Patching as Translation: The Data and the Metaphor

Yangruibo Ding
Columbia University
yangruibo.ding@columbia.edu

Baishakhi Ray
Columbia University
rayb@cs.columbia.edu

Premkumar Devanbu
University of California, Davis
ptdevanbu@ucdavis.edu

Vincent Hellendoorn
University of California, Davis
vhellendoorn@ucdavis.edu

ABSTRACT

Machine Learning models from other fields, like Computational Linguistics, have been transplanted to Software Engineering tasks, often quite successfully. Yet a transplanted model's initial success at a given task does not necessarily mean it is well-suited for the task. In this work, we examine a common example of this phenomenon: the conceit that "software patching is like language translation". We demonstrate empirically that there are subtle, but critical distinctions between sequence-to-sequence models and translation model: while program repair benefits greatly from the former, general modeling architecture, it actually suffers from design decisions built into the latter, both in terms of translation accuracy and diversity. Given these findings, we demonstrate how a more principled approach to model design, based on our empirical findings and general knowledge of software development, can lead to better solutions. We propose several models that leverage the same machine learning tools, but whose architecture, data presentation, and metrics are specialized for the software engineering task. The resulting models perform significantly better than the studied baseline, especially in more program repair appropriate metrics. Overall, our results demonstrate the merit of studying the intricacies of machine learned models in software engineering: not only can this help elucidate potential issues that may be overshadowed by increases in accuracy; it can also help innovate on these models to raise the state-of-the-art further. We will publicly release our replication data and materials at https://github.com/Anonymous-authors-2020/Patch-as-translation.git.

CCS CONCEPTS

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

KEYWORDS

neural machine translation, big code, sequence-to-sequence model, automated program repair

ASE '20, September 21–25, 2020, Melbourne, Australia

ACM ISBN 978-1-4503-XXXX-X/20/09…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent Hellendoorn. 2020. Patching as Translation: The Data and the Metaphor. In *ASE '20: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, September 21–25, 2020, Melbourne, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Recent work has applied a wide variety of machine learning models to practical software engineering tasks, including code completion, automated program repair, and code comment generation. These models excel at learning general patterns from large amounts of diverse data, even when training data is relatively unstructured. This combination enables one to simply *transplant* successful models from related fields, e.g. from computational linguistics, to software engineering. Yet, even if these models provide reasonable performance, the transplanted model may still not be appropriate for the task; many of these models were designed for paradigms that differ subtly, yet significantly.

In this work, we conduct a systematic empirical case-study to illustrate how transplanted models can fail in the targeted task domain, focusing specifically on the concept of "patching as translation" as a typical example of this phenomenon. A range of recent work has adopted neural machine translation (NMT) models to learn to repair programs by "translating" the buggy code to the repaired code [3, 4, 19, 27]. We argue that there are three general concerns with this type of approach, and show concretely how these manifest in "patching as translation" through empirical analysis:

Task design: Deep Learning (DL) models transform their inputs into a compact set of features that stores the important information, which it then uses to produce the required target. A wide range of DL architectures have been proposed that do so, but regardless of the specific architecture or task, it is self-evident that all the relevant information needed to generate the target must already exist in the input. While that is (largely) a fair assumption for natural language translation, where we can assume that the input & output sentences express the same idea, it is questionable for source code repair: we show evidence that buggy fragments often lack the information required to repair them. Reliably choosing the correct repair may even be impossible without access to a very broad context (including surrounding files), in the absence of which this task is inevitably ambiguous for many real-world bugs.

Architectural design: Given a task where deep learning is feasible, one must choose a model architecture that supports the transformation from input to output, in as realistic and simple a manner as possible. This is done by ensuring that prior knowledge of the

task (including dependencies and structural properties) are built into the model design. Architectures for machine translation rely heavily on the auto-regressive nature of text: language is generally produced one word (or token) at the time in left-to-right manner, e.g. in speech or writing; the standard NMT encoder-decoder architecture generates translations correspondingly. While this works very well for NMT, its relevance to practical program repair is tenuous at best: empirically, many repairs just copy (nearly all) tokens from the buggy line, with very few changed tokens (often just one). As such, both bug and patch share a large identical prefix, but the difference in the subsequent tokens is crucial. We demonstrate that models struggle to predict this transition, as the large amount of copying distracts (and inflates) the training quality signal.

Objective design: finally, models are trained by computing a loss for their predictions relative to a "gold" output, using a *loss function*. This function is usually a *differentiable* proxy for the actual quality of the model, because such qualitative assessments tend not to be differentiable. In machine translation, the training loss is usually based on the probabilities of the correct token; the actual quality of the trained model is measured with BLEU scores (or the like) that measure overlap between the generated and ground-truth translation. However, such overlap measures are inappropriate for program repair. For reasons stated earlier (few token changes, lack of contextual information), the quality of a produced repair often correlates very poorly with the number (and placement) of tokens it shares with the desired output. For instance, the trained model emits many syntactically incorrect repairs, as well as many very similar patches for a given bug, rather than exploring a range of alternatives. This yields poor performance in a search-based setting in which they are popularly used (given failing test cases).

Having studied these concerns empirically, we address them by designing a program repair tool for a given, localized bug from first-principles, incorporating the empirical and conceptual insights identified above. Specifically, we propose to add a substantial window of contextual information to the model and change the architecture of the model to predict insertions and deletions relative to the bug rather than the entire patch. The latter change especially produces a model that both generates the correct patch more often, and provides better sampling behavior (*i.e.,* higher top-K prediction accuracy using beam search). The contextual enhancement is less effective, which highlights that empirically observing an issue and effectively addressing it are not always the same; we leave this challenge for future work.

2 BACKGROUND

To study transplanting of architectures from neural machine translation (NMT) to automated program repair, we need to understand both domains. In this section, we first discuss NMT – its conceptual needs and corresponding architectural designs, and then Automated Program Repair – its empirical characteristics and practical use.

2.1 Neural Machine Translation

NMT aims to convert an expression in a source language (*e.g.,* English) into a semantically equivalent expression in a target language (e.g. French). This is generally both quite feasible and fairly deterministic: a given English sentence almost certainly has a French translation that is both a good French expression on its own and preserves all the information in the original English expression (*i.e.,* it could be translated back to a comparable English phrase).[1]

A natural fit for this task is the *encoder-decoder* architecture, which consists of two components: 1) an encoder that learns to compactly encode the important information from the source language expression, and 2) a decoder, which transforms that information into an equivalent expression in the target language. Encoder-decoder style models can address many types of transformations between two domains (*e.g.,* from image to textual description) and are typically instantiated with specific encoder-decoder architectures for a given problem that reflect some knowledge about that problem's domain. This simplifies the otherwise complex task of representing and producing a very wide range of inputs. For example, in computational linguistics, sequence-to-sequence *(seq2seq)* [25] models are a well-established way to generate text one token at a time, in a left-to-right manner – this linear order reflects how language is often generated in speech and writing. Seq2seq models exploit this structure by both representing and generating expressions with a strong emphasis on the left-to-right relations between tokens; especially in the decoder component, which (in nearly all popular models) produces tokens "auto-regressively", meaning that tokens are produced one by one, and every previously generated token is fed back to the model to produce the next token.

