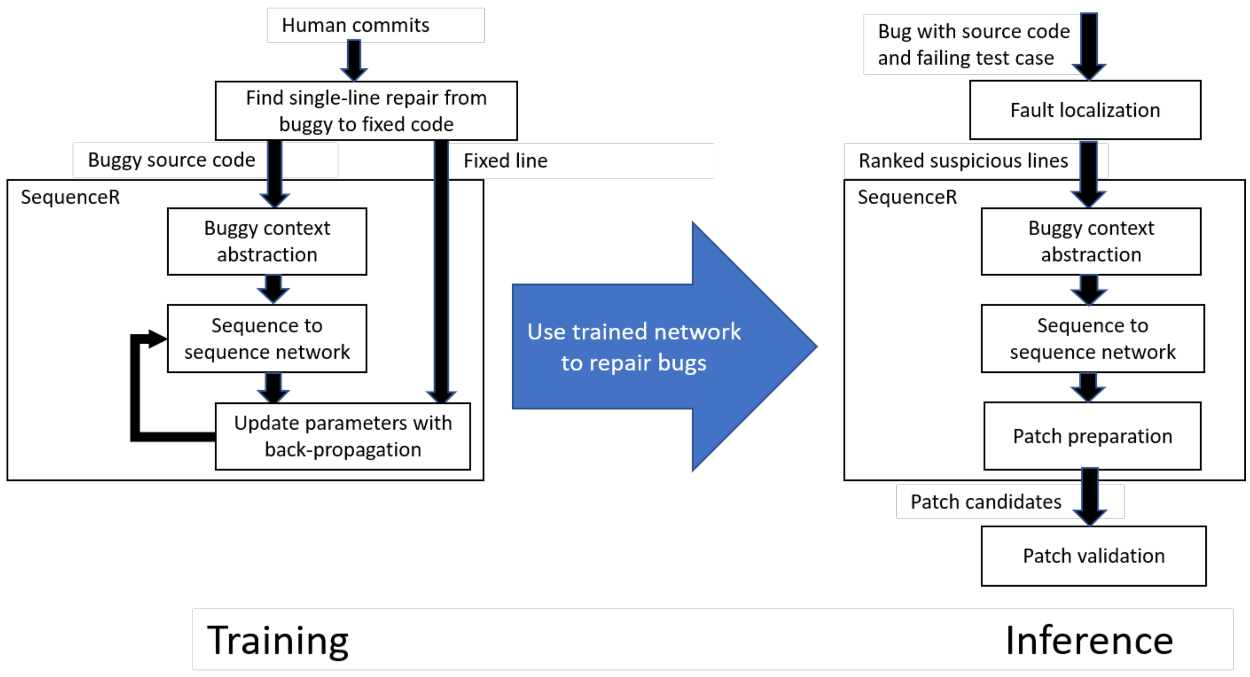
Human commits
Find single-line repair from buggy to fixed code
Buggy source code
Fixed line
SequenceR
Buggy context abstraction
Sequence to sequence network
Update parameters with back-propagation
Use trained network to repair bugs
Bug with source code and failing test case
Fault localization
Ranked suspicious lines
SequenceR
Buggy context abstraction
Sequence to sequence network
Patch preparation
Patch candidates
Patch validation
Training
Inference

Fig. 2: Overview of our approach using sequence-to-sequence learning for program repair.

```
class Foo {
  int i = 0;
  int bar;
  Foo (int bar){
    this.bar = bar;
  }
  int decrement(){
    return bar−1;
  }
  int increment(){
    return bar−1;
  }
}
```

Listing 1: Original code

```
class Foo {
  int i = 0;
  int bar;
  Foo (int bar){
  }
  int decrement(){
  }
  int increment(){
    <START_BUG>
    return bar−1;
    <END_BUG>
  }
}
```

Listing 2: *abstract buggy context*

```
class <unk> {
  int i = 0;
  int <unk>;
  <unk> (int <unk>){
  }
  int <unk>(){
  }
  int increment(){
    <START_BUG>
    return <unk>−1;
    <END_BUG>
  }
}
```

Listing 3: Context with <unk>

Fig. 3: Illustration of the *abstract buggy context* step in SEQUENCER. b^c is highlighted in yellow, b^m is highlighted in orange and b^l is highlighted in red.

problem by taking as input the fault localization data (*i.e.*, $l = \{l_1, l_2, ...\}$) of a buggy system and attempts to generate fixed line f_i^l for each l_i in order. The b^s, t, l, l_i, b_i^c, b_i^m, b_i^l, f_i^l and f^s notations are used throughout this work.

3.2 Buggy Context Abstraction

The *context* of a bug plays a fundamental role in understanding the faulty behavior and reasoning about the possible fix. During bug-fixing activities, developers usually identify the buggy lines, then analyze how they interact with the rest of the method's execution, and observe the context (*e.g.*, variables and other methods) in order to reason about the possible fix and possibly select several tokens in the context to build the fixed line [17].

SEQUENCER mimics this process by constructing the *abstract buggy context* and organizing the fault localization data into a representation that is concise yet retains the necessary context that allows the model to predict the possible fix. During this process SEQUENCER needs to balance two contrasting goals: (i) reduce the buggy context into a reasonably concise sequence of tokens (since sequence-to-sequence models suffer from long sentences [18]), (ii) while at the same time retaining as much information as possible to allow the model to have enough context to predict a possible fix.

Given the bug locations $l = \{l_1, l_2, ...\}$, for each $l_i \in l, l_i = \{b_i^c, b_i^m, b_i^l\}$, SEQUENCER performs the following steps:

Buggy Line <START_BUG> is inserted before the first token in the buggy line b_i^l and <END_BUG> is inserted after the last token. The rationale is that we would like to propagate the information extracted by the fault localization technique and indicate to the model what is a buggy line. In doing so, we mimic developers who focus on the buggy lines during their bug-fixing activities.

Buggy Method The remainder of the buggy method b_i^m is kept in the representation. The rationale is that the method provides crucial information on where the buggy line is placed and its interaction with the rest of the method.

Buggy Class From the buggy class b_i^c we keep all the instance variables and initializers, along with the signature of the constructor and non-buggy methods even if they are not called in the buggy method. The body of the non-suspcious methods is stripped out. The rationale for this choice is that the model could use variables and method signatures as potential sources when building the fixed line f_i^l.

After these steps, SEQUENCER performs tokenization and truncation to create the *abstract buggy context*. Truncation is used to limit the *abstract buggy context* to a predetermined size in cases where the input sequence is too long. This allows SEQUENCER to process input files of arbitrary size without running out of memory. The truncation process can be summarized as: 1) the truncation size will be chosen such that most input files do not require truncation 2) if the buggy line itself is over the truncation limit, as many tokens as possible from the start of the line are included up to the limit 3) otherwise, the buggy line is included in *abstract buggy context* and twice as many tokens are included before the line as after the line. For example, if the truncation limit is 1,000 tokens and a 5,000 token file has a buggy line with 100 tokens (including the START_BUG and END_BUG tokens) in the middle of the file, then *abstract buggy context* will consist of 600 tokens before the buggy line, then 100 tokens of the buggy line, then 300 tokens after the buggy line. Generally, truncation will delete the actual class definition from the input, but context near the buggy line is preserved to aid in patch generation.

The *abstract buggy context* represents the input to the sequence-to-sequence network which will be used to predict the fixed line. Internally, *abstract buggy context* is represented as a sequence of tokens belonging to a vocabulary V. The out-of-vocabulary tokens ($token \notin V$) are replaced with the unknown token `<unk>`. In Section 3.6 we describe how we empirically derive the vocabulary V and in Section 3.3.1 we explain how the copy mechanism helps in overcoming the unknown tokens problem.

Figure 3 shows the output of this process. The original class is presented in Listing 1 and Listing 2 displays the buggy class after Buggy Context Abstraction. Listing 3 illustrates the class when tokens that are out of vocabulary are replaced with the unknown token <unk>. Programming language tokens such as `class` and `int` are not replaced with <unk> because they are part of the vocabulary. Other in-vocabulary tokens include common variable names such as `i`. Our sequence-to-sequence network receives Listing 2 as input.

