ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair

Thibaud Lutellier
University of Waterloo
Canada
tlutelli@uwaterloo.ca

Lawrence Pang
University of Waterloo
Canada
lypang@edu.uwaterloo.ca

Viet Hung Pham
University of Waterloo
Canada
hvpham@uwaterloo.ca

Moshi Wei
University of Waterloo
Canada
m44wei@uwaterloo.ca

Lin Tan
Purdue University
United States
lintan@purdue.edu

June 21, 2019

Abstract

Automated generate-and-validate (G&V) program repair techniques typically rely on hard-coded rules, only fix bugs following specific patterns, and are hard to adapt to different programming languages.

We propose ENCORE, a new G&V technique, which uses ensemble learning on convolutional neural machine translation (NMT) models to automatically fix bugs in multiple programming languages. We take advantage of the randomness in hyper-parameter tuning to build multiple models that fix different bugs and combine them using ensemble learning. This new convolutional NMT approach outperforms the standard long short-term memory (LSTM) approach used in previous work, as it better captures both local and long-distance connections between tokens.

Our evaluation on two popular benchmarks, Defects4J and QuixBugs, shows that ENCORE fixed 42 bugs, including 16 that have not been fixed by existing techniques. In addition, ENCORE is the first G&V repair technique to be applied to four popular programming languages (Java, C++, Python, and JavaScript), fixing a total of 67 bugs across five benchmarks.

1 Introduction

To improve software reliability and increase engineering productivity, researchers have developed many approaches to fix software bugs automatically. One of the main approaches for automatic program repair is the G&V method [1, 2, 3, 4, 5, 6]. First, candidate patches are generated using a set of transformations or mutations (e.g., deleting a line and adding a clause). Second, these candidates are ranked and validated by compiling and running a given test suite. The G&V tool returns the highest ranked fix that compiles and passes fault-revealing test cases in the test suite.

While G&V techniques successfully fixed bugs in different datasets, a recent study [7] showed that very few correct patches are in the search spaces of state-of-the-art techniques, which puts an upper limit on the number of correct patches that a G&V technique can generate. It is possible to extend the search space of G&V techniques. However, a previous study [7] showed that this solution failed to help existing approaches find more correct patches. It can even reduce the number of correct fixes that these techniques can produce. Therefore, the G&V field is in need of a novel approach that can generate and search a large search space in a more intelligent and scalable manner.

```
1 import java.util.*;
2 public static int max_sublist_sum(int[] arr) {
3   int max_ending_here, max_so_far = 0;
4   for (int x : arr) {
5 -   max_ending_here=max_ending_here+x;
5 +   max_ending_here=Math.max(0,max_ending_here+x);
6     max_so_far = Math.max(max_so_far,max_ending_here);
7   }
7   return max_so_far;}}
```

Figure 1: A QuixBugs program fixed by ENCORE

Neural machine translation is a popular deep-learning (DL) approach that generates likely sequences of tokens given an input sequence. NMT has mainly been applied to natural language translation tasks (e.g., translating French to English). In this case, the model captures the semantic meaning of a French sentence and produces a semantically equivalent sentence in English.

Recently, NMT models have been used for program synthesis [8], where a model is trained to "translate" specifications in English to source code. Other studies have been conducted to detect and repair small syntax [9] and compilation [10] issues (e.g., missing parenthesis). While these models show promising results for fixing compilation issues, they only learn the syntax of the programming language (i.e., although the fixed program compiles and are syntactically correct, they often still contain runtime bugs). Additionally, DeepFix [10] showed the limitations of such approaches when repairing statements containing more than 15 characters.

Existing work applying NMT to repair bugs generally uses the best model with the "best parameters" based on a validation set. Due to the diversity of bugs and fixes, such approaches struggle to cope with the complexity of program repair. The current state-of-the-art NMT approach, SequenceR [11], performs significantly worse on the Defects4J benchmark than most G&V techniques that use handcrafted rules to generate repairs. In addition, NMT-based program repair approaches struggle with representing the context of a bug. Although context is important for fixing bugs, there is not yet an effective approach for incorporating context to the DL models. Naively adding the bug context to the input of the models makes the input sequences very long. As a result, the models struggle to correctly capture meaningful information and such approaches can only fix short methods. For example, Tufano et al. [11] focus on methods that contain fewer than 50 tokens. Further more, it is challenging to obtain meaningful context for millions of training instances since such training instances are partial code snippets that are not always compilable. As a result, previous work used a reduced training set of 35,578 instances [11] while most NMT techniques in other domains need millions of training instances to perform well. For example, training sets for translation tasks contain between 2.8M and 35.5M pairs of sentences depending on the language [12].

In this paper, we propose a new G&V technique called ENCORE that leverages an *ensemble* NMT architecture to generate patches. This new architecture consists of an ensemble of convolutional NMT models that have different levels of complexity and capture different information about the repair operations. Combining them allows ENCORE to learn different repair strategies that are used to fix different types of bugs. By ignoring the bug context while relying on a large training set, we significantly reduce the size of the input sequences, allowing us to fix bugs independently from the size of their context.

G&V methods are often based on complex hard-coded rules that require advanced domain knowledge and are programming language dependent. On the other hand, an NMT-based approach learns the rules directly from the data and can be trained to fix bugs in different programming languages without a major redesign of the technique. Thus, ENCORE can fix bugs that are not covered by such rules and is easily applicable to other programming languages.

ENCORE is trained on up to millions of pairs of buggy and fixed lines and produces patches that are validated against the program's test suite. ENCORE correctly fixes the bug in Figure 1. Using the buggy line as input (line 5, in red), ENCORE generates a correct patch (line 5, in green) that is identical to the developer's fix.

This paper makes the following contributions:

- A new NMT model that uses an ensemble approach to capture the diversity of bug fixes. We show that this model performs better than the standard NMT models for automatic program repair (APR).

- A new end-to-end automatic program repair technique, ENCORE, that leverages NMT to generate bug fixes automatically and validate generated patches against the test suite. When evaluated on two popular Java benchmarks
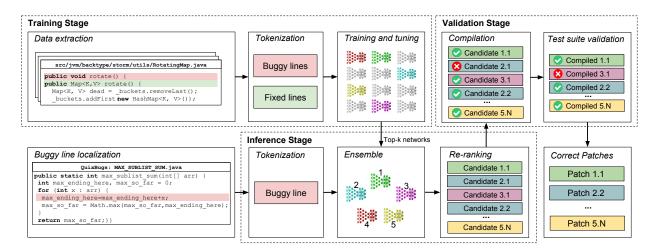
Figure 2: ENCORE overview

(Defects4J and QuixBugs), our technique fixes 42 bugs, including 16 that have not been fixed by any existing APR tools.

- An empirical study of the type of bugs that are successfully fixed by ENCORE. We show that ENCORE can fix very diverse bugs including condition modification, variable replacement or return statement modification.