Practical seq2seq models. Seq2seq models have achieved great success in the NMT field. Recurrent Neural Networks (RNNs) were popular for many years, but had difficulties in remembering long-term dependencies. In an RNN, all the information of a source sentences is encoded into a hidden state from left to right; the final hidden state is then passed to the decoder, which attempts to reconstruct the target expression from this information. This puts inordinate strain on that single hidden state, which tends to cause the model to forget tokens seen longer ago. Long short-term memory (LSTM) [12] and gated recurrent unit (GRU) [5] were introduced to mitigate this bottle-neck by better separating long-term and token-specific information, and did significantly improve the performance of RNN-based NMT models, but ultimately suffered from the same concerns. Attention-based mechanisms [2] were introduced to allow the decoder to "attend" to any given intermediate state from the encoder (rather than only the final one), which greatly improved performance. Most recently, the Transformer [29] model generalized this idea to relying entirely on attention mechanisms to both encode inputs and generate outputs. The Transformer model proposes multi-headed (self-)attention interspersed with feed-forward networks that enables both encoder and decoder to attend to any set of tokens across arbitrarily long distances. These models are also highly parallelizable. We adopt this model in our work.

Seq2seq models in SE.. Hindle *et al* observed that source code is "natural" [11], *viz.,* with strong local dependencies similar to natural languages like English. Many language models have been applied to software engineering tasks. More recently, this includes a range of applications of the *seq2seq* architecture in modeling source code. Existing work has exploited their potential in several SE tasks,

[1] In practice, context is sometimes required, *e.g.,* to determine if an expression is meant sarcastically, which may alter its translation. There can also be multiple valid translations for one expression (*e.g.* literal vs. idiomatic). Even so, generated translations that overlap strongly with the ground-truth are rated highly by human translators [20].

such as code summarization [13], code migration [7] and program repair [3, 4, 19, 27]. The prevalent approach is to treat source code as a sequence of tokens with implicit or explicit structures (*e.g.,* abstract syntax trees) [3]. The encoder learns the distribution of such structured language, which is then translated into the target domain, either program languages (PL) or natural languages (NL).

Relevance of Models to Tasks. All these models excel at learning generalizable patterns from large amounts of diverse data and are *prima facie* at least somewhat applicable to source code, to the extent that it reflects natural language characteristics. However, different tasks come with their own concepts and peculiarities, and the models should reflect the phenomena specific to the task. For example, code summarization and code migration are more like NL translation tasks, since both their goals are to encode and preserve the semantics of their inputs (code fragments), just in different vocabularies (concise natural language, and another code context respectively). On the other hand, software engineers behave differently when repairing a program. Developers tend to fix a buggy fragment by making minor changes rather than entirely rewriting it. Furthermore, the semantics of the buggy fragment are by definition not preserved; the express goal is to introduce semantically new content (and possibly remove some) so as to change the meaning of a fragment. None of this disqualifies the use of seq2seq models *per se*, but its built-in assumptions should at least be carefully evaluated empirically, and, if necessary, its application should be changed to better reflect the domain.

2.2 Automated Program Repair

Automated program repair (APR) is a task of keen interest in SE. The aim is to fix software bugs with minimal human intervention. Classic APR techniques can be categorized into 1) generate-and-validate (G&V) [6, 14, 21, 22] or 2) synthesis-based approaches [17]. G&V approaches automatically generate patches and validate the candidates using a set of test cases that reveals the bug. To generate fixes, one effective approach is to mutate (*e.g.,* insert, replace) the buggy code according to code snippets in the current project that occur in similar contexts [14]. Synthesis-based approaches create constraints that satisfy all test cases, and then solve them and produce patches from the solutions.

NMT for APR.. Tufano *et al.* [27] proposed to use machine translation to repair programs and empirically studied the feasibility of translating buggy programs into fixed ones. They applied multi-layer RNNs with either LSTM or GRU nodes to predict patches of abstracted, real bugs, and report promising performance. Chen *et al.* [4] subsequently introduced *SequenceR*, an end-to-end framework to repair one-line Java bugs. They used NMT models to learn the implicit bug-repair patterns by training the model with 35k bug-fix pairs. Besides the buggy line itself, they also considered code context to allow long-range dependencies in fixes; they include the entire class in which the bug is located, which they *abstract* to reduce the input size. CODIT [3] developed a tree-based NMT model to produce code edits and bug fixes. It first translates the tree structure of code and utilizes the structural information to assist the generation of code tokens. CODIT also includes the tree nodes around the bug as context to predict meaningful patches. ENCORE [19] ensembles multiple NMT models to capture diverse fix patterns. The authors argue that incorporating context is essential for fixing bugs, yet ineffective for deep learning models, so they ignore the buggy context.

Although these existing works apply a wide range of models, they all treat program repair as a translation task; these tools encode a limited program window around a bug and learn to transform it to repaired code based on historical repairs. The premise is that translation is both a suitable model and that the buggy code (with its context) provides sufficient information to succeed. It is thus past time to ask the following, high-level question, which has yet to be addressed from an empirical perspective:

RQ1. *Is it generally feasible to translate buggy programs to repairs?*

While these approaches all indicate that Seq2Seq learning holds promise for learning patterns of transformation between bugs and patches, they struggle to outperform many G&V tools that applied human-designed rules to fix defects. One explanation is that the search space of repairs is prohibitively large [18], among others due to the large and highly local vocabulary and patterns endemic to software [9], as well as the length of buggy fragments. Intuitively, however, the search space need not be so large at all: in real-world development, modifications made to code during repair are mostly small, limited to a few tokens rather than completely reconstructing a statement. Is the reliance on translation putting models at a disadvantage by artificially expanding the search space? It is again worth determining this empirically, by asking:

RQ2. *Do machine translation architectures mischaracterize real-world fixing behavior, and does this disadvantage their performance?*

Deep Learning for APR.. Besides translating buggy code to fix it, recent work has proposed deep learning models that learn to specify the buggy locations that need to be modified together with the edits to be made. DeepFix [8] implemented a *seq2seq* attention network to fix compiler errors. As input, the program is represented as a sequence of (line number, tokens) pairs, and the model predicts a single single (buggy line number, patch) pair as a repair. Vasic *et al.* [28] proposed using pointer networks [30] to jointly learn to localize and repair a specific class of bugs known as VarMisuses. Their network jointly predicts two output "heads", one to locate the buggy token and one for its replacement. Tarlow *et al.* [26] introduced an edit-based model called *Graph2Diff* that uses a graph neural network as an encoder and a Transformer as decoder. This model transforms a program graph into a *ToCoPo* sequence of AST edits that transform the buggy program into a repaired version.