3.3 Sequence-to-Sequence Network

In this phase we train SEQUENCER to learn how to generate a fix for a given bug. Specifically, we train a Sequence-to-Sequence Network with Encoder-Decoder model (with attention and copy mechanism) to translate the *abstract buggy context* of a bug to the corresponding target fixed line f_l. To train such a network we rely on a large dataset of bug fixes mined from different sources, explained in Section 4.3. The bug fixes are divided into training and testing data, which are used to train and evaluate the Sequence-to-Sequence Network described in Section 3.3.1.

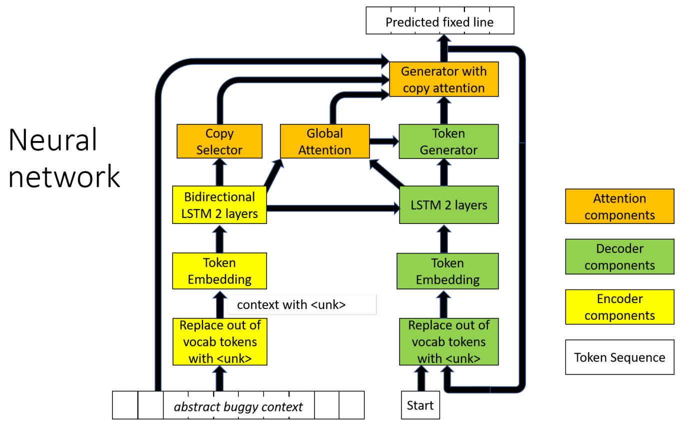

Fig. 4: Sequence-to-sequence model used in SEQUENCER.

3.3.1 Model

Figure 4 shows our model for sequence-to-sequence learning to create Java source code patches. The basis of our model is a recurrent neural network similar to a natural language processing architecture [5]. During training, the source token sequence $X = [x_1, ..., x_n]$ (*i.e., abstract buggy context*) is provided to the encoder, where n is the token length of *abstract buggy context*. Then, the decoder produces the target sequence $Y = [y_1, ..., y_m]$ (*i.e.,* the fixed line), where m is the token length of the fixed line. Back propagation is used to update the parameters in the network with stochastic gradient decent during training [19]. The trained parameters are unchanged during inference (patch generation in our case).

Encoder The encoder is a recurrent neural network using LSTM gates to process the input [20]. It is a bidirectional encoder which allows the encoding for a token to incorporate information from other tokens both before and after it in the input data [21]. The encoder converts the source sequence $X = [x_1, ..., x_n]$ into a sequence of encoder hidden states h_i using a learnable recurrence function g_e. After reading the last token, the last hidden state, h_n^e is used as the context vector c for use in initializing the decoder [22]:

$$h_i^e = g_e(x_i, h_{i-1}^e); \tag{1}$$

Decoder The decoder is also a recurrent neural network using LSTM gates. When initialized by the encoder, it begins production of the patch candidate by receiving the special *start* token as input y_0. For each previous output token y_{j-1}, the decoder updates its hidden state h_j^d using the learnable recurrence function g_d [22]:

$$h_j^d = g_d(y_{j-1}, h_{j-1}^d, c) \tag{2}$$

The initial value h_0^d is provided by a learnable bridge function of the encoder state. The decoder states h_j^d are used in for token generation by the attention and copy mechanisms in Equation 4 and Equation 5. The decoder stops generating new tokens when the last token generated by the model is a special end-of-sequence token.

Attention In addition, we use an attention mechanism that provides a way to create a more specific context vector c_j for each output token y_j from the decoder using a linear combination of the hidden encoder states h_i^e [23]:

$$c_j = \sum_{i=1}^{n} \alpha_i^j h_i^e \tag{3}$$

Where α_i^j represents learnable attention weights. This context vector c_j is used by a learnable function g_a to allow each output token y_j to pay "*attention*" to different encoder hidden states when predicting a token from vocabulary V:

$$P_V(y_j \mid y_{j-1}, y_{j-2}, ..., y_0, c_j) = g_a(h_j^d, y_{j-1}, c_j) \tag{4}$$

Copy In Section 2 we presented the intuition behind the copy mechanism, while in this section we describe how it operates during patch generation. The copy mechanism can significantly improve the performance of the system by allowing the model to select a token from any of the tokens provided in the *abstract buggy context*, even when the tokens are not contained in the training vocabulary. We empirically show the improvements offered by this approach by comparing it to the vanilla sequence-to-sequence model without a copy mechanism in Section 4.4.2. The copy mechanism contributes to Equation 4 to produce a token candidate. This component calculates p_{gen}, the probability that the decoder generates a token from its initial vocabulary. And $1 - p_{gen}$ is the probability to copy a token from the input tokens depending on the attention vector α^j in Equation 3 [9]:

$$p_{gen} = g_c(h_j^d, y_{t-1}, c_j) \tag{5}$$

$$P(y_j) = p_{gen} P_V(y_j) + (1 - p_{gen}) \sum_{i:x_i=y_j} a_i^j \tag{6}$$

g_c in Equation 5 is learnable function. Using Equation 6, the output token y_j for the current decoder state is selected from the set of all tokens that are either: 1) tokens in the training vocabulary (including the <unk> token) or 2) tokens in the *abstract buggy context*. Although there are no <unk> targets in the training set for patches, if the P_V computation is very uncertain which token is correct, it may happen to have a high likelihood for <unk>. If at the same time, p_{gen} is high then a <unk> token will be produced as the copy mechanism did not replace it. Such outputs are discarded as discussed in Section 3.5.

3.4 Patch Inference

Once the sequence-to-sequence network is trained, it can be used to generate patches for projects outside of the training dataset. During patch inference, we still generate *abstract buggy context* for the bug, as described in Section 3.2. But we will use beam search to generate multiple likely patches for the same buggy line, as done in related work [10], [24]. Beam search works by keeping the n best sequences up to the current decoder state. The successors of these states are computed and ranked based on their cumulative probability; and the next n best sequences are passed to next decoder state. n is often called the width or beam size, and beam search with an infinite n corresponds to doing a complete breath-first-search. In Listing 5, we have an example of predictions with beam size 5 for the bug presented in Listing 2. Each row is one prediction from the model, representing one potential bug fix, and each of them is further processed by the patch preparation step described below.

```
1 return 1 ;
2 return i ;
3 return <unk> ;
4 return <unk> + 1 ;
5 return <unk> . <unk>;
```

Listing 4: Without copy mechanism

```
return 1 ;
return i ;
return <unk> ;
return bar + 1 ;
return Foo . bar ;
```

Listing 5: Network output

```
return 1;
return i;
// discarded
return bar+1;
return Foo.bar;
```

Listing 6: After patch preparation

Fig. 5: Patch preparation step using copy mechanism

3.5 Patch preparation

The raw output from the sequence-to-sequence network cannot be used as a patch directly. First, the predictions might still contain <unk> tokens not handled by the copy mechanism. Listing 4 illustrates token values before the copy mechanism replaces <unk> for samples 4 and 5. But the copy mechanism may not replace all such tokens as seen in sample 3 of Listing 5. Second, the predictions contain a space between every token, which is not well-formed source code in many cases. (For example, a space is not allowed between the dot separator, ".", and a method call, but a space is required between a type and the corresponding identifier name.)

Consequently, we have a final patch preparation step as follows. We discard all line predictions that contain <unk> and we reformulate the remaining predictions into well-formed source code by removing or adding the required spaces. An example is shown between Listing 5 and Listing 6, whitespaces are adjusted and the third prediction from Listing 5 is removed since it contains <unk> token. Each one of the line predictions is used to create a candidate program by replacing the original buggy line b_i^l (*i.e.*, the <START_BUG>, <END_BUG> and all tokens in between are replaced with the model output).