- A study showing the portability of ENCORE to different programming languages. Our approach has been extended to three additional programming languages with little manual effort and respectively fixed 17, 5, and 3 bugs in the Python, C++, and JavaScript benchmarks.

- A use of attention maps to explain why certain fixes are generated or not by ENCORE.

2 Background and terminology

We present DL background necessary to understand ENCORE.

Terminology: A DL *network* is a structure (i.e., a graph) that contains nodes or *layers* that are stacked to perform a specific task. Each type of layer represents a specific low-level transformation (e.g., convolution, pooling) of the input data with specific parameters (i.e., *weights*). We call DL *architecture* an abstraction of a set of DL networks that have the same types and order of layers but do not specify the number and dimension of layers. A set of *hyper-parameters* specifies how one consolidates an architecture to a network (e.g., it defines the convolution layer dimensions or the number of convolutions in the layer group). The hyper-parameters also determine which optimizer is used in training along with the optimization parameters, such as the learning rate and momentum. We call a *model* (or trained model), a network that has been trained, which has fixed weights.

Attention: The attention mechanism [13] is a recent DL improvement. It helps a neural network to focus on the most important features. Traditionally, only the latest hidden state of the encoder are fed to the decoder. If the input sequence is too long, some information regarding the early tokens are lost, even when using LSTM nodes [13]. The attention mechanism overcomes this issue by storing these long-distance dependencies in a separate attention map and feeding them to the decoder at each time step.

Neural Machine Translation (NMT): NMT models typically leverage the Recurrent Neural Network (RNN) architecture to generate the most likely sequence of tokens given an input sequence. Long short-term memory (LSTM) [14] and Gated Recurrent Unit [15] are the two most popular RNN layers for NMT. A popular NMT approach is the encoder-decoder method [15], also called sequence-to-sequence (seq2seq). The encoder, consisting of a stack of layers, processes a sequence of tokens of variable length (in our case, a buggy code snippet) and represents it as a fixed length encoding. The decoder translates this representation to the target sequence (in our case, a fixed code snippet).

3 Approach

Our technique contains three stages: training, inference, and validation. Figure 2 shows an overview of ENCORE. In the training phase, we extract pairs of buggy and fixed lines from open-source projects. Then, we preprocess these

lines to obtain sequences of tokens and feed the sequences to an NMT network that is tuned with different sets of hyper-parameters for one epoch (i.e., a single pass on the training instances). We further train the top-k models until convergence to obtain an ensemble of k models. Since each of the k models has different hyper-parameters, each model learns different information that helps fix different bugs.

In the inference phase, a user inputs a buggy line into ENCORE. It then tokenizes the input and feeds it to the top-k best models, which each outputs a list of patches. These patches are then ranked and validated by compiling the patched project. If a test suite is available, ENCORE runs it on the compilable fixes to further filter incorrect patches. The final output is a list of *candidate patches* that pass the validation stage.

Section 3.1 presents the challenges of using NMT to automatically fix bugs, while the rest of Section 3 describe the different components of ENCORE.

3.1 Challenges

APR presents several challenges, in particular when using NMT:

(1) Choice of NMT architecture: Due to the complexity of the task and the diversity of transformations used to fix bugs, traditional NMT architectures such as LSTM [16], do not perform well for APR. Previous work addressed this issue by limiting the scope of the repair to simple compilation errors [10] or by focusing on a few specific transformations [17]. However, these solutions are not ideal since it restricts the model to fixing a limited range of bugs. We address this issue by using a new type of NMT architecture, called fconv [12], that relies on convolutional layers instead of RNN layers. This architecture performs better because the convolutional layers better capture immediate context information than RNN layers while the multi-step attention allows the architecture to keep track of long-term dependencies. This type of architecture has been shown to work well for grammatical error correction [18] and translation [12] but has not been applied to other domains.

(2) Diversity of Bug Repair: Bugs are very diverse, with different fixes needed to fix the same buggy line depending on the context. Since an NMT model has no access to the entire project (due to limitation of sequence size of existing NMT models), the model might not have enough information to generate a correct patch for its first try. In addition, a single model might overfit the training data and fail to capture the diverse fix patterns. To address this challenge, we propose an ensemble approach that combines models with different hyper-parameters, to increase the diversity of patches generated, combined with a validation stage that is used to discard incorrect patches. Our ensemble approach allows us to fix 47% more bugs than using a single model.

(3) Adaptability to different Programming Languages: Existing APR techniques rely on handcrafted fix patterns that require domain knowledge to create and are not easily transferable to different languages. Leveraging NMT allows ENCORE to learn how to fix bugs instead of relying on handcrafted patterns. Thus, ENCORE is generally applicable to other programming language, as we can obtain training data automatically. ENCORE is the first approach to successfully fix bugs in four popular programming languages.

(4) Large vocabulary size: Compared to traditional natural language processing (NLP) tasks such as translation, the vocabulary size of source code is larger and many tokens are infrequent or composed of multiple words. In addition, letter case indicates important meanings in source code (e.g., Zone is generally a class, `zone` a variable, and ZONE a constant), which increases further the vocabulary size. For example, previous work [11] had to handle a code vocabulary size larger than 560,000 tokens. Since such a high number of token is not scalable for NMT, practionners need to cut the vocabulary size significantly, which leaves a large number of infrequent tokens out of the vocabulary. We address this challenge by using a new tokenization approach that reduces the vocabulary size significantly without increasing the number of words out of the vocabulary. Overall, less than 2% of tokens in our test sets are out of the vocabulary (Section 3.3).

3.2 Data Extraction

We train ENCORE on pairs of buggy and fixed lines of code extracted from the commit history of 1,000 open-source projects. To remove commits that are not related to bug fixes, we apply a few filters. First, we only keep changes that have the words "fix," "bug," or "patch" in their associated commit message. This is a standard method that has been done in previous work [19]. However, by manually investigating a random sample of 100 pairs, we found that these patterns were not enough to filter unrelated commits. Indeed, only 62/100 pairs were related to real bug fixes. The main reason was that many developers use the keywords "fix" and "patch" in commits that are unrelated to bugs. To address this issue, we also exclude commits that contain the following six anti-patterns in their messages: "rename," "clean up," "refactor," "merge," "misspelling," and "compiler warning." Using these anti-patterns increases the number of correct pairs in our random sample to 93.

We further remove sequences which size is than 2 times the standard deviation, cosmetic changes (e.g., spacing and indentation). We also removes comments and sequences that contain non-utf-8 characters. We focus on single-statement changes as smaller sequences are easier to learn for NMT models. Finally, to propose a fair comparison with other APR techniques, we remove from our training data all changes that are identical to bug fixes in the Defects4J benchmark. After filtering, we obtain 1,159,502 pairs of buggy and fixed lines.