By directly learning the locations of incorrect tokens and the edits to be made, these edit-based methods provide an approach to learning bug fixing with a very different *loss*, which is not trivially reduced by maximizing the token overlap between the bug and repair. The impact of this loss can be substantial in determining the kind, and robustness, of local minima that the neural network finds during training. We thus implement a simple version of this model ourselves to empirically study the impact of the objective function on both our baseline model and this edit-based model. This way, we study the impact of the objective function on the models by studying *the models' results* themselves, asking the following:

RQ3. *How well does the NMT objective function apply to Automated Program Repair?*

3 METHODOLOGY

The goal of this work is to provide an empirical and conceptual analysis of the relevance of deep learning models (originally developed for NMT) in SE contexts. As such, we emphasize that it is *not our goal* to produce a state-of-the-art bug detector, or replicate prior work. Rather, we identify a general, representative approach (seq2seq for program repair), that reflects a direct adoption of models from a related field to SE tasks, and study its limitations. Naturally, prior work has covered a wide range of applications and modifications of this method, and may be immune to some of our findings, but this does not discount the general result of our analysis: that adapting deep-learning models designed for other fields to SE requires a principled, empirically and conceptually grounded approach.

3.1 Scope

Concretely, we focus on a relatively simple form of automated program repair in which we translate a given buggy line to its repaired counter-part. We thus assume that we have the bug already localized and that it is confined to exactly a single line. This is the most direct form of "repair as translation", in which an off-the-shelf translation model is used on two software "sentences": the buggy version and the repair.

3.2 Data

We collect our bugs from the history of the 10,235 most-starred Java repositories on Github on March 30th, 2020. We analyzed each project's entire commit history and extracted any commits that altered precisely a single line in a single Java file, disregarding any (spurious) changes to whitespace. We then compared the corresponding commit messages against a relatively simple keyword-based check [23] to heuristically find commits labeled as e.g. "fix" or "bug". We note that, although this heuristic is not particularly precise, the characteristics we found in our data were very similar between those marked as fixes and other one-line changes, so we expect this to have little impact on our analysis. This process resulted in *ca.* 60,000 bug fixes across 8,644 projects in our dataset. In the course of our analysis of this data, we manually checked and confirmed that most of these were indeed bugs.

3.3 Experiment Setup

Given the collected dataset, we first analyze the characteristics of real fixes and then train NMT models on these samples to predict patches. To answer our research questions, we study both characteristics of the the real-world bug-fixing behaviors and of the model-generated patches.

3.3.1 Bug context. We design experiments to explore the importance of a bug's lexical context when fixing defects. In natural languages, context (the text surrounding an expression) has a direct effect on the way people understand a specific expression and can help avoid ambiguity in communication. Similarly, the context of a buggy line is the code surrounding, as in, both preceding and succeeding, the bug. This can variously be chosen to include up to N lines of code above and below the bug, the surrounding function, or even the whole file (or project). This context can provide vital information (*e.g.,* variable definitions, conditional statements) for understanding the defect and the necessary repair. We study the role of variously sized contexts for both disambiguation and providing necessary information in section 4.1.

3.3.2 Similarity analysis. We noted earlier that software engineers tend to make small changes when fixing defects, likely because bugs correspond to only minor flaws in the code, and perhaps also because making few changes reduces the risk of introducing new bugs. Given this, we evaluate the similarity between real bugs and patches empirically across three similarity metrics:

(i) *Edit distance* (precisely, *Levenshtein distance*) is a metric that quantifies the difference of two sequences by the minimum number of edits (deletions, insertions, or substitutions) required to transform one into the other.

(ii) *Jaccard similarity* (effectively intersection-over-union) calculates the ratio of overlapping n-grams between two sequences divided by their union. Jaccard similarity is usually just applied to token-level similarity; we extend it to the average of 1 through 4-gram overlap to better capture both token-level similarity and matches in their ordering.

(iii) *Bilingual evaluation understudy* (BLEU) is popularly used to evaluate the quality of machine translations, and is considered to have a high correlation with human assessments of similarity [20]. This is an asymmetric measure that captures how similar the model prediction (the "hypothesis") is to the ground-truth translation (the "reference"). BLEU also counts n-gram matches between the prediction and the ground-truth, and normalizes these w.r.t. the predicted sequence length, which causes the asymmetry.

These three metrics above will be used frequently across our analysis to measure similarity in different aspects.

3.3.3 Model training and BPE. To inspect the performance of NMT models on program repair, we trained and evaluated a vanilla Transformer model [29] on our dataset. We split the whole dataset into three parts across organizations, TRAIN/VALID/TEST, with a ratio of 90%:5%:5%. We trained the model on the ca. 55K bugs in the TRAIN set, optimized it for held-out performance relative to the VALID set and finally evaluated the performance on TEST set.

In the field of natural language processing, *Byte-pair-encoding* (BPE) [24] is a widely used method to encode rare and long words into frequent sub-tokens; this way, tokens that were not seen in full during training can still be predicted accurately at inference time. BPE splits a word (*e.g.,* "coding") into a list of more frequent subtokens (*e.g.,* ['cod', 'ing']). In programming languages, vocabulary innovation is even more rampant, as developers tend to name a variable or method using a combination of words (*e.g.,* `isNullOrEmpty`) [9]. Karampatsis *et al.* [15] show that BPE can effectively address this issue in big code applications, so we apply this to our model input as and predictions as well.

4 ANALYSIS

In this section, we empirically analyse the characteristics of program repair on real-world bug fixes with a joint focus on the relation to natural language translation and on factors that influence accuracy for program repair. In particular, we study the characteristics of the ground-truth data (*i.e.,* the real bugs and patches with their context), and of the patches generated by our NMT model, as trained in Section 3.3.3. For the rest of this section, we scrutinize the adequacy of translation as a paradigm in Section 4.1, we identify

architectural concerns in Section 4.2, and we quantify their impact on model performance in Section 4.3.

4.1 Task Design

Translation, as a task, is intended to facilitate communication across language barriers. Hence, by design, it must preserve the semantics between the source language, and the target language—any translation that changes the semantics is unacceptable. By contrast, in program repair, the semantics of source and target are meant to differ, as the buggy version contains incorrect program behavior that the fixed version is supposed to correct. To do that, engineers deliberately change (add/delete/replace) incorrect tokens with correct ones. Imitating such changes with a machine learner is non-trivial, especially since the learner usually only has access to the bug and the fix, but not the knowledge latent in the developer's mind to reason about the fix.