More formally, the remaining candidate fixed lines, $cand_i = \{pre_i^1, pre_i^2, ..\}$, will replace the buggy line b_i^l in buggy system b^s and generate candidate patches $\{patch_i^1, patch_i^2, ...\}$, which should be verified with any patch validation technique, such as test suite validation. When the test suite is weak to specify the bug, we can have different patches $\{patch_i^1, patch_j^1, ...\}$ for different bug locations $\{l_i, l_j, ...\}$ that passed the test suite. Then, the correctness can be verified, for example, by manual inspection.

3.6 Implementation Details & Parameter Settings

Library. We have implemented our Encoder-Decoder model using OpenNMT-py [25], built in the Python programming language and the PyTorch neural network platform [26].

Vocabulary In this paper, we consider a vocabulary of the 1,000 most common tokens. To the best of our knowledge, this is one of the largest vocabularies considered for machine learning for patch generation: for comparison, DeepFix [27] has a vocabulary size of 129 words, and Tufano *et al.* [10] considered a vocabulary size of 430 words.

Limit for truncation We truncate if the *abstract buggy context* is longer than 1,000 tokens. It is motivated by Figure 7, where we can see the most of *abstract buggy context* are less than 1,000 tokens long. SEQUENCER truncates by removing statements, class definitions, and method definitions until *abstract buggy context* is 1,000 tokens or less, but keeping the buggy line within the truncated buggy class.

Network parameters We explored a variety of settings and network topologies for SEQUENCER. Most major design decisions are verified with ablation experiments that change a single variable at a time as detailed further in Section 5. We train our model with a batch size of 32 for 10,000 iterations. To prevent overfitting, we use a dropout of 0.3. In relation to the components shown in Figure 4, below are the primary matrix sizes associated with each component along with a reference to the equations in Section 3.3.1 to which they relate:

- Token embedding (our model uses the same embedding for both g_e and g_d): 1,004x256 (1,000 + 4 special tokens)
- Encoder bidirectional LSTM (part of g_e fuction): 256x256x4x2x2
- Decoder LSTM (part of g_d function): 512x256x4x2 + 256x256x4x2
- Token generator (part of g_a function): 256x1004
- Bridge between encoder and decoder (path for h_i^e to initialize h_0^d): 256x256x2
- Global Attention (α_i^j weights): 256x256 + 512x256
- Copy selector (g_c function): 256x1

We use a beam size of 50 during inference, which is the default value used in the literature [10][24] and which proves to be good empirically.

Input and output summary The input to SequenceR is a Java class of any size. The non-empty faulty line within a method on which to attempt repair has been identified by another technique (usually line-based fault localization). The output is the fixed line which must have fewer than 100 tokens with our current model.

Usage After SEQUENCER is trained, we can use it to predict fixes to a bug. SEQUENCER takes as input the buggy file and a line number indicating where the bug is. The output is a list of patches in the diff format, so that the user can run their own patch validation step, which could either be test validation or manual inspection.

The source code of SEQUENCER is available at https://github.com/kth/SequenceR, together with the best model we have identified and the synthesized patches.

4 EVALUATION

In this section, we describe our evaluation of SEQUENCER.

4.1 Research Questions

The two first research questions focus on machine learning:

- RQ1: To what extent can the fixed line be perfectly predicted?
- RQ2: How often does the copy mechanism generate out-of-vocabulary tokens for a patch, and which parts of *abstract buggy context* are referenced for the copy?

The last two research questions look at the system from a domain-specific perspective: we assess the performance of SEQUENCER from the viewpoint of program repair research.

- RQ3: How effective is SEQUENCER's sequence-to-sequence learning in fixing bugs in the well-established Defects4J benchmark?
- RQ4: What repair operators are captured with sequence-to-sequence learning?

4.2 Experimental Methodology

4.2.1 Methodology for RQ1

We train SEQUENCER with the parameter settings described in Section 3.6. The training and validation accuracy and perplexity will be plotted. Perplexity (ppl) is a measurement of how well a model predicts a sample and is defined as:

$$ppl(X, Y) = exp\left(\frac{-\sum_{i=1}^{|Y|} \log P(y_i \mid y_{i-1}, \ldots, y_1, X)}{\mid Y \mid}\right)$$

where X is the source sequence, Y is the true target sequence and y_i is the i-th target token [25]. Luong *et al.* found a strong correlation between a low perplexity value and high translation quality [28].

The resulting model is tested on our testing dataset, CodRep4 (see Section 4.3.1). Next, in order to compare SEQUENCER against the state-of-the-art approach by Tufano *et al.* [10], we created CodRep4Medium. It is a subset of CodRep4 containing 1,116 samples where the buggy method length is limited to 100 tokens.

4.2.2 Methodology for RQ2

To evaluate the effectiveness of the copy mechanism (described in Section 3.3.1), we consider all samples from CodRep4. For each successfully predicted line, we categorize tokens in that line based on whether the token is in the vocabulary or not. And at the same time, for tokens that are out-of-vocabulary but are copied from the input sequence, we try to find the original location of the copied token. By analyzing the original location of out-of-vocabulary tokens, we can measure the importance of the context, in particular of the *abstract buggy context* we define in this paper. The copy mechanism allows the system to be more powerful by providing more tokens beyond the vocabulary to be used in the patch.

4.2.3 Methodology for RQ3

We evaluate SEQUENCER on Defects4J [16], which is a collection of reproducible Java bugs. Most recent approaches in program repair research on Java use Defects4J as an evaluation benchmark [3], [12], [29]–[31].

Since the scope of our paper is on one-line patches, we first focus on Defects4J bugs that have been fixed by developers by replacing one single line (there are 75 such bugs). In order to study the effectiveness of sequence-to-sequence itself, we isolate the fault localization step as follows: the input to SEQUENCER is the actual buggy file and the buggy line number. SEQUENCER then produces a list of patches (recall that beam search produces several candidate patches). All patches are compiled and then executed against the test suite written by the developer.

Each candidate patch generated by SEQUENCER is then categorized as follows:

- **Compilable patch**: The patch can be compiled.
- **Plausible patch**: The patch is compilable and passes the test suite. The patch may yet be incorrect because of the overfitting problem [32].
- **Correct patch**: The patch passes the test suite, and is semantically equivalent to the human patch. We hand-check for semantic equivalence for this evaluation.

As per the definitions, there is a strict inclusion structure in those categories: correct patches are necessarily plausible and compilable, plausible patches are necessarily compilable.

4.2.4 *Methodology for RQ4*

For RQ4, we aim at having a qualitative understanding of the cases for which our sequence-to-sequence repair approach works. This research question is motivated by the need to understand what grammatically correct code transformations are captured by SEQUENCER, even though it is purely a token-based approach with no first class AST or grammar knowledge. For gaining this understanding, we use a mixed method combining grounded theory and targeted analysis. The results would be an understanding of the variety of repair operators and programming language syntax captured by SEQUENCER in cases where the model output correctly matches the test data. For the grounded theory, we have been regularly sampling successful cases, *i.e.*, cases in our testing dataset CodRep4 for which SEQUENCER was able to predict the fixed line, for each case, the authors reached a consensus to know whether 1) the case is interesting from a programming perspective (*e.g.*, it represents a common bug fix pattern), and 2) the case highlights a phenomenon that has already been covered in a previously found case. For the targeted analysis, we specifically searched for 2 kinds of results: cases where the copy mechanism was used and cases where a specific programming construct was involved (method call, field reference and string literals).

4.3 Training Data

SEQUENCER is trained based on past modifications made to source code, *i.e.*, it is trained on past commits. In our experiments, we combine two sources of past commits, the CodRep dataset [33] and the Bugs2Fix dataset [10], into what appears to be the largest dataset of one-line bug fixes published to date. Both datasets 1) consider Java code and 2) have been built based on the history of open-source projects.

The CodRep dataset focuses solely on one-line source code fixes (aka one-line patches), it contains 5 datasets curated from real commits on open-source projects. The Bugs2Fix dataset contains diffs mined from Github between March 2010 and October 2017 for bug-fixing commits (based on heuristics to only consider bug-fixing commits). Neither dataset requires the buggy project to have a test suite for exposing the buggy behavior, instead they are focusing on collecting bug fix commits.