3.3 Input Representation

Input Sequences: The input of ENCORE is a tokenized buggy line. We chose this simpler representation over representations used in previous work (e.g. full function as input) for the following reasons:

(1) NMT approaches work well with small sequences, but struggle when an input sequence contains more than 30 tokens [20]. Adding context to the input significantly increases the size of the input sequences. For example, the median size of previous work [11] training instances is above 300 tokens. Only feeding the buggy line allows us to reduce the size of training instances to an average of 24.3 tokens per sequence, making the learning process easier.

(2) Context generally contains many tokens that are unnecessary for fixing the bug. Thus, increasing the size of the context also has the disadvantage of significantly increasing the vocabulary size. For example, while having only 35,578 training instances, previous work's vocabulary contains 567,304 tokens. Not using context gives us an unfiltered vocabulary of over 200,000 tokens for about 1 million instances. The tokenization process described below further reduces the vocabulary to 64,044 tokens while keeping the ratio of out-of-vocabulary tokens during inference below 2%.

(3) Although context is important for fixing bugs, there is not yet an effective approach for incorporating large context to NMT models. In practice, we found that straightforward approaches of adding context do not increase the number of bugs fixed.

Tokenization: Typical NMT approaches take a vector of tokens as input. Therefore, our first challenge is to choose a correct abstraction to transform source code into sequences of tokens.

The lowest level of tokenization we can use is character-level tokenization. This has the advantage of using a small set of different tokens. However, this level of tokenization makes the training of the model harder as it has to learn what a "word" is before learning correct statements. This tokenization method has been used successfully to fix compilation errors in DeepFix but showed limits for statements that contained more than 15 characters [10].

For most natural languages, using word-level tokenization provides better results than character-level tokenization [21] so we decided to use word-level tokenization. While source-code is analogous to natural language, word-level tokenization presents several challenges that are specific to programming languages.

First, the vocabulary size becomes extremely large and many words are infrequent or composed of multiple words without separation (e.g., getNumber and get_Number are two different words). To address this issue specific to source-code generation, we enhanced the word-level tokenization by also considering underscores, camel letter and numbers as separators. Because we need to correctly regenerate source code from the list of tokens generated by the NMT model, we also need to introduce a new token (`<CAMEL>`) :wqfor camel cases, in order to mark where the split occurs. In addition, we abstract all strings in one specific token "STRING" and all infrequent numbers (i.e., different from 1 and 0) to "NUMBER." The reason for the string abstraction is that tokens in strings represent a completely different language that might confuse the model.

Previous work [22] uses a complete abstraction of all tokens of the buggy line. We do not do it for several reasons: first, such abstraction loses the semantics contained in the variable name. We believe that such semantics provide useful information to the network. Second, the abstraction is implicitly included in the original code snippet. If the network needs it to repair a bug, the model will learn it from the source code. In fact, the provided abstraction might not be the best abstraction for the network.

3.4 NMT Architecture

Figure 3 shows an overview of ENCORE's neural machine translation architecture. For simplicity, Figure 3 only displays a network with one convolution layer. In practice, depending on the hyper-parameters, a complete network has 2 to 10 convolution layers for each encoder and decoder. Our architecture consists of three main components: an encoder, a decoder, and an attention module.

In training mode, the model has access to both the buggy and the fixed lines. The model is trained to generate the best representations of the transformation from buggy to fixed lines. In practice, this is conducted by finding the best
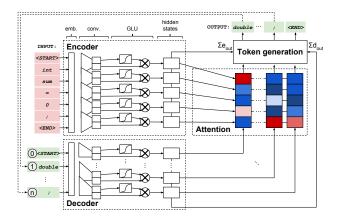
Figure 3: The NMT architecture used in ENCORE.

combination of weights that translates buggy lines in the training set to fixed lines. Multiple passes on the training data are necessary to obtain the best set of weights.

In inference mode, since the model does not have access to the fixed line, the decoder processes tokens one by one, starting with a generic ¡START¿ token. The output of the decoder and the encoder are then combined through the multi-step attention module. Finally, new tokens are generated based on the output of the attention, the encoder and the decoder. The generated token is then fed back to the decoder until the ¡END¿ token is generated.

Following the example input in Figure 3, a user inputs the buggy statement "int sum=0;" to ENCORE. After tokenization, this sequence is fed to the encoder which generates an encoded representation (e_{out}). Since ENCORE did not generate any token yet, the token generation starts by feeding the token ¡START¿ to the decoder (iteration 0) The output of the decoder (d_{out}) is then combined with the encoder output using a dot product to form the first column of the attention map. The colors of the attention map indicate how important each input token is for generating the output. For example, to generate the first token, the tokens int and 0 are the two most important input tokens as they appear in dark and light red respectively. The token generation combines the output of the attention, as well as the sum of the encoder and decoder outputs (Σe_{out} and Σd_{out}) to generate the token double. This new token is added to the list of generated tokens and the list is given back as a new input to the decoder (iteration 1). The decoder uses this new input to compute the next d_{out} that is used to build the second column of the attention map and generate the next token. The token generation continues until the token ¡END¿ is generated.

We describe below the different modules of the network.

Encoder and Decoder: The purpose of the encoder is to provide a fixed length vectorized representation of the input sequence while the decoder translates such representation to the target sequence (i.e., the patched line). Both modules have a similar structure that consists of three main blocks: an embedding layer, several convolutional layers, and a layer of gated linear units (GLU).

The embedding layers represent input and target tokens as vectors, with tokens occurring in similar contexts having a similar vector representation. In a sense, the embedding layers represent the model's knowledge of the programming language.

The output of the embedding layer is then fed to several convolutional layers. The size of the convolution kernel represents the number of surrounding tokens that are taken into consideration. Stacking such convolutional layers provides multiple levels of abstraction for our network to work with. The convolution layers also provide information regarding surrounding tokens. The number of tokens considered depends on the size of the convolution kernels. This layer is different in the encoder and the decoder. The encoder uses information from both the previous and the next tokens in the input sequence (since the full input sequence is known at all time), while the decoder only uses information about the previously generated tokens (since the next tokens have not been generated yet). This difference is represented in Figure 3 by full triangles in the encoder and half triangles in the decoder.

After the convolutional layers, a layer of GLU (represented by the sigmoid and multiplication boxes in Figure 3). is used to decide which information should be kept by the network. For more details about the encoder and decoder, refers to previous work [12].

Multi-Step Attention: As mentioned in Section 2, the attention mechanism helps to keep track of early information by connecting all the hidden states of the encoder to the decoder instead of only the last hidden state. In practice, this is done with a simple dot product between the output of the encoder and decoder convolution layers.

Compared to traditional attention, multi-step attention uses an attention mechanism to connect the output of each convolutional layers in the encoders and decoder. When multiple convolutional layers are used, it results in multiple attention maps. Multi-step attention is useful because it connects each level of abstraction (i.e., convolutional layers) to the outputs. This increases the amount of information from the decoder that is shared with the decoder when generating the target tokens.