For instance, the fix may introduce, *de novo*, tokens that are not in the buggy lines, *e.g.,* a new API call. In such cases, a model has to learn to pick those new tokens from across its entire known vocabulary. If the replacement is a common fix pattern, this might be easy enough to learn; otherwise, this leads to a vast search space of candidate repairs. The latter case is common enough; developers often use methods of their own creation, defined in adjacent files, or string or numerical patterns specific only to that project. It is thus important to *quantify*, even just approximately, how much information the model needs and how much it has access to from its training data, which tends to comprise the buggy lines and an optional window of code context. Although it is non-trivial to inspect the learned "black-box" model and extract what it infers about a given buggy line, we can identify the **gaps** between "what program repair needs" and "what machine translation can supply".

4.1.1 Program repair needs (lots of) contextual information. Patches that introduce new vocabulary (relative to the buggy line) require the model to conjure up novel tokens, *ex nihilo*. Given that code vocabulary is highly diverse and often strongly specific to a given project, package and file [9], doing so from the buggy line alone may require an unreasonable level of ingenuity from the model. Table 1 quantifies this: first, nearly 90% of patches introduce new vocabulary relative to their buggy source, which is true regardless of sub-tokenization (even using the BPE approach). Furthermore, these are not at all just typical program tokens or local variables; we paired the buggy line with increasing windows of context (explained in Section 3.3.1) and find that the unseen tokens introduced by the patch are still rarely borrowed from any immediate buggy context; they are sometimes present in the file as a whole, but in locations far away from the bug. Nearby tokens are a bit more likely to share some sub-token(s) with the patch, but rarely provide the entire missing link. Given that modeling large volumes of code (*i.e.,* many hundreds, or thousands of tokens) at once is often prohibitively expensive for current deep learned models, this can seriously affect models that incorporate only modest levels of context, such as the surrounding few lines or function.

This is not merely a matter of richer training data either; a large proportion of project-specific tokens are not found in any other projects [9], so it is quite unlikely that our model would have seen many of these at training time. We note that this is in contrast to other paradigms of program repair, such a many G&V models, which instead *search* for patches from across many surrounding files, rather than aim to encode a context directly into a translation.

Table 1: Ratio of patches with new vocabulary relative to the buggy snippet given a context window that ranges from *None* (*i.e.,* only buggy lines – a typical translation setting), to a given number of lines (symmetrically around the bug), and finally to the entire file.

Context included	Patches introducing unseen tokens	
	without BPE	with BPE
None	89.5%	86.2%
10 lines	73.0%	64.2%
20 lines	68.7%	59.9%
Whole file	49.5%	38.6%

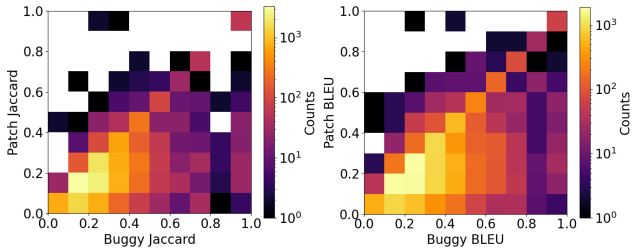
4.1.2 Without context, program repair is inherently ambiguous. As discussed, a learner would certainly struggle to capture enough information from the buggy program alone. Fortunately, these learners are equipped with the capacity to transfer many insights from their training data to new examples. Perhaps they can predict the missing semantic information from those bug-repair pairs?

Although it is again impossible to quantify what the model can do, we also again argue that it is perfectly sound to lower-bound its potential by estimating how much of the requisite information it has access to at training time. Concretely, the training data contains many "similar" bugs to those seen at test time (which we will quantify in various ways), so the model might learn the transformations that produced patches from those similar bugs and apply the same insight, *e.g.,* to predict the missing vocabulary. But, this is contingent on similar bugs indeed producing similar repairs; if repairs for similar bugs routinely diverge, then the model is reasoning about highly ambiguous data and will have to learn a wide range of valid transformations for a single defect in the training data.

To simplify this discussion, let b be a bug in the HELD-OUT portion of our dataset and p be the patch of b . Assuming we had some oracle that can provide "similar" bugs $\tilde{b}$ (with patch $\tilde{p}$) for b, specifically from our TRAINING data, we would ideally expect information about the relative change needed to repair b to be transplanted from $\tilde{p}$. If that is generally true, then our model can learn similar transformations for similar bugs and thereby generate new vocabulary and patterns that are not present in the buggy context.

To quantify this, we find the top-3 most similar bugs for each b in the HELD-OUT using 4-gram Jaccard index (see section 3.3.2), which we label $\tilde{b}_1$, $\tilde{b}_2$, $\tilde{b}_3$, among bugs in the TRAINING data. We then extract the corresponding p and $\tilde{p}_i$ and evaluate the patch similarity SIM$[p, \tilde{p}_i]$ in relation to the bug similarity SIM$[b, \tilde{b}_i]$. To facilitate transferring repair patterns, we should hope that similar bugs produce similar repairs. We visualize this as a heat map (Figure 1) to show the correlation between bugs' similarity and patches' similarity: for each grid in the heat map, we count the number of samples with SIM$[b, \tilde{b}_i]$ and SIM$[p, \tilde{p}_i]$ in the corresponding range, and then color the grid based on these counts. To make the color contrast more identifiable, we log-normalize the counts.

We calculate the $b \rightarrow \tilde{b}$ and $p \rightarrow \tilde{p}$ similarity scores using both the Jaccard index and BLEU scores; the latter is more appropriate for translation because it is asymmetrical, capturing the overlap



(a) Intersection-over-union **(b) BLEU**

Figure 1: Correlation between bugs similarity and patches similarity. X-axis indicates the bug/similar-bug similarity, and y-axis indicates the patch/similar-patch similarity. The corresponding count of each grid is normalized on a log scale.

Table 2: Similar bugs do not always have similar patches.

bug/similar-bug BLEU	# Samples	patch/similar-patches BLEU	
		< 0.5	≥ 0.5
≥ 0.5	2038	66.0%	34.0%
≥ 0.6	1143	62.3%	37.7%
≥ 0.7	561	54.2%	45.8%
≥ 0.8	258	46.1%	53.9%
1	173	49.1%	50.9%

from the perspective of the translation target. This aligns well with the task's directionality: we want to quantify what information is transferred from TRAINING to HELD-OUT, not vice versa. The result is shown in Figure 1; both metrics yield a similar pattern: bug-similarity only partially correlates with patch-similarity. Both graphs show a "smeared out" pattern in which similar bugs tend to produce patches with typically less similarity, rather than a strongly pronounced diagonal, that would indicate that patches relate to one another as their bugs do. Worse, many bugs have only neighbors with low similarity to begin with. These lower scores tend to just reflect spurious overlap due to the large portion of "closed-vocabulary" tokens (*e.g.,* brackets, keywords) in source code, which is also evident from the main hotspot being at (0.25, 0.25).