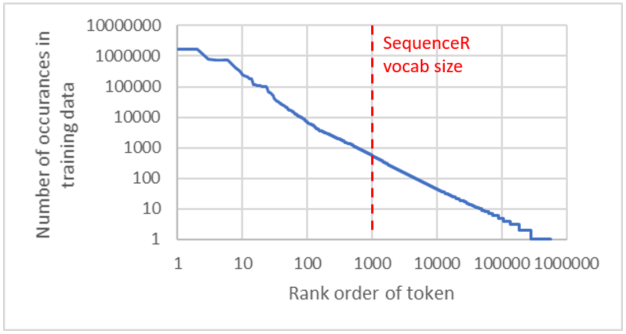

Fig. 6: Overview of vocabulary: token count occurrences follow a Zipf's law distribution.

4.3.1 *Data Preparation*

Since CodRep and Bugs2Fix datasets are in different formats, we first unify these two datasets as follows. First, we only keep diffs from Bugs2Fix which are fixes with a single line replacement. Further, we filter out certain diffs if the changes are outside of a method.

Since the Bugs2Fix dataset comes from a generic bug-fix data mining which includes multi-line fixes and fixes outside of methods, we can look at its statistics to help understand the generality of SEQUENCER. Bugs2Fix contains 92,849 commits. 15,548 of these (17%) are one-line patches within a method, and are within the problem domain of SEQUENCER.

After preparing the dataset, we divide it into training and testing data. CodRep is originally split into 5 parts, numbered from 1 to 5, with each part containing commits from different groups of projects. Our training data consists of CodRep datasets 1,2,3 & 5 and the Bugs2Fix dataset. Our testing data is CodRep dataset 4 (or CodRep4 for short). We chose dataset 4 because it is approximately 20% of the entire CodRep data (data set 1 is less than 10% and data set 5 is over 30%) and because CodRep 4 contains a broad and representative set of projects on which to evaluate [33].

Furthermore, we ensure there are no duplicate samples between the training and testing datasets. During the model setup, we use a random subset of 95% of the training data for model training and 5% as our validation dataset.

4.3.2 *Descriptive Statistics of the Datasets*

In total, we have 35,578 samples in our training set and 4,711 samples in our testing set.

Input Size Figure 7 shows the size distribution of the *abstract buggy context* in number of tokens before truncation is done. The CodRep training data has a median token length of 372; the Bugs2Fix dataset has a median length of 340 tokens; and the testing dataset has a median length of 411. These variations are a result of using different Java projects in the datasets, but we observe that the distribution of lengths is similar.

Prediction Size The lines from the *abstract buggy context* samples in our dataset had a median length of 6. 99% of the lines were 30 tokens or fewer, which fits well typical output

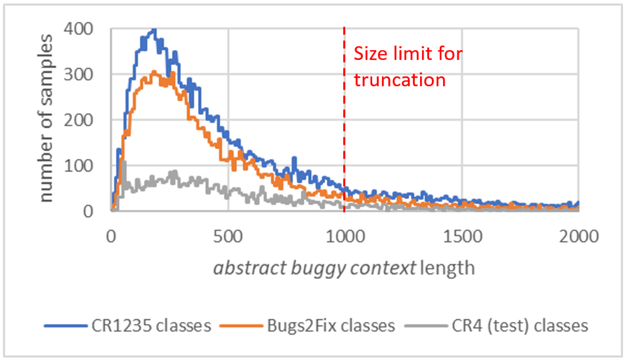

Fig. 7: Only 14% of samples exceed the 1K token length limit and require truncation.

Approach	Prediction Accuracy	
	CodRep4Medium	CodRep4
simple seq2seq line2line, no copy	77/1116 (6.9%)	206/4711 (4.4%)
Tufano *et al.* [10]	157/1116 (14.1%)	N/A
SEQUENCER	344/1116 (30.8%)	950/4711 (20.2%)

TABLE 1: Comparison with state-of-the-art approach by Tufano *et al.*

sizes used for natural language processing. To sum up, the order of magnitude of the sequence-to-sequence prediction receives an input sequence with an average length of 350 tokens and produces an output sequence with an average length of 6 tokens.

Vocabulary Size In our training data, the full vocabulary is 567,304 different tokens. Figure 6 shows the distribution of the number of occurrences for the whole vocabulary. It is a typical power-law like distribution with a long tail. We limit our training vocabulary to the 1,000 most common tokens.

4.4 Experimental Results

4.4.1 Answer to RQ1: Perfect Predictions

We trained our model on a GPU (Nvidia K80) for 1.2 hours. For a typical training run on our golden model, Figure 8 shows the training and validation accuracy per token generated (the accuracy for the entire patch would be lower) and Figure 9 shows the perplexity (ppl) per token generated over the training and validation datasets. In this particular run, the best results for both the perplexity and accuracy on the validation dataset occur at 10,500 iterations. We chose 10,000 iterations as the standard training time for our model.

CodRep4 On the 4,711 prediction tasks of our best model, SEQUENCER is able to generate the perfect fix in 950 cases (from Table 1). In all those cases, the predicted line that replaces the buggy line is exactly the line fix implemented by the developer. The copy mechanism is used in a number of cases, this will be further discussed in subsubsection 4.4.2.

Comparison to state-of-the-art To the best of our knowledge, the state-of-the-art approaches are from Tufano *et al.* [10] and Hata *et al.* [34]. We only compare against Tufano *et al.* since their approach has been open sourced while that one of Hata *et al.* was not made available at the time of writing this paper. The approach used by Tufano *et al.* is limited to fixes only inside small methods, consisting of less than 100 tokens. The limitation is due to the fact that their approach generates the entire fixed source code method as output of the decoder. This means that the decoder may need to generate a long sequence of source code tokens, which is one of the major challenges for NMT models [35]. SEQUENCER does not make any assumption on the size of the buggy method. In order to compare against [10], we select those 1,116 tasks from CodRep4 where the buggy line resides in a method smaller than 100 tokens. Those 1,116 tasks are called the CodRep4Medium testing dataset.

Our testing accuracy for both CodRep4 and CodRep4Medium are shown in Table 1. From the table, we see that the accuracy of SEQUENCER is 344/1,116 (30.8%) while Tufano *et al.* [10] is 157/1,116 (14.1%). This is a clear indicator that SEQUENCER outperforms the current state-of-the-art showing twice as many correct predictions. It shows that our construction of the *abstract buggy context*, together with the copy mechanism, leads to higher accuracy than only having the buggy method as context with a specific encoding for variables. Recent fault localization research [15] indicates that best-in-class techniques can predict the faulty line 44% of the time and the faulty method 68% of the time. If we extrapolate these percentages to our data, SEQUENCER is more likely to find correct one-line patches than the prior work [10] is to find method replacements, and SEQUENCER can process and repair larger methods as demonstrated by the right-hand column of Table 1.

We now concentrate on the effectiveness of the approach depending on the buggy method length. Overall, we observe that SEQUENCER has a lower accuracy on longer methods (30.8% accuracy on CodRep4Medium, 20.2% accuracy on CodRep4). This phenomenon is explained by the fact that fixes in long methods are usually more complex and involve more context variables, identifiers and literals that are not easily captured by the learning system. This phenomenon has also been previously observed [10].

4.4.2 Answer to RQ2: Copy Mechanism

We now look at to what extent the copy mechanism is used. Figure 10 shows the origin of tokens in successfully predicted lines, per patch size. Let us consider the highest bar, corresponding to all successfully predicted lines consisting of 7 tokens. For those 7-token patches, the black bar means that all tokens are taken from the vocabulary. The non-black bars mean that the copy mechanism has been used to predict the line fix. Overall, there is a minority of patches (216/950, 23%) for which all tokens come from the vocabulary. At the extreme, the longest successful patch generated by SEQUENCER was 68 tokens long, but the longest successful patch without the copy mechanism was only 27 tokens long.