The attention map represent the impact of input tokens for generating a specific token and can help understand why a specific output is generated. As shown in Figure 3, the rows of the map represent the input while the columns represent the generated tokens. For each generated token, the attention map shows which input tokens are the most important. For example, in Figure 3, the most important tokens to generate `double` were the first one (`int`) and the second last one (`0`). We analyze attention maps to answer RQ5.

Token Generation: The token generation combines the output of the attention layers, encoder ($\Sigma\ e_{out}$) and decoder ($\Sigma\ d_{out}$) to generate the next token. Each token in the vocabulary is ranked by the token generation component based on their likelihood of being the next token in the output sequence. If we are only interested in the top-1 result generated by the model, the most likely token is selected and appended to the list of generated tokens. The list is then sent back as the new input of the decoder. The token generation stops when the ¡END¿ token is generated.

Beam Search: Since our model has no information about the context of the buggy line, it is unlikely for the first patch generated by the model to be correct. Therefore, we want ENCORE to generate multiple patches that will then be validated. Generating and ranking a large number of patches is challenging because since the patches are generated token by token, we cannot know the probability of the final sequence before generating all the tokens. and choosing the most likely token at each iteration might not lead to the most likely sequence. To address this issue, we use a search strategy called beam search that is commonly used for NMT. The goal of beam search is to find the most likely sequence instead of the most likely token at each step. For each iteration, the beam search algorithm checks the t most likely tokens (t corresponds to the beam width) and ranks them by the total likelihood score of the next s prediction steps (s correspond to the search depth). In the end, the beam search algorithm outputs the top t most likely sequences ordered based on the likelihood of each sequence.

3.5 Ensemble Learning:

Fixing bugs is a complex task because there are very diverse bugs with very different fixing patterns that vary in term of complexity. Some fix patterns are very simple (e.g., to change the operator $<$ to $>$) while others require more complex modifications (e.g., adding a null checker or calling a different function). Training an effective generalized model to fix all types of bug is difficult. Instead, it is easier to overfit models to fix specific types of bugs. Then, we can combine these models into a general ensemble model that will fix more bugs than one single model.

Therefore, we propose an ensemble approach that combines models with different hyper-parameters (different networks) that performed the best on our validation set. The complexity of the model is represented by hyper-parameters such as the number and dimension of convolutional layers in the encoder and decoder.

As described in Section 2, hyper-parameters are parameters that consolidate an architecture to a network. Hyper-parameters include the number of layers, the dimensions of each layer and specific rates such as the learning rate, the dropout or the momentum. Different hyper-parameters have a large impact on the complexity of a network, the speed of the training process and the final performance of the trained model. For this tuning process, we chose to apply random search because previous work showed that it is an unexpensive method that performs better than other common hyper-parameter tuning strategies such as grid search and manual search [23]. For each hyper-parameter, we define a range from which we can pick a random value. Since training a model until convergence is very expensive, the tuning process is generally for only one epoch (i.e., one pass on the training data). We trained n different models with different random sets of parameters to obtain models with different behavior and kept the top k best model based on the performance of each model on a separate validation set.

One challenge of ensemble learning is to combine and rank the output of the different models. In our case, the commonly used majority voting would not work well since we specifically chose models that are likely to generate different fixes. Instead, we use the likelihood of each sequence (i.e., fixes) generated by each model. Since we use beam search to generate the top-k patches for each model, each patch has an associated negative log likelihood. We use this score to rank the output of all models. When two models produce the same patches with different scores, we only consider the highest score. The intuition for selecting the higher score instead of the average is that the models

Table 1: Training dataset information. # token PL indicates the average number of tokens per line of code.

Language	# projects	# bug fixing commits	# instances	src. Vocabulary size	trg. Vocabulary size	# token PL
Java	1,000	1,752,212	1,159,502	51,703	49,303	24.3
Python	1,000	1,717,249	711,091	78,999	61,687	21.2
C++	1,000	4,466,504	3,599,472	234,039	222,583	17.0
JavaScript	1,000	1,341,121	831,711	43,919	38,543	17.8

are designed to fix different types of bugs, so if one model is confident that one specific patch is correct, its confidence should not be impacted by other models.

3.6 Patch Validation

Statement Reconstruction: Our model outputs a list of tokens that form a fix for the input buggy line. The statement reconstruction module generates a complete patch from the list of tokens. This step is mostly an inversion of the tokenization step. However, there are two abstractions that cannot be reversed to source code directly: the <STRING> and <NUMBER> tokens. For these two tokens, we extract candidate numbers and strings from the original buggy line and attempt to replace the corresponding tokens with them. If there are no strings or numbers in the buggy line, the patch is discarded. Since there are not many different numbers or strings in one line of code, the impact of the total number of reconstructed patches is negligible. Once the fix is generated, it is inserted at the buggy location and we move to the validation step.

Compilation and Test Suite Validation: The neural network does not have access to the entire project; therefore it does not know whether the generated patches are compilable or pass the test suite. For this reason, we use a validation step to filter out patches that do not compile or do not pass the triggering test cases. This step is similar to the validation process done by traditional APR approaches with one difference. We do not require all the test cases to pass to consider a patch plausible. Indeed, there can be multiple bugs in one projects and several test cases might fail because of another bug [4]. Therefore, even the fixed version might still fail for some test cases. To alleviate this issue, we use the same two criteria as previous work [4]. First, the test cases that make the buggy version pass should still pass on the patched version. Second, test cases that failed on the version fixed by developers can still fail.

3.7 Generalization to Other Languages

Since ENCORE learns patterns automatically instead of relying on handcrafted patterns, it can be generalized to other programming languages with minimum effort. The main change required is to obtain new input data in the correct language. Fortunately this is easy to do since our training set is extracted from open-source GitHub projects. Once the data for the new programing language has been extracted, the top k models can be retrained without any re-implemetation. ENCORE will learn fix patterns automatically for the new programming language.

4 Experimental Setup

Dataset: To train our technique on different programming languages, we collect data from the top 1,000 GitHub projects (based on star rating) in 4 popular languages (i.e., Java, Python, C++, and JavaScript). Table 1 presents a summary of our dataset. We extract more training samples for Java and C++ because these two languages contain many extremely large projects such as Linux and Apache projects. To tune hyper-parameters, we pick a random sample of 2,000 instances as our validation dataset and use the rest for training.

We evaluate ENCORE on five benchmarks. For Java, we use Defects4J [24] and QuixBugs [25]. For Python, we use Python's version of QuixBugs. For C++, we used the 69 real-world defects from prior work [26, 2]. Since there is no automatic repair benchmark in JavaScript, we use the 12 examples associated with common bug patterns in JavaScript described in prior work [27].

To ensure fairness, if a bug appears in both our test and training sets (or both our test and validation sets), we remove the bug from our training set, which is rare anyways.