We are particularly interested in pairs that share a relatively large number of tokens and patterns; *i.e.,* those with similarity scores greater than 0.5. For example, the code: "`private boolean isName = false;`" and code: "`private boolean isName = true;`" yield a BLEU-score of 0.57, and they indeed look alike (only differ in boolean value). If similar bugs are (predominantly) fixed in similar ways, then we should expect that to translate into high patch similarity, which would allow the model to copy the appropriate repair patterns. Unfortunately, Table 2, which breaks down the highly similar bugs specifically, paints a different picture: here too, the similarity between bugs has nearly no discernible relation to that of their patches. Even highly similar bugs' patches do not score above 0.5 half the time, which is actually lower than their respective bugs. For instance, a common bug in our dataset, "`LOG.error(e);`", presents with many dissimilar patches including "`LOG.warn(e);`" and "`LOG.error("Can't read settings for " + tool, e);`". The BLEU score between these two patches is just 0.12, and we can tell that this bug was fixed with very different intentions. In other words, relying on similar bugs to transplant patch information is almost entirely ineffective.

This demonstrates a substantial *inherent ambiguity* in program repair based on just a buggy line (though not necessarily to program repair in general): for given a bug, the learned program repair history provides a mixed signal of many candidate repairs with distinct semantics. This matches our intuition as well: just how a given fragment is buggy, and what specific repair among many valid semantic transformations is appropriate depends on a vast array of factors, many of which are not enshrined in the code at all (*e.g.,* project requirements, developer preferences), let alone the buggy line (or even function) itself.

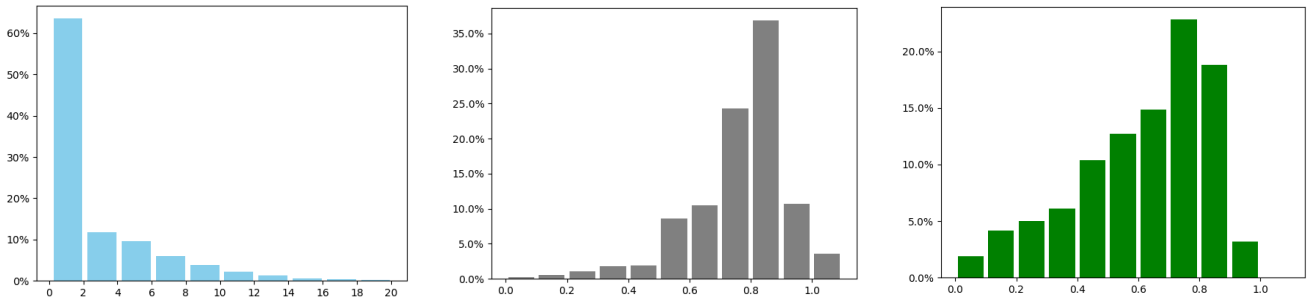
4.1.3 The challenges of new vocabulary. Finally, it might still be feasible for the model to "guess" at novel tokens and break the ambiguity if they can be constructed fairly obviously from the context, *e.g.,* by applying known transformations to existing ones, like converting singular to plural or incrementing a provided integer. Whereas the former results provided a lower-bound on what is feasible, it is quite impossible to quantify precisely what the model "could do", as the patterns learned by its millions of parameters can be highly complex. So we instead use the model's performance itself (studied in more depth in section 5) as an empirical datapoint: given that it is trained carefully and with ample capacity, we should expect that it provides at least evidence of this ability to produce correct new vocabulary. In contrast, we studied the trained model's accuracy on our 2,599 test samples; the patch introduced one or more tokens not present in the bug in over 75% of the cases, yet the model predicts this new vocabulary only 5.6% of the time. Worse, many of the "new" tokens are not even entirely new; they may just constitute the addition of `null` check, which the model still does not anticipate. Even when (beam) searching across the top 25 most-probable sampled patches, the model only anticipates 14% of the required new vocabulary. We stress that this is a well-trained model, which was able to achieve a high accuracy on its training data and for which we used the most generalizable checkpoint after training for 100 epochs. As such, machine translation models are already at a serious disadvantage here compared with NLP applications. This allows us to conclude our investigation of RQ1:

> The lack of information in the training data, vocabulary, and immediate context makes repairing as translation in its current form largely **infeasible**.

4.2 Architectural Design

Our second point of concern with translation models for program repair relates to the structural constraints assumed inherent in natural language generation: that text is auto-regressively produced left-to-right. This constraint is built in to the translation model's (sequence-to-sequence) architecture and implies that a simple adoption for program repair requires the model to output the entire repair, producing the correct token at each point.

The flaw with this particular decision is different from the one in Section 4.1 in that it does not affect the *feasibility* of the task (generating the entire repaired line is just as possible as *e.g.,* generating the change only). Instead, architectural mismatches between the model and the task impact the difficulty of training and the corresponding rate, and even the ultimate limit, of convergence on test data. This is because a) our models do not have infinite



(a) Edit Distance **(b) Intersection-over-union** **(c) BLEU**

Figure 2: Different Similarity Metrics regarding (bug, patch) pairs. X-axis of each histogram indicates the similarity score *w.r.t.* different metrics, and y-axis shows the ratio of samples within the corresponding range. The average edit distance between bugs and patches is 3.29. The edit distances of 51.1% samples are 1 and 63.6% samples have an edit distance ≤ 2. The average intersection-over-union similarity is 0.76, and 88.4% samples have a similarity ≥ 0.6. The average BLEU score is 0.61, and 72.5% samples have a BLEU score ≥ 0.5.

capacity and b) stochastic gradient descent is a local optimization; thus, these models tend to find a local minimum that matches the signal conveyed by the loss function. If this loss function prioritizes exact repetition of many tokens from the input, or a strong reliance on left-to-right production, this may negatively affect the actual quality (*e.g.,* overall accuracy) of the ultimate local minimum. In this section, we quantify this effect from the data statistics; in the next we explore it further based on the model's convergent quality.

4.2.1 The patch preserves most of the tokens in the bug. [2] Bug-fixing modifications to committed code are often minor; the buggy line usually is already *per se* a close approximation of the correct code, with very subtle, minor flaws. To quantify this assertion, we first measure the similarity between real bugs and patches. We use three different metrics to evaluate the similarity of each (bug, patch) pair in our dataset, outlined in section 3.3.2: token-level edit distance, (1-gram) intersection-over-union (which contributes a denominator to token-level overlap); and finally, mean BLEU-4 similarity to balance the overlap between tokens and sequences.