Figure 10 also lets us analyze the location origin of the copied token. The brown bars represent those patches for which copied tokens all come from the buggy line: this is the majority of cases (641/950, 68%). However, we also observe cases where some copied tokens have been taken from the buggy method (green bars) and cases where the copied tokens has been taken from the buggy class (red bars), *i.e.*, taken from the class context as captured in our encoding.

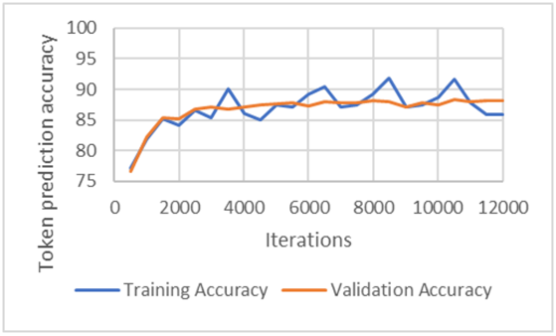

Fig. 8: Training and validation accuracy

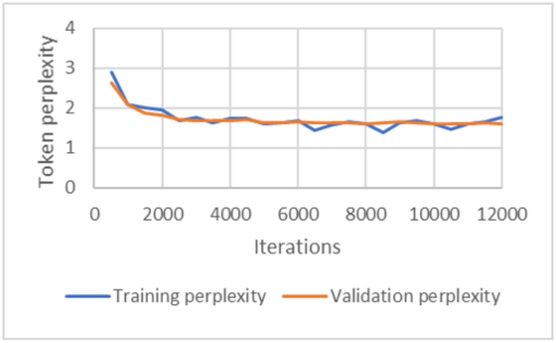

Fig. 9: Training and validation perplexity

As an example, Listing 7 replaces variable `masterNode` with `nonMasterNode` as in the correct human patch. `nonMasterNode` in the fixed line does not occur in our training data and hence it is not in our 1000 token vocabulary. Therefore, SEQUENCER was able to generate this patch because it copied the out-of-vocabulary token `nonMasterNode` from within the buggy method. As this example is a 4 token long patch, it would contribute to the green bar for patch length 4 in Figure 10.

```
while( nonMasterNode == null ) {
  nonMasterNode=randomFrom( internalCluster().getNodeNames());
   if ( nonMasterNode.equals( masterNode ) ) {
–       masterNode = null;
+       nonMasterNode = null;
   }
}
```

Listing 7: Example of the copy mechanism creating a correct patch by incorporating a variable which is not in the vocabulary from the broader context around the buggy line.

Overall, Figure 10 shows that the copy mechanism is extensively used (734/950, 77%) and that our class level abstraction enables us to predict difficult cases where only the buggy line or the buggy method would not have been enough.

In order to understand the benefits of context size with the copy mechanism, we measured the distance in tokens to reach a copied token used to generate a patch. In the 87 cases where a copied token was needed from the buggy method b_m, the median distance from the buggy line b_l to the nearest use of the copied token was 9 tokens, 90% of the 87 cases were within 49 tokens of b_l, and 100% were found within a 122 token distance. In the 7 cases when a copied token was needed from the buggy class b_c, the median distance to the copied token from b_m was 25 tokens, and 100% were found within a 241 token distance. In addition to ablation study results discussed is Section 5, the preceding data supports our decision to create the *abstract buggy context*.

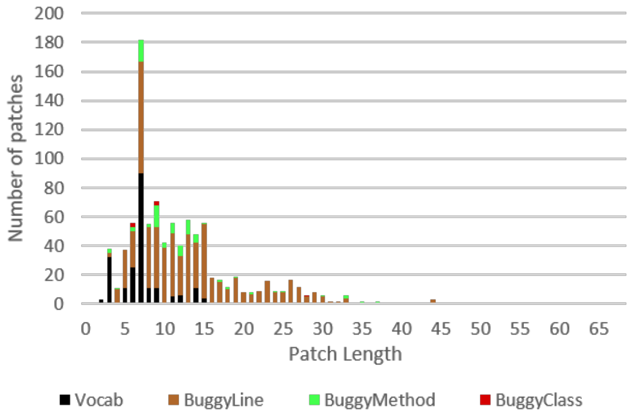

Fig. 10: Histogram showing correctly generated patches: 1) that only use tokens in our 1,000 token vocabulary, 2) that need to copy tokens from the buggy line, 3) from the buggy method and 4) from the buggy class.

4.5 Answer to RQ3: Defects4J Evaluation

As explained in Section 4.2.3, we consider 75 Defects4J bugs that have been fixed with a one-line patch by human developers. In total SEQUENCER finds 2,321 patches for 58 of the 75 bugs. The main reason that we are unable to fix the remaining 17 bugs is due to fact that some bugs are not localized inside a method, which is a requirement for the fault localization step that SEQUENCER assumes as input. Listing 8 is one such example where the Defects4j bug is not localized inside a method. We have 2,321 patches instead of 2,900 (58x50) because some predictions are filtered by the patch preparation step (Section 3.5), *i.e.*, patches that contain the <unk> token. The statistics about all bugs can be found in Figure 11. Out of 75 bugs, SEQUENCER successfully generated at least one patch for 58 bugs, 53 bugs have at least one compilable patch, 19 bugs have at least one patch that passed all the tests (*i.e.*, are plausible) and 14 bugs are considered to be correctly fixed (semantically identical to the human-written patch). Of these 14 bugs, in 12 cases the plausible patch with the highest ranking in the beam search results was the semantically correct patch.

```
– private  static  final  double DEFAULT_EPSILON = 10e−9;
+ private  static  final  double DEFAULT_EPSILON = 10e−15;
```

Listing 8: An example of Defects4J defect (Math 104) where the bug is not localized inside a method. In this case, a class variable is changed.

Figure 12 gives a different perspective on this data, focusing on patches (and not bugs). SEQUENCER is able to generate 761 compilable patches (33% of all patches). SEQUENCER finds 61 plausible patches spread over 19 bugs, thus there can be several plausible patches for the same bug, a phenomenon well-known in the program repair field [12]. One reason is that some Defects4J bugs have a weak test suite. To the best of our knowledge, we are the first to

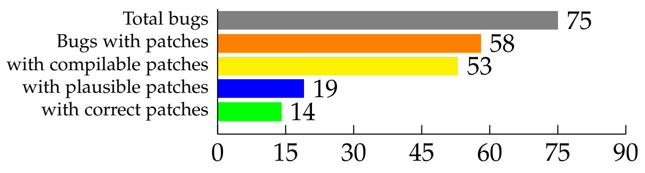

Fig. 11: SEQUENCER results on the 75 one-line Defects4J bugs.

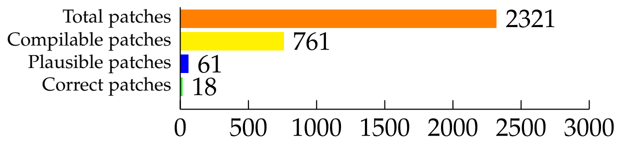

Fig. 12: Stastistics on patches synthesized by SEQUENCER for the 75 one-line Defects4J bugs.

report the correctness of patches generated by a sequence-to-sequence model, where correctness means passing the test suite and being semantically equivalent to the human patch. In the end, SEQUENCER is able to generate 18 patches that are semantically equivalent to the correct bug fix.

For SEQUENCER applied to Defects4J bugs, we observe that out of 61 plausible patches, 18 are correct, which is a ratio of 30%. An analysis of prior techniques which used a different benchmarck in C (GenProg [36], RSRepair [37], and AE [38]) shows that they have a correct patch ratio of less than 12% [11]. We did not evaluate SEQUENCER on the same benchmark as this prior work (we target Java not C), but the ratio is evidence that SEQUENCER has learned to produce outputs which represent reasonable patch proposals.

Although we did not directly include fault localization in our evaluation of SEQUENCER, we can estimate the performance of a repair system which includes state-of-the-art fault localization techniques [15] as follows. It has been shown that there is an estimated 44% success of correctly identifying a faulty line in the top 10 candidates. Hence, in order to process 75 total bugs from Defects4J, 750 candidate abstract buggy contexts would need to be prepared for input to our model. We have run fault localization with Gzoltar [39] and found that it successfully localized the faulty line for 9 of the 14 bugs for which SEQUENCER found a correct fix.