Training and Tuning: We use random search to tune hyper-parameters. We limit the search space to reasonable values: embedding size (50-500), convolution layer dimensions (128*(1-5), (1-10)), number of convolution layers (1-10), drop out rate (0-1), gradient clipping level (0-1), learning rate (0-1), and momentum (0-1). We first uniformly

pick a random set of hyper-parameters within the search space. Then, we train the model for one epoch using the training data. The trained model is evaluated using the validation set. We repeat the process for five days and rank the hyper-parameters sets based on their perplexity measures [28], which is a standard metrics in NLP that measure how well a model generates a token.

Infrastructure: We use the implementations of LSTM, Transformer, and FConv provided by fairseq-py [29] running on Pytorch [30]. Our models were trained and evaluated on an Intel Xeon E5-2695 and two Gold SKL 5120 machines and NVIDIA TITAN V and Xp GPUs.

Performance: The median time to train our NMT model for 1 epoch during tuning is 38 minutes. Training our top 5 models sequentially until convergence took 93 hours (23 hours if training concurrently on the two Xeon servers). In inference, generating 1000 patches for a bug takes on average 8 seconds.

5 Evaluation and Results

ENCORE generates a list of candidate patches that successfully pass all the bug triggering test cases. For evaluation purpose only, we manually compare the candidate patches to the developer patch and consider a patch as a correct fix if it is identical or semantically equivalent to the developer patch. Upon acceptance, we will make available the hyper-parameter values of our ensemble models, the list of bugs correctly fixed by ENCORE, the top 5 trained models, as well as the patches generated by ENCORE.

5.1 RQ1: How does ENCORE perform against state-of-the-art NMT models?

Approach: We compare ENCORE with SequenceR [11], the state-of-the-art program NMT-based APR and two other state-of-the-art NMT architectures (i.e., LSTM [16] and Transformer [31]). These two models have not been used for program repair so we implemented in the same framework as our work (i.e., using Pytorch [30] and the fairseq-py [29] library). We then trained and tuned them similarly to ENCORE. SequenceR [11] is an approach on arXiv concurrent to ENCORE that uses NMT to automatically repair bugs. We use the numbers reported by SequenceR's authors on the Defects4J dataset for comparison. We cannot compare with SequenceR on the QuixBugs dataset because they did not run their tool on the QuixBugs dataset and the tool is unavailable.

Following SequenceR [11], we assume perfect localization to ensure a fair comparison. As mentioned in previous work [32], the effect of fault localization on automatic repair techniques should be analyzed separately, since different localization techniques provide significantly different results [32].

Comparison with State-of-the-art NMT: ENCORE with k=5 fixes 28 and 14 bugs in the Defects4J and QuixBugs benchmarks respectively, including 20 and 13 that are ranked first. Specifically, Table 2 displays the number of bugs fixed by the different approaches on our two benchmarks. The Fconv column shows the results for our approach without ensemble. In the top-1 column, we report the number of bugs that were fixed correctly with the first candidate patch generated by the tool. The column "All" display the total number of bugs each technique can fix, regardless of the ranking. Our technique fixes the most number of bugs considering the top-1 candidate, with 20 bugs fixed in the Defects4J benchmark and 13 in the QuixBugs benchmark. Considering all correct patches, ENCORE also fixes the most bugs with 28 correct fixes in the Defects4J benchmark and 14 in the QuixBugs benchmark.

ENCORE fixes all bugs fixed by the LSTM and Transformer approaches and most of the bugs fixed by SequenceR.

Advantage of Ensemble Learning: To demonstrate the advantage of our ensemble learning approach, Figure 4 shows the total number of bugs fixed using the top-k models, with k varying from 1 to 10. With k=1 (i.e., without using ensemble), our model only fixes 18 bugs in the Defects4J dataset. As k increases, the number of bugs fixed increases, until reaching a plateau of 29 bugs for k=7.

Since increasing k, increases the number of models considered, the number of generated patches increases too. Thus, it might have a negative impact on the ranking of correct fixes. The light blue dotted line in Figure 4 displays the evolution of the number of patches that are ranked first when the number of models increases. Surprisingly, this evolution is very similar to the evolution of the total number of bugs fixed and adding more models does not significantly reduced the number of correct patches ranked first.

Ensemble learning provides a significant improvement compared to using one single model. With k=5, the ensemble model fixes 47% more bugs than with a single model, with 66% more correct patches ranked first.

Impact of Hyper-parameter tuning: To ensure our results are the consequence of using different hyper-parameters and not because of the randomness in training, we train our first model 8 times with identical hyper-parameters. The
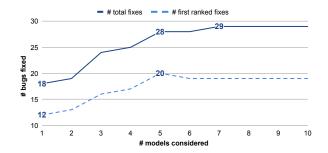
9

Figure 4: Number of bugs fixed as the number of models considered in ensemble learning increases.

Table 2: Comparison with other deep learning approaches.

		Baselines		SeqR [11]	Fconv	ENCORE (k=5)
		LSTM	**Trans.**			
Defects4J	Top-1	6	6	12	12	**20**
	All	10	9	14	18	**28**
QuixBugs	Top-1	8	9	NA	6	**13**
	All	9	9	NA	8	**14**

average number of bugs fixed in the Defects4J benchmark by this first model is 18.9, with a variance of 0.125. This shows that using an ensemble approach of the exact same model would not work well since the randomness in training does not have too much impact on the evaluation results.

Summary: Compared to state-of-the-art NMT-based approaches, ENCORE fixes the most number of bugs, **42 bugs**, across two popular Java benchmarks, including **33** that are ranked first.

5.2 RQ2: How does ENCORE perform against state-of-the-art APR techniques?

Approach: We compare ENCORE with state-of-the-art G&V approaches on all six projects in the Defects4J benchmark and the QuixBugs benchmark. We extracted the results for each technique from previous work [32] and cross-checked against the original paper of each technique. As we did for RQ1, we only consider bug fixes that have been manually verified as correct and are ranked first by the APR technique. For ENCORE we also show in parentheses the number of bugs fixed that have not been fixed by other techniques.

Results on the Defects4J benchmark: Table 3 shows how ENCORE performs against state-of-the-art G&V approaches. ENCORE generated 20 fixes that are ranked first, making it close to the best six G&V APR technique. and outperforming 11 techniques. In addition, three of the patches ranked first have not been fixed before, indicating that ENCORE can potentially fix different bugs than previous work. If we consider correct fixes generated by ENCORE independently from ranking, it is the second best G&V approach, fixing 28 bugs, fixing 6 bugs no other techniques fixed.

Results on the QuixBugs benchmark: Most G&V techniques were not evaluated on the QuixBugs benchmark, however; the ASTOR framework [33] (at the time, a combination of jKali, jGenProg, and jMutR), as well as Nopol [5], have been evaluated on this benchmark, fixing respectively 6 and 1 bugs. Compared to these two approaches, ENCORE fixes 14 bugs (13 of them being ranked first and 10 of them being unique to ENCORE) significantly outperforming both techniques.