The results are shown in Figure 2. The average edit distance for the samples in our dataset is 3.29, but the distribution is long-tailed so this mean is somewhat inflated by the few large edits; the median distance is simply 1 – 51.1% of the samples only edit a single token to fix the bug, and 63.6% of the samples have an edit distance up to 2. Thus, bug fixing modifications are often limited to just a select few tokens. Figure 2b further shows that bugs and patches share the majority of their vocabulary as well: the average Jaccard similarity is 0.76, and half the time the patch reserves more than 80% of the bug's tokens. This overlap extends to sequences of tokens as well: the mean BLEU score of a patch relative to its bug is 0.61. Two lines of code are considered very similar when their BLEU score is greater than 0.5, so bugs and their patches overlap strongly. For reference, the state-of-the-art results in NMT at this time are ca. 0.4, depending on the language pair. Program repair achieves far higher performance by simply copying the bug verbatim; yet, doing so would in no way approximate a *good* repair.

This also confirms our intuition that bugs and patches are highly similar, and patches retain most tokens from the buggy version, rather than assembling code *de novo*. This principally suggests that searching for the correct patch token by token, from left to right is a poor use of search space; a smart program repair tool should just predict which tokens are supposed to be preserved and focus on searching for the ones that require modifications. But is it really so bad to generate the entire patch; wouldn't copying the preserved tokens simply prove no concern for the models? We answer this question in the negative in section 4.3; first, we further analyze the *types* of changes made in real repairs.

[2]This result applies to our study of small (one line) bug fixes; this may not hold for larger patches, which may be more likely to reconstruct the whole buggy module.

Table 3: Proportion of repairs in which the syntactic structures remains unchanged relative to bug, both for all samples in our training data and for those in which the patch both does and does not introduce novel tokens (relative to the bug).

Setting	Proportion
All bugs	52.4%
Patches introducing new tokens	56.2%
Patches without new tokens	20.6%

4.2.2 The patch tends to make minor changes to the bug's syntax. Grammars vary widely across languages. For example, subject-verb-object sequences ("I eat an apple") are abundant in English, but people seldom use them in verb-final languages like Tamil or Japanese. Because of this distinction, translating by merely substituting words in one language with another is often inappropriate. Instead, neural architectures capture the syntactic transformation between languages, as well as the translation of the underlying words. A recent study [1] shows that the difference in word order among various languages is a significant feature that models learn, and *e.g.,* neural attention mechanisms are effective at this task. We are similarly curious whether this feature is prominent in the conversion of bugs to patches, as such information gives hints about how to adapt machine translation models appropriately. For example, given a buggy line: `"if (level >= damage - damage / 2)"` with patch: `"if (level <= damage - damage / 2)"` (a real sample from our dataset), we can see that the patch does not modify the syntax (in terms of the AST) of the bug, but only changes its semantics by changing the underlying tokens. We thus empirically study how often modifications that fix logical errors introduce changes to the syntactic structure of code.

Given a pair of bug and patch, we tokenize the code and use *javalang* package [3] to identify the syntactic type (*e.g.,* identifier,

[3]https://github.com/c2nes/javalang

separator, integer) of each token. If a bug and patch have the exact same token type sequence, their syntactic structure is unchanged. Table 3 summarizes the resulting ratios. Slightly more than half of our patches preserve the exact syntactic structure of their bugs. Furthermore, the results are starkly different based on whether a patch introduces new tokens (relative to the bug; see section 4.1.1); those that do even more rarely (56.2%) change the syntax, while the other patches are often some kind of permutation of the bug's tokens that very rarely preserves syntactic ordering.

These observations have important implications. For one, they suggest that searching for unseen words across the entire vocabulary is rarely necessary; rather, the model could simply search for the tokens given a specific syntactic type [3]; *e.g.,* many patches replace just an operator to fix a bug. More generally, this suggests that the left-to-right generation process is thus not just inefficient, but all-but misdirected for such bugs: it requires the model to both copy a precise prefix, and then generate a single alternative from that context, where the original token was often already "close" to being correct (*i.e.,* in the right syntactic ballpark). Patching such bugs, more than half of those in our dataset, in this way likely puts inordinate and unnecessary strain on the model, which we will quantify in the subsequent sections. First, we partially conclude our second research question:

> The machine translation architecture's generation process is a poor fit for program repair, which frequently retains most tokens from the bug while replacing just a few, and from a small candidate set.

4.3 Program Repair via NMT (Objective)

Finally, when generating natural language translations, the goal is to correctly predict as many words of the target sentence as possible. The idea is that a translator that is likely to predict any one word given the input and previously predicted words (if any) is also likely to correctly generate the entire desired sentence by simply repeating this process until termination. Indeed, this tends to be quite accurate in general, in part because of the naturally auto-regressive, Markovian nature of text; a given prefix typically has only a small set of plausible continuations.

Given the observations in the preceding sections, this Markovian assumption seems precarious at best for program repair: the bug and repair often share a large, identical prefix (and suffix) that is then followed by incorrect tokens in the former and different, corrected ones in the latter. As such, we must question the validity of an objective function (both loss and metric) that values per-token prediction quality so strongly. Having said that, the aforementioned observations alone do not prove that there is a problem with this transplanted objective; the buggy token(s) may simply have been a particularly unnatural successor to its context [23], from which the corrected token(s) do, in fact, follow naturally. In this section, we empirically assess this concern.

Specifically, if the Markovian, auto-regressive objective used in natural language translation is a good fit for program repair as well, we would expect two things to be true:

(1) The per-token accuracy under auto-regressive teacher forcing correlates closely with the quality (*i.e.,* total accuracy) of the produced patch. That implies that the model correctly identifies the "challenging" tokens, that need to be altered, as these dictate the overall correctness of the resulting patch.
(2) The model efficiently explores the repair space when sampling multiple patches (*e.g.,* using beam search). That implies that choosing the repair point by first copying tokens unaltered and then (auto-regressively) generating a different continuation is no distraction to the model.

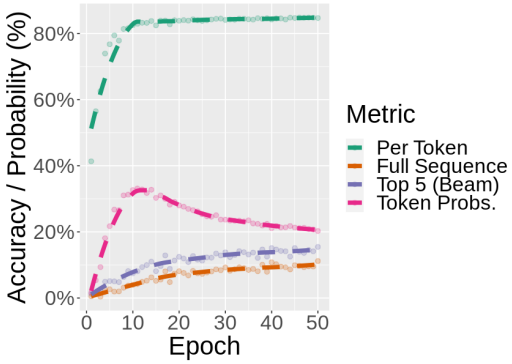


Figure 3: Performance trends (dashed line) and per-epoch results (points) on held-out data as training progresses, in terms of per-token accuracy subject to teacher forcing, accuracy at generating the complete sequence, and top-5 accuracy using beam search.

We put both these expectations to the (empirical) test. Figure 3 shows first the progression of various accuracies on our held-out data over the course of training. At the top, the teacher-forced token-level prediction accuracy increases steeply early on, throughout the first ca. 10 passes through the training data, but after that it all-but plateaus. It does, in fact, still increase, but only very slightly after epoch 10 (from ca. 83% to 85%). This clearly shows two "phases" (a bimodal pattern) in training this type of model: the model first trivially minimizes its loss (and thus achieves a high accuracy) through simple copying, but then struggles to match that strategy with predicting the correct change to achieve any more progress.