Let us now discuss timing. We estimate the machine time required to automatically find patches for 75 bugs with the summation below[1]:

- Estimated time to run fault localization on 75 bugs and identify 10 likely faulty line locations: 112 minutes
- Time to create 750 *abstract buggy contexts* (10 created for each bug): 29 minutes
- Time to create 37,500 patch candidates (50 candidates created from beam size 50 for each *abstract buggy context*): 9 minutes
- Estimated time to prune raw patches down to 23,210 total patches: 2 minutes
- Time to attempt compile on 23,210 patches: 1378 minutes
- Time to run test cases on 7,610 patches: 6287 minutes
- Final result estimated to take 130 total machine hours to find patches which correctly fix 9 bugs.

Listing 9 shows the SEQUENCER patch for Math 75, which is semantically equivalent to the human patch. We observe that it contains some unnecessary parentheses, and the same behavior occasionally occurs in other patches found by SEQUENCER. We have observed unnecessary parenthesis in some of the human-generated patches in our training data and SEQUENCER occasionally replicates this human style. In this case, the parentheses do not change the order of evaluation. Therefore the SEQUENCER patch for Math 75 is semantically equivalent to the human patch.

Interestingly, `getPct` is not part of the vocabulary, and it did not appear in the buggy method. The `getPct` method is defined in the same buggy class, as captured by our *abstract buggy context*. In Defects4J, the copy mechanism is also useful to capture the right tokens to add in the patch.

```
− return getCumPct((Comparable<?>) v);
+ return getPct((Comparable<?>) v); // Human patch
+ return getPct(((Comparable<?> )(v))); // SEQUENCER patch
```

Listing 9: Found patch for Math 75

We now compare those results against the patches found by recent program repair tools that are publicly available. Elixir [4], CapGen [3] and SimFix [31] have reported 26, 22, 34 correctly repaired bugs for all Defects4J bugs, where the patch is identical to the human patch or claimed as correct. Of those correctly repaired bugs, 22, 19 and 17 respectively are for the 75 one-line bugs that we consider for SEQUENCER. We notice that the majority of claimed correct patches are for one-line bugs. We observe that SEQUENCER does not fix more one-line Defects4J bugs.

While Elixir, CapGen, and SimFix are driven with intelligent design and require substantial configuration and handcrafted rules, our goal with SEQUENCER is to be agnostic and to *not* design any repair operator upfront. For example, CapGen implements context-aware operator selection and context-aware ingredient prioritization [3]. The CapGen implementation heavily relies on code transformation tools and carefully selected algorithms/parameters/-metrics. In constrast, our SEQUENCER can be considered less heavyweight. We note that the required parameter tuning in SEQUENCER can easily be performed using grid search or other meta-optimization techniques [40]. To that extent, it is remarkable that such a generic approach is able to learn bug-fixing patterns and synthesizes 18 patches that are semantically equivalent to the human repair, without any static or dynamic analysis. By providing a generic approach, SEQUENCER will improve in the future as machine learning sequence-to-sequence techniques improve, and as more bug fix training data is provided. Also, since SEQUENCER learns repair operators from examples, it could be trained on less common languages (such as COBOL).

We assume perfect fault localization while other related tools ran fault localization to localize the buggy source code. Yet, different papers use different fault localization algorithms, implementations, and granularity (*e.g.*, methods versus line). Liu *et al.* pointed out that because of different

1. Our Defects4J testing was run on an Intel Core i7 at 3.5GHz and our sequence-to-sequence model was run on an Nvidia K80.

assumptions about fault localization, it is hard to compare different repair techniques [41]. By assuming perfect fault localization, we purely focus on the patch generation step of the algorithm.

4.6 Answer to RQ4: Qualitative Case Studies

We now present the diversity of repair operators that are captured by SequenceR. These cases are culled from the 950 correct patches SequenceR generated for the CodRep4Full test dataset. Both the buggy line that was part of the input is shown and the correct patch which includes examples of repair operators. We also highlight again the effectiveness of the copy mechanism by using a **bold underlined** font for those tokens that were copied (*i.e.,* that are outside the vocabulary of the 1,000 most common tokens).

4.6.1 *Case study: method call change*

Our training and evaluation data consist of object-oriented Java software. We observe that SequenceR captures different kinds of operations related to method calls.

Call change Here a call to method **writeUTF** is replaced by a call to method **writeString**.

```
− out.writeUTF( failure );
+ out.writeString( failure );
```

Listing 10: Call change

Call deletion The buggy line chains two method calls; this successful prediction consists of deleting one of them.

```
− FieldMappers x = context.mapperService() .
      smartNameFieldMappers( fieldName );
+ FieldMappers x = context.smartNameFieldMappers( fieldName );
```

Listing 11: Call deletion.

Argument addition In this patch, SequenceR adds an argument (which in Java, means calling another method).

```
− stage.getViewport() .update( width, height );
+ stage.getViewport() .update( width, height, true );
```

Listing 12: Argument addition

Target change In this successful case, the patch also calls method **isTerminated** but on another target (**scheduledExecutorService** instead of **executorService**, which is copied from the input context).

```
− if( !( executorService.isTerminated() ) ){
+ if( !( scheduledExecutorService.isTerminated() ) ){
```

Listing 13: Target change

4.6.2 *Case study: if-condition change*

SequenceR can change if conditions, and in this particular case, removes two clauses from the boolean formula.

```
− if( ( ( t >= 0 ) && ( t <= 1 ) ) && ( intersection != null ) )
+ if( intersection != null )
```

Listing 14: if-condition change

4.6.3 *Case study: Java keyword change*

SequenceR is also able to generate patches involving the replacement of programming language keywords, indicating clues of syntax understanding.

```
− break ;
+ continue ;
```

Listing 15: Java keyword change

4.6.4 *Case study: change from field access to method call*

A good practice of software engineering is to implement encapsulation by calling methods instead of directly accessing fields, this is handled by SequenceR as follows (`size` to `size()`)

```
− app.log( "PixmaPackerTest", ( "Number of textures: " + ( atlas.
      getTextures() .size ) ) );
+ app.log( "PixmaPackerTest", ( "Number of textures: " + ( atlas.
      getTextures() .size() ) ) );
```

Listing 16: change from field access to method call

4.6.5 *Case study: off-by-one repair*

Finally, SequenceR is also able to repair classical off-by-one errors.

```
− nextIndex = currentIndex;
+ nextIndex = ( currentIndex ) − 1;
```

Listing 17: off-by-one repair

Overall, SequenceR uses all three kinds of token operations: 1) Token deletion, *e.g.,* Listing 11; 2) Token addition, *e.g.,* Listing 12; 3) Token replacement, *e.g.,* Listing 10.

5 Ablation Study

We perform an ablation study to understand the relative importance of each component of our approach. The process is as follows. First, we identify the golden model based on a greedy optimization in the parameter search space. This is the model that we described in section 4. Then we change one single parameter to a different reasonable value and report the performance on the same testing dataset. The ablation results demonstrate that parameter selections for the golden model produce the highest acceptance rates for the configurations we tested. The model parameters we found with our dataset are likely to yield reasonable results when training for other computer languages so long as a form of *abstract buggy context* can be done to provide context related to the buggy line. We provide details on our ablation results to aid future researchers in understanding which variables are most likely to improve their own models.

Due to randomness in learning, for each parameter, we run each configuration multiple times and report the mean and standard deviation for the model as recommended for assessment of random algorithms [42]. As our goal is to select the best model for use in our Defects4J evaluation, we use the test set from CodRep4Full to select the best run of each model, hence we report the percentage decrease of the best run for a given model from the best result found with the golden model. Due to computational constraints, we only run each model 10 times; for the 18 configurations reported, almost 200GB of disk storage was used and 400 machine-hours. When using SEQUENCER to learn new datasets, we would recommend a similar approach where a validation set is used to select the best performing model after multitple training runs.