Impact of Fault Localization: It is difficult to conduct a fair comparison of all APR techniques since the localization results used are different. For example, HDRepair uses Ochiai [45] but assumes that the correct file and method of the bug is known. SimFix uses GZoltar 1.6.0 combined with test case purification, while approaches such as jKali and Nopol used a now outdated version of GZoltar that is less accurate. SketchFix and ELIXIR did not report the localization framework they used. Such differences among the bug localization component make it difficult to do a fair comparison [32]. Following SequenceR, we assume perfect localization. However, looking at the output of GZoltar 1.6.0 implementation of Ochiai, only two of the bugs ENCORE fixes have a missing line-level localization,

Table 3: Comparison with state-of-the-art G&V approaches. The number of bugs only ENCORE can fix are in parentheses.

Projects	Astor [33]			HDR [34]	Nopol [5]	ACS [35]	Elixir [6]	JAID [36]	ssFix [37]	CapGen [38]	SketchFix [39]	FixMiner [40]	LSRepair [41]	SimFix [42]	SOFix [43]	SeqR [11]	ENCORE	
	jGP	jKali	jMutR														Top 1	All
Chart	0	0	1	0	1	2	4	2	3	4	6	5	3	4	5	1	2 (0)	3 (0)
Closure	0	0	0	0	0	0	0	5	2	0	3	5	0	6	0	3	8 (2)	11 (5)
Lang	0	0	0	2	3	3	8	1	5	5	3	2	8	9	4	2	4 (1)	4 (1)
Math	5	1	2	4	1	12	12	1	10	12	7	12	7	14	13	5	4 (0)	9 (0)
Mockito	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0 (0)	0 (0)
Time	0	0	0	0	0	1	2	0	0	0	0	1	0	1	1	0	1 (0)	1 (0)
Defects4J	5	1	3	6	5	18	26	9	20	21	19	25	19	34	23	12	20 (3)	28 (6)
QuixBugs		6 [44]		NA	1	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	13 (10)	14 (10)

indicating that for the remaining 26 of the bugs fixed by ENCORE, their localization results could be found by the state-of-the-art bug localization technique, including 4 bugs only fixed by ENCORE.

Summary: Considering the top-1 patches, ENCORE is on par with the best G&V APR techniques, outperforming the rest. In addition, 16 of the bugs fixed by ENCORE have not been fixed before (6 in the Defects4J benchmark and 10 in the QuixBugs benchmark).

5.3 RQ3: What type of bugs can ENCORE fix?

Approach: To understand the type of bugs fixed by ENCORE, we consider the list of actions associated with each Defects4J fixes proposed by previous work [46]. These actions indicate which transformations are used to repair the bugs. We also consider which parts of the program ENCORE modifies to generate a correct patch.

Results: ENCORE performed 25 different types of action to fix bugs in the Defects4J dataset, indicating that our models learn to fix bugs in different manners. On average, ENCORE performs 1.8 actions per bug fix while SequenceR performs 1.2 actions per bug fix, indicating that ENCORE might be fixing more complex bugs than SequenceR. Only 4 patches are deletions, while 11 of the bugs fixed by ENCORE require 2 or more actions to generate the correct patches, indicating that it can generate complex patches. ENCORE fixes bugs by modifying many different parts of a program, including conditional expressions (16 bugs), method calls (9 bugs), return statements (7 bugs), assignment (5 bugs), object instantiation (3 bugs), and method definition (1 bug). We show below some of the bugs ENCORE successfully fixed.

Summary: ENCORE fixes bugs from the Defects4J dataset by performing 25 different repair actions, showing that it can fix different types of bugs. In addition, ENCORE fixes bugs that require more repair action than SequenceR, indicating that ENCORE fixes more complex bugs.

5.4 RQ4: Can ENCORE be applied to other programming languages?

Approach: To evaluate the generalizability of ENCORE to other programming languages, we retrained our models on historical data for 3 popular programming languages (i.e., Python, C++, and JavaScript). For Python, we used Python's version of the QuixBugs benchmark (40 bugs). For C++, we evaluate ENCORE on a popular benchmark used in previous work [26, 2]. Finally, since to the best of our knowledge, there is no benchmark for automatic repair in JavaScript, we use the 12 examples associated to common bug patterns in JavaScript described in the Appendix of previous work [27]. For this RQ, we focus on the output of our neural network, ignoring the validation stage. Since we ignore the validation stage, we only consider a fix as correct if it's identical to the developer's and consider all generated patches.

Results: ENCORE performs very well on the Python benchmark, fixing 17 bugs. Figure 5 shows a bug that is correctly fixed in the Python dataset. ENCORE fixes 5 bugs on the C++ benchmark,. We only identified 16 bugs in this benchmark that require modifying a single localization, therefore the majority of the bugs in the C++ benchmark are out of the scope of ENCORE. While ENCORE performs less well than SPR [2] and Prophet [26] (with 18 and 16 bugs fixed respectively), it performs better than Kali [47] and GenProg [1] (with 2 and 1 bugs fixed respectively). In addition, one of the bugs fixed by ENCORE, *php-309579-309580* , has not been fixed by previous work. SPR and Prophet performs well on the C++ benchmark but require handcrafted patterns and would need a complete redesign to adapt to another language. In the JavaScript benchmark, ENCORE fixes 3 bugs associated to the "protect with value check," "add null place holder," and "propagate to callback" repair patterns. Figure 6 shows a correct fix by ENCORE for the bug pattern "protect with value check."

```
- for x in arr:
+ for x in arr[k:]:
```

Figure 5: Correct patch for *KHEAPSORT* Python bug

```
- if (val <= 0) {
+ if (val <= 0 || val == null) {
```

Figure 6: Correct JavaScript patch for a jQuery bug

Summary: ENCORE is the first approach that has been successfully applied without major re-implementation to different languages, fixing a total of **67 bugs** in four popular programming languages.

5.5 RQ5: Can we explain why ENCORE can (or fail to) generate specific fixes?

Approach: In this RQ, we provide explanations on how ENCORE generates patches to fix some bugs. These explanations are mostly for researchers. We do not provide explanations for developers because, unlike classification or defect prediction approaches that generate a line number which is difficult to analyze, ENCORE outputs a complete patch that passes all fault-revealing cases, which the developer can directly analyze.

The Majority of the Fixes are not Clones: By learning from historical data, the depth of our neural network allows ENCORE to fix complex bugs, including the ones that require generating new variables. As discussed in Section 4, the evaluation is not valid if the same bug appears both in the test benchmark and the training or validation sets. While we removed such cases from our training and validation sets, the same patch may still be used to fix different bugs introduced at different times in different locations. Having such patch clones in both training and test sets is valid, as recurring fixes are common. It is reasonable to and existing static analysis approaches have been proposed to, learn from past fixes to generate fixes for recurring bugs [48]. To understand this effect, we investigate whether the fixes generated by ENCORE are identical to bug fixes in our training sets. The majority of the bugs fixed by ENCORE do not appear in the training sets: only two patches from the C++ benchmark and one from the JavaScript benchmark appear in the training or validation sets. This suggests that simple clone-based approaches would fail to generate the fixes, and our NMT-based ensemble learning is effective in learning and generating completely different fixes.