This initial copying translates into little real accuracy; the Full Sequence (*i.e.,* complete repair) prediction reaches just 4.5% after 10 epochs, making nearly all its substantial progress afterwards. This has real training ramifications: we also visualize the progression of the per-token entropy (transformed to probabilities). In the first 10 epochs, the model quickly becomes very polarized, assigning high probability to the copied tokens; then it becomes clear that this yields very low probabilities for the few changed tokens, which entropy penalizes strongly. As a response, the model instead adopts a more balanced prediction to achieve higher overall repair quality.

To quantify the correlations in the face of this bimodality, a non-parametric (Spearman's rho) correlation test is in order. This does show that the two metrics (per-token accuracy and full sequence accuracy) are highly correlated ($\rho = 0.914$), even, though less so, after epoch 10 ($\rho = 0.863$). The latter result reflects that the remaining per-token accuracy increase translates into a disproportionally higher complete repair rate – the missing 2% token accuracy becomes ca. 7% complete repair accuracy, nearly triple the levels at epoch 10. This implies a mixed answer to our first premise: the complete patch accuracy certainly follows the per-token accuracy, but

the relation is far from direct and the latter is a highly misleading metric *in ipso* due to its bimodal nature.

Finally, the figure shows that the odds of finding the correct patch in the top-5 generated samples is only a little higher than the top-1 prediction (ca. 5% points at most); that gap actually shrinks as the top-1 prediction becomes more accurate, which suggests that the beam search finds few good novel/alternative patches. We would hope that, given the natural ambiguity in choosing the correct patch, the model learns to sample a diverse set of plausible corrections. Instead, from inspection of the generated samples, the model produces many very similar candidates, usually differing by just a few tokens. This too is likely an artifact of the training criteria, which prioritizes copying 80% of the tokens over predicting the correct variation. Thus, we answer our final research question:

> The objective functions of NMT models are **inappropriate** for program repair, leading to reduced training efficacy on more appropriate metrics.

5 SEQ2SEQ MODEL FOR PROGRAM REPAIR

As a fitting conclusion to our empirical and conceptual evaluation of the basic "transplanted" approach to program repair as translation, it is appropriate to try and redesign the existing approach. This section demonstrates how observing and quantifying issues with an outside approach relates to principled and innovative modeling design: while observing concerns does not guarantee that improvements are straightforward (as we show in relation to context), it can improve performance by better relating the model to the task. We do this below, by eschewing past practice of trying to generate patch tokens directly, and instead generating *edits*.

5.1 Model Changes

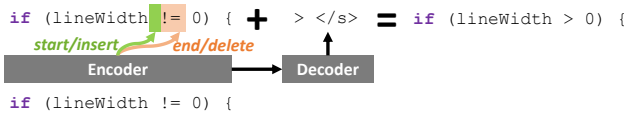
We observed three main deficiencies with the existing translation approach: the inadequacy of relying on just the bug for enough information to produce a patch, the mismatch between typical repair actions and generating the entire corrected line, and the related divergence of training-objective, between per-token accuracy and whole-repair (both top-1 and beam-search) accuracy. Had we designed a machine learning approach for this problem from scratch, we would certainly attempt to incorporate both bug context, and a notion of repair *edits* to reflect these aspects of program repair, as has also been proposed by some recent work [26]. We propose to make both changes: Figure 4 shows the two main architectural mechanisms we add to the base model to achieve this.

Edits: we model edits directly, as a token-level "diff" between the bug and patch. Our analysis of typical changes indicated that the bug and patch nearly always share a substantial prefix and/or suffix, with the repair occurring at some point in the middle of the line. We thus parse each bug/patch pair and find the longest overlapping prefix and suffix. Our model is augmented with two additional *pointers* that correspond to insertion and deletion; the original decoder component (of the encoder-decoder architecture) is now pressed into service to output the diff (rather than directly generate the raw tokens in the fix). There are three possible scenarios:

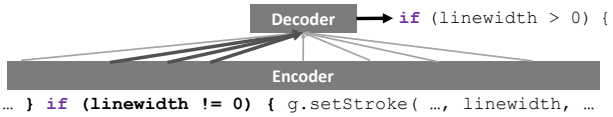
No additions: The prefix and suffix combined span the entire bug. This means that only tokens were added in the patch. In this case, the deletion pointer will just point to the start of the line, and the insertion pointer will indicate where the new tokens (which the decoder will emit) are to be added.

No removals: The prefix and suffix combined span the entire repair. In this case, only token deletions are needed, to go from the bug to the patch. So, the insertion and deletion pointer should correspond to the start and end of the segment to be deleted within the buggy statement, and the decoder should just emit the "</s>" termination symbol (an "empty" patch).

Additions & Deletions: A non-trivial change in both bug and patch. As a combination of the above, the two pointers should identify the segment to erase from the bug, while the decoder should generate all newly required tokens to insert instead.



(a) An edit-based repair model, which emits two pointers based on the encoder states that indicate the insertion start position and the removal end position. The decoder generates any missing tokens.



(b) Representation of a context-enriched repair model. The encoder functions as usual on a broader set of tokens; the decoder's attention is biased towards the highlighted (buggy) tokens.

Figure 4: Proposed architectural changes to the basic repair model on an example from our test data.

Context: we also observed that the bug alone rarely provides enough (syntactic and semantic) information to reliably predict the necessary repair. The natural solution is to add a large amount of contextual tokens from the file containing the bug. Unfortunately, Transformers struggle to model very long sequences as their memory usage increases quadratically with sequence length. At the same time, section 4.2 showed that even 20 lines of context is rarely enough to provide much missing vocabulary (which is itself only part of the information needed). We do not provide a new solution in this paper; rather, we empirically quantify the deficit from the model's perspective by adding up to 500 tokens of context and comparing the resulting performance. We ensure that the model is "aware" of which tokens to repair by biasing the decoder's attention to the buggy tokens using the same biasing mechanism as in [10], in this case with a simple unary relation (*i.e.*, "is part of the bug").

5.2 Results

As table 4 shows, there are two main characteristics of the resulting models' performances. First, the edit-based enhancement clearly and substantially improves the accuracy over the baseline model, fixing an additional 22 bugs on our test set with its top prediction alone. Second, the contextual enhancement does not seem to help in its current form. We discuss both these results here.

Table 4: Repair accuracy on the (de-duplicated) test data of the various models that we propose in this paper.

Model	Top-1	Top-5	Top-25
Baseline	3.30%	5.96%	**8.20%**
Edits	**4.31%**	**6.14%**	7.83%
Context	1.83%	3.57%	5.18%
*Edits + Context**	3.39%	4.08%	4.76%

*At the time of this writing, this model completed just 50 epochs compared to 100 for the others; but, we do not expect its results to change much.