First, we consider the very coarse grain features. Table 2 shows the performance of four models, starting from a simplistic seq-to-seq model that only takes a single buggy line b_l as input when learning to produce the fixed line f_l. Then we show beam search, copy, and the use of the *abstract buggy context* improving the model performance. These results confirm our answer to RQ2 that the copy mechanism is essential to the performance of the system.

Second, Table 3 shows the results of our 'Golden model' against the results of single specific, targeted changes made to the model. Ablation ID 1 shows that our 10K training limit is sufficient given our training data. ID 2 shows that a vocabulary smaller than 1K tokens performs worse - likely due to a loss of learned tokens that can be used even if an instance of the token is not in the *abstract buggy context*. ID 3 shows that a vocabulary larger than 1K tokens performs worse - perhaps due to the additional tokens having insufficient training examples for learning a proper embedding. To further understand the effect of vocabulary size, we analyzed the raw output of our model before the patch preparation step. For the golden model (vocab=1000), 38% of the generated patches on CodRep4 have <unk> tokens and would be discarded; with ID 2 (700) it is 43%, and with ID 3 (1400) it is 37%. Hence, although a larger vocabulary had fewer raw <unk> tokens, the 1000 token vocabulary was able to produce better optimized models.

ID 4 is about pretraining; in order to provide more opportunities to learn a quality embedding, we created unsupervised pretraining data for the encoder/decoder. Using this unsupervised data did not improve the model, it worsened it.

ID 5 a and b show the value of combining the CodRep and Bugs2Fix data sets to improve the generalization of the model. ID 6 demonstrates the effect of removing the bridge between the encoder and decoder, which improved the mean for the model but tightened the standard deviation and hence produced a lower best result that the golden model. This is perhaps due to the bridge layer allowing for more variation in the encoder hidden state embedding and decoder hidden state embedding.

IDs 7 through 10 demonstrate that our LSTM network is sized correctly; presumably a smaller network cannot generalize on the model data well enough whereas a larger network has too many degrees of freedom. Our speculation is that a 2 layer encoder/decoder network allows the layer connected directly to the token embedding to 'focus' the weight matrix on input syntax while the layer connected to the attention/copy mechanism 'focuses' on output generation. ID 11 shows the loss in accuracy when *abstract buggy context* is reduced to just the buggy line.

ID 12 shows that truncation is necessary otherwise an out-of-memory error crashes the system, due to too many time steps being stored in memory per token in the sequence. ID 13 shows that if we truncated to 4,000 tokens then the system passes, but the increased context size (4,000 vs the golden model 1,000) did not improve accuracy of the model. ID 14 shows that using a 500 token limit for *abstract buggy context* hurts accuracy presumably because there are less opportunities for token copy. We also speculate that a possible advantage of 1K truncation instead of 500 could be that 1K provides a type of unsupervised learning for the encoder hidden states, the global attention, and the copy mechanism.

Model description	CR4Full	ratio
50K vocab, no copy, beam size 1, no context	55	baseline
50K vocab, no copy, beam size 50, no context	206	3.7x
1K vocab, copy, beam size 50, no context	826	15.0x
Golden Model (with *abstract buggy context*)	950/4711	17.3x

TABLE 2: Performance impact of the key features of beam size, copy, and context.

ID 15 removes the <START_BUG> and <END_BUG> tokens from the *abstract buggy context* input. The target output is still the correct single-line patch. Without these labels, SEQUENCER must learn line break positions and learn a type of fault localization in order to create a valid patch. Because *abstact buggy context* does not include test coverage data or other information useful for fault localization, there is a significant accuracy loss for this ID, but the network was still able to create 356 correct patches.

Our primary use case modeled in this paper is to use our golden model for SEQUENCER on projects for which it was not trained. This allows for a simpler use model than retraining the model periodically on an ongoing project. ID 16 explores the use case where SEQUENCER is trained with samples from the same projects that the buggy test cases come from. CodRep4 is added to the training set data and then 4,711 random samples are removed for testing (these samples may be from CodRep or Bugs2Fix project files). When the training data includes bugs from the same projects as the test data, we see a 12% improvement in the best model. This use model is viable, but it does require more complete integration of SEQUENCER into a project regression system.

6 RELATED WORK

The work presented here is on built on top of two big and active research fields: program repair and machine learning on code. We refer to recent surveys for getting a good overview on them: [1] for program repair and Allamanis *et al.*'s [8] for the latter. In the following, we focus on those works that are about learning and automatic repair.

sk_p is a program repair technique for syntactic and semantic errors in student programs submitted to MOOCs [43]. First, it uses the previous and next statement to predict the statement in the middle, *i.e.,* to replace the current statement. The probability of a patch is the product of the probabilities for all chosen statements. As we do, sk_p uses beam search to produce the top n predictions.

Another paper on MOOCs [44] repairs student submissions in Python by combining learning and sketch-based synthesis. The approach by Wang *et al.* [45] considers MOOC but the technique itself is completely different: [45] does deep learning on program traces in order to predict the kind

ID	Model description	mean	SD	max	chng
0	Golden Model	859	61	950	—
1	more training iterations (20K vs 10K)	832	78	901	-5%
2	smaller token vocabulary (700 vs 1000)	824	70	886	-7%
3	larger token vocabulary (1400 vs 1000)	868	32	907	-5%
4	with unsupervised pretraining	821	65	922	-3%
5a	less training data (CR vs CR+Bugs2Fix)	742	47	810	-15%
5b	less training data (Bugs2Fix vs CR+Bugs2Fix)	748	24	785	-17%
6	no bridge layer from encoder to decoder	887	34	942	-1%
7	fewer LSTM layers on enc/dec (1 vs 2)	281	203	513	-46%
8	more LSTM layers on enc/dec (3 vs 2)	833	49	914	-4%
9	fewer LSTMs per layer (128 vs 256)	848	40	888	-11%
10	more LSTMs per layer (512 vs 256)	812	89	907	-5%
11	without context (input only buggy line)	738	63	826	-13%
12	no truncation of *abstract buggy context*	crash			
13	truncate to larger context (4K vs 1K)	848	79	950	-0%
14	truncate to smaller context (500 vs 1K)	826	54	890	-6%
15	remove START_BUG & END_BUG	331	33	412	-57%
16	Intraproject training (4,711 test-cases from CR+Bugs2Fix)	984	47	1068	+12%

TABLE 3: Results with selected configurations in the parameter neighborhood of the golden model. For ID 0 through 15, results are total exact matches when model is tested on 4,711 testcases from CR4Full. ID16 results selected 4,711 testcases after merging CR1,2,3,4, and 5 with Bugs2Fix

.

of bug affecting a student submission. The main differences between those works and ours are that 1) we consider a larger context (the buggy class) and 2) we consider real programs for training and testing that are bigger and more complex than student's submissions. Shin *et al.* [46] consider simple programs in the educational programming language Karel. As SEQUENCER, their system predicts to delete, insert or replace tokens. Henkel *et al.* [47] compute an embedding for symbolic traces and perform a pilot experiment for fixing error-handling code, which is very different from concrete bug fixing as we do here.

DeepFix is a program repair tool for fixing compiler errors in introductory programming courses [27]. The input is the whole program, (100 to 400 tokens long for their data), and the output is a single line fix. The vocabulary size is set to 129, which was enough to map every distinct token type to a unique word in the vocabulary. TRACER is another program repair tool for fixing compiler errors which outperforms DeepFix in terms of success rate [24]. Santos *et al.*'s [48] further refines the idea and evaluates it with an even larger dataset. The focus of those three works and ours is very different, they focus on compiler errors, we focus on logical bugs. For compiler errors, one does not need to consider the whole vocabulary, but only token types. On the contrary, we have to address this problem and we do so by using the copy mechanism.

DeepRepair [49] is an early attempt to integrate machine learning in a program repair loop. DeepRepair leverages learned code similarities, captured with recursive autoencoders [50], to select repair ingredients from code fragments that are similar to the buggy code. Our usage of learning is different, DeepRepair uses machine learning to select interesting code, SEQUENCER uses machine learning to generate the actual patch.