Closure 93 **from Defects4J:** *Closure 93* in the Defects4J benchmark is one of the bugs only ENCORE can fix and is displayed in Figure 7. The bug is fixed by replacing the method call `indexOf` with the method `lastIndexOf`. ENCORE is able to make this correct change because of a similar change in our training set (in the hibernate-orm project, also shown in Figure 7). However, the change is not a simple clone since all the variable names are completely different, which indicates that ENCORE can learn some abstraction of the variable names in the original statement.

Lang 26 **from Defects4J:** ENCORE can also fix bugs that require more complex changes. For example, the fix for *Lang 26* from the Defects4J dataset shown in Figure 8 requires injecting a new variable `mLocale`. This new variable only appears four times in our training set, and never in a similar context. ENCORE is still able to generate the correct fix because, thanks to our tokenization approach, `mLocale` is divided into the tokens `m` and `Locale`. The token `Locale` occurs in our training set in similar context to the tokens `Gregorian`, `Time`, and `Zone` which are all in the buggy statement.

The attention map in Figure 10 confirms that these three tokens are important for generating the `Locale` variable. Specifically, the tokenized input is shown on the y-axis while the tokenized generated output is displayed on the x-axis. The token `<CAMEL>` between `m` and `Locale` indicates that these two tokens form one unique variable. The attention

```
// Correct Closure 93 patch generated by ENCORE
- int indexOfDot=namespace.indexOf('.');
+ int indexOfDot=namespace.lastIndexOf('.');

// Similar change occuring in the training data
- int end = message.indexOf(templateEnd, start );
+ int end = message.lastIndexOf(templateEnd, start );
```

Figure 7: Closure 93 patch and similar training data change

```
// Correct Lang 26 patch generated by ENCORE
- Calendar c = new GregorianCalendar(mTimeZone);
+ Calendar c = new GregorianCalendar(mTimeZone, mLocale);

// Similar change occuring in the trainning data
- Calendar calendar = new GregorianCalendar();
+ Calendar calendar = new GregorianCalendar(
          TimeZone.getDefault(), Locale.getDefault());
```

Figure 8: Lang 26 patch and similar training data change

```
// Correct BUCKETSORT patch generated by ENCORE
- for ( Integer count : arr ) {
+ for ( Integer count : counts ) {

// Correct Lang 59 patch generated by ENCORE
- str.getChars(0, strLen, buffer, size);
+ str.getChars(0, width, buffer, size);
```

Figure 9: Patches that requires replacing variable

map shows the relationship between the input token (vertical axis) and the generated tokens (horizontal axis). The color in a cell represents the relationship of corresponding input and output tokens with red colors indicating that an input token is influential in generating the output token, while darker blue colors indicate that the input token has little to no influence on the output token. For example, to generate the token `Locale`, the most influential input token is `Gregorian` (in light red), followed by `Time` and `Zone` (in light blue). On the other hand, the `c` token (in dark blue) does not influence the generation of `Locale`.

This example shows that ENCORE can generate patches, even if the exact same pattern has not occurred in the training set. For *Lang 26*, ENCORE generates a correct variable name, that never appears in the same context during training.

***BUCKETSORT* from QuixBugs and *Lang 59* from Defects4J:** Figure 9 shows two examples (*BUCKETSORT* from QuixBugs and *Lang 59* from Defects4J) of correct patches generated by ENCORE that require replacing a variable name by another one. These two bugs can only be fixed by the fourth model of our ensemble learning approach, which indicates that this specific model might be more successful than the others in learning this specific pattern.

***SUBSEQUENCE* from Quixbug:** Figure 11 shows an overfitted patch—an incorrect patch that makes the tests pass— generated by ENCORE. Our overfitted patch returns `Arrays.asList`, which is an `Array.ArrayList` (which has fixed length according to the JavaDoc) while the correct patch returns a normal `ArrayList`. At first glance, the patch that returns a fixed length `Array.ArrayList` should not pass the test cases because the algorithm needs to add elements to this `Array.ArrayList` later. After further investigation, we found that when an `ArrayList` is used as
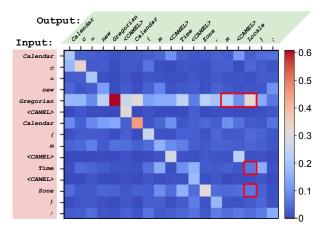

Figure 10: Attention map for the correct patch of *Lang 26* from the Defects4J benchmark generated by ENCORE

```
// Overfitted patch generated by ENCORE
- return new ArrayList();
+ return new ArrayList<>(Arrays.asList(new ArrayList()));

// Correct Patch
- return new ArrayList();
+ return new ArrayList<>(new ArrayList()));
```

Figure 11: Overfitted patch for SUBSEQUENCE

input to `asList`, the returned `Array.ArrayList` is backed by that `ArrayList` and does not have fixed length, which allows our patch to pass all the test cases. This overfitted patch highlights a potential bug in the `asList` method of the `java.util` library since the return value of `asList` does not behave as expected. This shows ENCORE also has the potential of generating mutants that can find bugs in software.

6   Limitations

There is randomness in the training process of deep learning model. We perform multiple runs and find that the randomness in training has little impact on the performances of the final trained models. The hyper-parameters tuning process also contains randomness. We tuned our model for five days, investigating almost 400 different sets of hyper-parameters using random search (Section 3.5). A main challenge of deep learning is to explain the output of a neural network. Fortunately, for developers, the repaired program that compiles and passes test cases should be self-explanatory. For users who build and improve ENCORE models, we leverage the recent multi-step attention mechanism [12] to explain why a fix was generated or not. For a complete end-to-end automatic program repair solution, the buggy line locatlization is needed. While we assume perfect localization, we also show that for all except two of the bugs that ENCORE fixes, their localized lines can be identified.

7   Related Work

Deep Learning for APR: Deep learning has been used to fix compilation errors [10]. More recently, NMT has been used for automatic program repair [22, 17, 11]. Devlin et al. work [17] used a neural network architecture to repair bugs in Python, focusing on four common transformations. They evaluated their model by injecting bugs generated from the same four transformations. While their model performed relatively well on injected bugs, it is limited as it can only fix bugs that follow the specific types of transformations and it is unclear whether it would work for fixing real-world bugs. A similar approach using an LSTM-based NMT was investigated by Tufano et al. [22]; however, this approach is limited to fixing bugs inside small methods and only generates a template of the fix, without generating the full correct statements. SequenceR[11] is a concurrent work that includes patch validation. Compared to SequenceR, ENCORE uses a different deep learning model combined with ensemble learning that fixes more bugs. In addition, ENCORE is generalizable to different programming languages while SequenceR focused on fixing Java bugs.