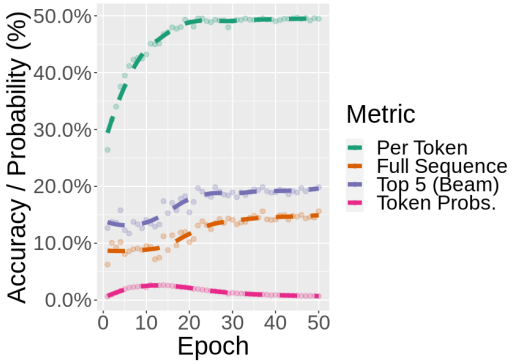


Figure 5: Performance of the edit-based model on held-out data as training progresses, in terms of per-token accuracy subject to teacher forcing, accuracy at generating the complete sequence, and top-5 accuracy using beam search.

Edit model: the edit-based model produces better-quality patches on our test data than the corresponding baseline. Its design is informed by our data analysis, and so it is arguably a better fit for this task. Figure 5 shows its training behavior, to compare with that of the baseline model in fig. 3; its "per-token", teacher-forced accuracy increases much more smoothly[4] and more in line with increases in the full repair prediction quality. It also displays a larger improvement in sampling accuracy between the top-1 and top-5 prediction, which remains consistently wide during training, suggesting that it better explores the search space with more diverse predictions.

Its design also allows the edit model to predict more newly introduced vocabulary in the patch relative to the bug; it does so 6.8% and 15.7% of the time (for the top-1 and top-25 samples respectively), compared to 5.6% and 14% of the base model. One notable difference is the gap between top-5 and top-25 sampling accuracy; the edit model is stronger in the former, but loses to the baseline in the latter. This appears to be due to the edit model having to commit to an insert & delete pointer first, conditional upon which sampling is more bounded. To be clear, we did also sample these two pointers from their corresponding probability distributions and initialized the beam search with the 25 most probable different combinations of start and end pointers; but, in practice the model tended to choose a single pair with very high probability, so that it effectively only explored that set. This may be an interesting issue to pursue in future work.

Context information: the second missing element was the reliance on the bug alone as a source of patching information; in section 4.1, we showed that the absence of context is an insurmountable obstacle that deprives the model of the necessary information to patch most bugs. However, identifying a problem and solving it are quite different things, as our results in table 4 show. Although we added a substantial amount of surrounding tokens (*i.e.,* 500) to the model's input, the resulting models' performance is quite poor, actually performing slightly worse than their context-free counterparts. This is likely due to the challenge of modeling large amounts of contextual information; although our models were trained to similar accuracies, they did so much slower and evidently with worse generalization.

This may point at several issues, but none seem quite responsible. For one, the attention mechanism we used may not adequately help the model locate the buggy bits; however, the model always emitted patches that were very similar to the bug. Similarly, the amount of context may simply be too little; table 1 suggests that many useful tokens are only available far away from the bug. However, that table also implies that the immediate context *should* help with ca 10-20% of missing tokens, so this too does not explain the lack of performance. The model itself may simply have insufficient capacity to capture this much context, though we used a relatively large Transformer architecture and the model was trained to high accuracy. All this is to say that we do not know how to better integrate context in these models. This is not a bad thing; not all modeling improvements are obvious, but it is important that we understand the deficits first. Our empirical analysis helped us both identify it, and has laid a useful foundation for the kind of information to integrate in further improvements, even if it is not yet clear how.

6 THREATS TO VALIDITY

This paper presents a case-study of a specific type of program repair, which we explore in great empirical detail. As such, the main threats to the validity of our conclusion are external, relating to the generalization of our findings to both other types of defects and other model "transplants" into SE research.

First, our data collection and analysis focused only on small, one-line fixes, since such bugs (and single-statement bugs) are both common and important, realistic target to current program repair models [16]. In addition, many existing NMT-based program-repair tools [4, 19] are trained and tested on one-line bugs. As such, studying such bugs is both representative and impactful. Having said that, we do not claim, nor believe, that their empirical properties generalize to larger, more complex defects; these no doubt have their own non-trivial characteristics that deserve further investigation, especially if/when they become the subject of new models.

Secondly, we did not compare our model(s) Section 5 with state-of-the-art, NMT-based, program repair tools. The goal of this work is not to present models with the best performance; rather, we are evaluating the feasibility of the general idea of "patching as translation" using a general, representative modeling setup, especially in contrast to variations that depart from the translation metaphor. More broadly, there are many other cases of modeling transplants into our community, often with some alterations to fit the task; these may not all be harmful or mismatched, but they do all deserve careful empirical analysis to ensure that they achieve their potential efficacy in our community.

[4]Their probability also displaying less of a "spike" in early training.

7 CONCLUSION

In this work, we first present a comprehensive study to evaluate the conceit that "software patching is like the language translation" as a prototypical example of "model transplant" from neighboring communities into SE. We empirically show that the translation paradigm does not capture bug-fixing very well for a range of reasons. We also use models themselves as empirical devices; we adapt the *seq2seq* models used for translation to generate edits rather than raw tokens, which leads to promising improvements. We hope this work inspires more empirically-grounded research into transplanting machine learning models to program repair, and other software engineering applications.

REFERENCES

[1] Wasi Uddin Ahmad, Zhisong Zhang, Xuezhe Ma, Eduard Hovy, Kai-Wei Chang, and Nanyun Peng. 2019. On Difficulties of Cross-Lingual Transfer with Order Differences: A Case Study on Dependency Parsing. In *NAACL*.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2015).

[3] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. CODIT: Code Editing with Tree-Based NeuralMachine Translation. arXiv:cs.SE/1810.00314

[4] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transaction on Software Engineering* (2019).

[5] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *ArXiv* abs/1409.1259 (2014).

[6] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72.

[7] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-Modal Sequence to Sequence Learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 3675–3681.

[8] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI*. 1345–1351.

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

[10] Vincent J Hellendoorn, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. 2020. Global relational models of source code. In *2020 8th International Conference on Learning Representations (ICLR)*.

[11] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[13] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[14] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code *(ISSTA)*. https://doi.org/10.1145/3213846.3213871

[15] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. arXiv:cs.SE/2003.07914

[16] Rafael-Michael Karampatsis and Charles Sutton. 2019. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. arXiv:cs.SE/1905.13334

[17] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[18] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*.

[19] Thibaud Lutellier, Lawrence Pang, Viet Hung Pham, Moshi Wei, and Lin Tan. 2019. ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair. arXiv:cs.SE/1906.08691

[20] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[21] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[22] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[23] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[24] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112.

[26] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2019. Learning to Fix Build Errors with Graph2Diff Neural Networks. *arXiv preprint arXiv:1911.01205* (2019).

[27] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering*.

[28] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Singh Rishabh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair.. In *ICLR*.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[30] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer network.. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*. 2692–2700.