Tufano *et al.* investigated the feasibility of using neural machine translation for learning bug-fixing patches via NMT [10]. The authors first perform a source code abstraction process that relies on a combination of Lexer+Parser which replaces identifiers and literals in the code. The goal of this abstraction is it reduce the vocabulary while keeping the most frequent identifiers/literals. In their work the authors analyzed small methods (no longer than 50 tokens) and medium methods (no longer than 100 tokens) and observed a drop in performance for longer methods. Since their approach takes a buggy method as input and generates the entire fixed method as output, the maximum method length Tufano *et al.* considered is only 100 tokens. Their work addressed the vocabulary problem by renaming rare identifiers through a custom abstraction process. SEQUENCER is different in the following ways. First, we consider the entire context of the buggy class, rather than only the buggy method, in order for the model to access more tokens when predicting the fix. Second, our abstraction process uniquely utilizes the copy mechanism (which they do not), which allows SEQUENCER to utilize a larger set of tokens when generating the fix and to include information about the context within the *abstract buggy context* in which a token is used. Beyond those two major qualitative differences, a quantitative one is that they only consider small methods, no longer than 100 tokens, while we have no such restriction; SEQUENCER can potentially generate a one-line patch within a method of any size.

Parallel work by Hata *et al.* [34] discusses a similar network architecture, also applied to one-line diffs. The major differences between [34] and our work are the following: First, they do project-specific training, which means that their approach is only evaluated on testing data coming from the same project. On the contrary, we do global training and we show that SEQUENCER captures repair operators applicable to any project. Our qualitative case studies are unique with that respect. Second, they only look at wellformedness of the output, while we also compile and execute the predicted patch. Our work is an end-to-end test-suite based repair approach. Third, their input is limited to the precise buggy code to replace, while SEQUENCER uses *abstract buggy context*, which allows for a broader set of tokens for the copy mechanism to select from.

7 CONCLUSION

In this paper, we have presented a novel approach to program repair, called SEQUENCER, based on sequence-to-sequence learning. Our approach uniquely combines an encoder/decoder architecture with the copy mechanism to overcome the problem of large vocabulary in source code. On a testing dataset of 4,711 tasks taken from projects which were not in the training set, SEQUENCER is able to successfully predict 950 changes. On Defects4J one-line bugs, SEQUENCER produces 61 plausible, test-suite adequate patches. To our knowledge, our paper is the first ever to show the effectiveness of the copy mechanism for program repair, which provides a mechanism to alleviate the unlimited vocabulary problem.

This work opens promising research directions. First, we aim to improve and adapt SEQUENCER with the goal of addressing multi-line patches. We believe there are different ways we can tackle this: (i) for fixes modifying contiguous lines of code (*i.e.,* hunk) we can extend SEQUENCER to learn to generate multiple lines of code as output, with the special tokens (*i.e.,* `<START_BUG>` and `<END_BUG>`) surrounding the entire hunk; (ii) for fixes modifying multiple lines in different locations, we could envision SEQUENCER generating a finite set of combinations of the program containing a predicted fixed line for each of the suspicious locations. Second, there is some preliminary work on tree-to-tree transformation learning [51], which conceptually is very appropriate for code viewed as parse trees. Such techniques may augment or supersede sequence-to-sequence approaches. Finally, the originality of our context abstraction is to capture class-level, long range dependencies: we will study whether such a network architecture is able to capture dependencies beyond that, at the package or application level.

REFERENCES

[1] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, pp. 1–24, 2017. DOI: 10.1145/3105906. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf.

[2] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, ACM, 2016, pp. 691–701.

[3] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," ICSE, 2018.

[4] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, IEEE, 2017, pp. 648–659.

[5] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[6] R. Solé and D. Valbelle, *The Rosetta Stone: the story of the decoding of hieroglyphics*. Profile, 2001.

[7] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 763–773.

[8] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[9] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *CoRR*, vol. abs/1704.04368, 2017. arXiv: 1704.04368. [Online]. Available: http://arxiv.org/abs/1704.04368.

[10] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology*, 2018.

[11] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: ACM, 2015, pp. 24–36, ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771791. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771791.

[12] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[13] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*, 2019.

[14] R. Karampatsis and C. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. arXiv: 1903.05734. [Online]. Available: http://arxiv.org/abs/1903.05734.

[15] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014, pp. 437–440.

[17] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, no. 12, pp. 971–987, 2006.

[18] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *CoRR*, vol. abs/1409.1259, 2014. arXiv: 1409.1259. [Online]. Available: http://arxiv.org/abs/1409.1259.

[19] J. Kiefer, J. Wolfowitz, *et al.*, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[22] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

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

[24] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: For the student programs, from the student programs," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ACM, 2018, pp. 78–87.

[25] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proc. ACL*, 2017. DOI: 10.18653/v1/P17-4012. [Online]. Available: https://doi.org/10.18653/v1/P17-4012.

[26] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[27] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning.," in *AAAI*, 2017, pp. 1345–1351.

[28] M.-T. Luong, I. Sutskever, Q. V. Le, O. Vinyals, and W. Zaremba, "Addressing the rare word problem in neural machine translation," *arXiv preprint arXiv:1410.8206*, 2014.

[29] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017, pp. 416–426.

[30] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, 2017, pp. 660–670.

[31] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," 2018.

[32] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015, pp. 532–543.

[33] Z. Chen and M. Monperrus, "The CodRep Machine Learning on Source Code Competition," *ArXiv e-prints*, Jul. 2018. arXiv: 1807.03200 [cs.SE].

[34] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *arXiv preprint 1812.07170*, 2018.

[35] P. Koehn and R. Knowles, "Six challenges for neural machine translation," *CoRR*, vol. abs/1706.03872, 2017. arXiv: 1706.03872. [Online]. Available: http://arxiv.org/abs/1706.03872.

[36] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 3–13, ISBN: 978-1-4673-1067-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337225.

[37] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 254–265, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568254. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254.

[38] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13, Silicon Valley, CA, USA: IEEE Press, 2013, pp. 356–366, ISBN: 978-1-4799-0215-6. DOI: 10.1109/ASE.2013.6693094. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693094.

[39] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: An eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, Essen, Germany: ACM, 2012, pp. 378–381, ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351752. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351752.

[40] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*, Springer, 2012, pp. 437–478.

[41] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," *arXiv preprint arXiv:1812.07283*, 2018.

[42] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 1–10, ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985795. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985795.

[43] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "Sk_p: A neural program corrector for moocs," *CoRR*, vol. abs/1607.02902, 2016. arXiv: 1607.02902. [Online]. Available: http://arxiv.org/abs/1607.02902.

[44] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *2018 IEEE/ACM 40th International Conference on Software Engineering*, 2018, pp. 60–70.

[45] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *arXiv preprint arXiv:1711.07163*, 2017.

[46] R. Shin, I. Polosukhin, and D. Song, "Towards specification-directed program repair," in *ICLR Workshop*, 2018.

[47] J. Henkel, S. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proceedings of ESEC/FSE*, 2018.

[48] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 311–322.

[49] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proceedings of SANER*, 2019.

[50] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, Singapore, Singapore: ACM, 2016, pp. 87–98, ISBN: 978-1-4503-3845-5. DOI: 10 . 1145 / 2970276.2970326. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326.

[51] S. Chakraborty, M. Allamanis, and B. Ray, "Tree2tree neural translation model for learning source code changes," *arXiv*, vol. abs/1810.00314, 2018.

PLACE PHOTO HERE

Zimin Chen Biography text and photo will be provided.

PLACE PHOTO HERE

Steve Kommrusch Biography text and photo will be provided.

PLACE PHOTO HERE

Michele Tufano Biography text and photo will be provided.

PLACE PHOTO HERE

Louis-Noël Pouchet Biography text and photo will be provided.

PLACE PHOTO HERE

Denys Poshyvanyk Biography text and photo will be provided.

PLACE PHOTO HERE

Martin Monperrus Biography text and photo will be provided.