G&V Program Repair: Many APR techniques have been proposed [1, 2, 6, 5, 33, 49, 37, 35, 42, 39, 38, 43, 36, 34]. We use a completely different approach compared to these techniques, and as shown in Section 5, our approach fixes bugs that existing techniques have not fixed. In addition, these techniques require significant domain knowledge and manually crafted rules that are language dependent, while thanks to our ensemble NMT approach, ENCORE automatically learns such patterns and is generalizable to several programming languages with minimal effort.

Grammatical Error Correction: The counterpart of automatic program repair in NLP is grammatical error correction (GEC). Recently, work in the field of GEC mostly focuses on using machine translation in fixing grammatical errors [50, 51, 52, 53, 54, 18, 55, 56, 57, 58, 59]. Recent work [50] applied an attention-based convolutional encoder-decoder model to correct sentence-level grammatical errors. ENCORE is a new application of NMT models on source code and programming languages, which addresses unique challenges.

Deep learning in software engneering: The software engineering community had applied deep learning to performing various tasks such as defects prediction [19, 60, 61], source code representation [62, 63, 64, 65], source code summarization [66, 67] source code modeling [68, 69, 70, 71], code clone detection [72, 73], and program synthesis [74, 75, 76, 8]. Our work uses a new deep learning approach for automated program repair.

8 Conclusion

We propose ENCORE, a new end-to-end approach using NMT and ensemble learning to automatically repair bugs in multiple languages. We evaluate ENCORE on five benchmarks in four different programming languages and found that ENCORE can repair 67 bugs including 16 that have not been fixed before by existing techniques. In the future, we plan to improve our approach to work on multi-localization bugs and find an effective way to represent the context of a bug.

References

[1] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2012.

[2] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[3] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 85–91. ACM, 2016.

[4] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[5] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[6] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE Press, 2017.

[7] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 702–713. IEEE, 2016.

[8] Carol V Alexandru. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1068–1070. ACM, 2016.

[9] Eddie A Santos, Joshua C Campbell, Abram Hindle, and José Nelson Amaral. Finding and correcting syntax errors using recurrent neural networks. *PeerJ PrePrints*, 2017.

[10] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.

[11] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *arXiv preprint arXiv:1901.01808*, 2018.

[12] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

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

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

[16] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[17] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. *arXiv preprint arXiv:1710.11054*, 2017.

[18] Shamil Chollampatt and Hwee Tou Ng. A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction. 2018.

[19] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 297–308. IEEE, 2016.

[20] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[21] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8, 2012.

[22] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM*, pages 832–837, 2018.

[23] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[24] René Just, Darioush Jalali, and Michael 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*, pages 437–440. ACM, 2014.

[25] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.

[26] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *POPL'16*, page 298, 2016.

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

[28] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity – a measure of the difficulty of speech recognition tasks. *Journal of the Acoustical Society of America*, 62:S63, November 1977. Supplement 1.

[29] Fairseq-py. https://github.com/pytorch/fairseq, 2018.

[30] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[32] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. *ICST 2019*, 2019.

[33] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[34] Xuan-Bach D Le, David Lo, and Claire Le Goues. History driven automated program repair. 2016.

[35] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[36] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 637–647. IEEE, 2017.

[37] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.

[38] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. ICSE, 2018.

[39] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 888–891. ACM, 2018.

[40] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791*, 2018.

[41] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé François D Assise Bissyande. Lsrepair: Live search of fix ingredients for automated program repair. 2018.

[42] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. 2018.

[43] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129. IEEE, 2018.

[44] He Ye, Matias Martinez, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *CoRR*, abs/1805.03454, 2018.

[45] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[46] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *arXiv preprint arXiv:1801.06393*, 2018.

[47] Zichao Qi, Fan Long, Sara Achour, and Martin 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*, pages 24–36. ACM, 2015.

[48] Na Meng, Miryung Kim, and Kathryn S McKinley. Sydit: creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 440–443. ACM, 2011.

[49] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, volume 2017, 2017.

[50] Shamil Chollampatt and Hwee Tou Ng. A multilayer convolutional encoder-decoder neural network for grammatical error correction. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, February 2018.

[51] Zhuo Ran Liu and Yang Liu. Exploiting Unlabeled Data for Neural Grammatical Error Detection. *Journal of Computer Science and Technology*, 32(4):758–767, 2017.

[52] Allen Schmaltz, Yoon Kim, Alexander M. Rush, and Stuart M. Shieber. Adapting Sequence Models for Sentence Correction. 2017.

[53] Courtney Napoles and Chris Callison-Burch. Systematically Adapting Machine Translation for Grammatical Error Correction. *Proceedings of the 12th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 345–356, 2017.

[54] Keisuke Sakaguchi, Matt Post, and Benjamin Van Durme. Grammatical Error Correction with Neural Reinforcement Learning. 2017.

[55] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Shubha Guha, and Kenneth Heafield. Approaching Neural Grammatical Error Correction as a Low-Resource Machine Translation Task. (2016):595–606, 2018.

[56] Helen Yannakoudakis, Marek Rei, Øistein E Andersen, and Zheng Yuan. Neural Sequence-Labelling Models for Grammatical Error Correction. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2795–2806, 2017.

[57] Tao Ge, Furu Wei, and Ming Zhou. Reaching Human-level Performance in Automatic Grammatical Error Correction: An Empirical Study. (3):1–15, 2018.

[58] Tao Ge, Furu Wei, and Ming Zhou. Fluency Boost Learning and Inference for Neural Grammatical Error Correction. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 1:1–11, 2018.

[59] Masahiro Kaneko, Yuya Sakaizawa, and Mamoru Komachi. Grammatical Error Detection Using Error- and Grammaticality-Specific Word Embeddings. *Proceedings ofthe The 8th International Joint Conference on Natural Language Processing*, (2016):40–48, 2017.

[60] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 318–328. IEEE, 2017.

[61] Jinyong Wang and Ce Zhang. Software reliability prediction using a deep learning model based on the rnn encoder–decoder. *Reliability Engineering & System Safety*, 2017.

[62] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[63] Ke Wang, Zhendong Su, and Rishabh Singh. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations*, 2018.

[64] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[65] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[66] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM, 2018.

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

[68] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.

[69] Vincent J Hellendoorn and Premkumar 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, ser. ESEC/FSE*, pages 763–773, 2017.

[70] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *arXiv preprint arXiv:1709.06182*, 2017.

[71] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Tree2tree neural translation model for learning source code changes. *arXiv preprint arXiv:1810.00314*, 2018.

[72] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[73] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.

[74] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

[75] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[76] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211*, 2015.