Code Patching using Transformer Models

Hariharan Sezhiyan
Fernando Pieressa

June 2019

1 Task Description

Code patching, or debugging, is a very common task among developers. Professional developers spend around 70% of their time debugging. Moreso, of the approximately 70 buggy lines of code that developers write for every 1000 lines, 15 make it to production. Compilers and interpreters have been designed with this in mind, often providing programmers with error-fixing tips. However, these methods have little ability to reason about the error.

Neural networks have become a candidate for bug patching. Numerous researchers have applied recurrent and sequence-to-sequence models on buggy lines, hoping to get the model to translate a buggy line to a bug-free line. This paper discusses the use of a Transformer model, which has no recurrent components, to translate buggy lines to bug-free lines. Transformer models have been known to work better than other models when dealing with long-distance dependencies. Reasoning about code patches often requires considering long-distance dependencies.

2 Prior Work

Two key papers set the precedent for this project: *DeepFix: Fixing Common C Language Errors by Deep Learning* [2] and *On Learning Meaningful Code Changes via Neural Machine Translation* [3].

The DeepFix paper involved using a multilayer sequence-to-sequence models with GRU units to translate a buggy line to a bug-free line. Similair to our project, they first mined code, looking at files before and after a pull request and extracted the changed lines (which are considered buggy). They used an 'Oracle' model, in which their model first localizes the error, suggests a fix, and either accepts or denies the fix by checking if the total number of compiler errors is reduced. This technique is referred to 'iterative repair'. This model completely fixed 27% of programs in the dataset, partially fixed a further 19%, and reduced the total number of error messages by 32%.

The latter paper used a neural machine translation model consisting of a stack of encoder-decoder RNNs to translate a buggy line to a bug-free one. In constrast to DeepFix, this model does not work by iterative repair. Instead, it localizes errors at various levels of context (lines, methods, classes, files, etc), and fixes errors within these levels of context. The model is shown to produce meaningful code changes in addition to basic bug fixes. For example, it is able to make functions cleaner through refactoring redudant code and poorly named variables.

To our knowledge, this project is the first to use a full Transformer model in a code patching task.

3 Data Gathered

3.1 Initial Attempts

Our first attempt was with a dataset that contained commit information on various Github projects, with around 12,000 different entries. With this commit information, it was possible to check in a browser the specific line changes. We used a webscrapper with the `BeautifulSoup` library to obtain line changes in a format we could use in our model.

After scrapping and parsing the data, we realized two big flaws with the data:

- **Multiple line changes:** We were looking for one line changes to reduce the complexity of the problem. Around 50% of the data from this dataset involved multi-line changes, and by filtering them, we ended up with a database too small to train with.
- **Multiple languages:** Within this already reduced dataset, we realized that the one line changes were in commits written in different programming languages. Having the Transformer model learn different languages, detect syntax structures in those languages, and code patch would have been a hard challenge.

3.2 Final Database

We decided to manually select Github repositories we knew were written in a specific language. We chose Java, considering its popularity and adoption in the programming community. For each repository we selected, we had to collect every single commit, and keep only one-line changes. This could be obtained by checking those commits which had just 1 removed and 1 added line.

We also looked for specific keywords in the commit message to ensure the line changes represented meaningful code changes (as opposed to comments). We

obtained this insight from this paper: [1]. We used the following keywords: "fix", "bug", "typo", "error", "mistake", "fault", "defect", "flaw", "incorrect".

Then, we just needed to find enough repositories to have a good amount of data entries to train the model. We ended with a total of 90,111 entries used for training and 15,899 used for testing, corresponding to 15% being used for testing and the rest for training.

We processed this data using 4 different files: "train.enc", "train.dec", "test.enc" and "test.dec". Both .enc, .dec files had in each line the representation of one commit line change, where .enc files represent the line prior to the change and .dec files represent the line after the change. In this way, the code patching problem can be modeled as a translation problem, from buggy code to working code.

4 Results

4.1 Initial Attempts

At first we based our Transformer model on the following implementation: repository. We initially tried a large model with the following hyperparameters: 512 dimension size, 4 layers, and batch size of 128. This resulted in extreme overfitting, reaching a loss of 0 in training within the first 20,000 iterations but a BLEU of only 0%. By adding BPE (byte pair encoding) with a vocabulary size of 5,000, we managed to reach a 0.4% BLEU score at 4k iterations, but which quickly fell to 0%.

We then proceeded to reduce the model size drastically to see if it would improve the performance. The new hyperparameters were: 3 layers, 30 dimension size, and a batch size of 16. This model managed to obtain slightly better results, with better loss values for both training and testing with a BLEU reaching and settling at 10% after 10k iterations. Nonetheless, the predictions by the model were very poor, often resulting in code only loosely resembling the buggy line.

We realized that, unlike language translation, one-line code translation often only results in 1 or 2 token changes. For example, a line could involve a data type declaration change, which is simply a 1 token change (e.g. int to float). To help our model learn this, we trained the model to first predict itself (no change) for the first 120,000 steps, and then learn to predict changes at a finer level. We hoped this would allow the model to preserve the syntax of the commit line, and then properly predict. However, the results were not good, and can be seen in figure 2.

4.2 Final Model

We then changed to the Tensor2Tensor implementation of the Transformer model. In order to use this model, we had to change the source code of the Tensor2Tensor model in order to download our data. Further details of this procedure are discussed in the repository.

We used the base single-GPU Transformer model with all the standard hyper-parameters provided, and trained for 100,000 steps. All variants we tried are presented here:

- **Base Model with Beam Size 4** Direct use of the Tensor2Tensor Transformer model.
- **Base Model with Beam Size 10:** Base model, but a larger beam size during the decoding phase.
- **Base Model with BPE:** Base model with byte-pair encoding with a vocabulary size of 2,000. Software is unique from natural language in that the vocabularly size is almost unlimited. Developers routinely create new identifiers. To combat this, we reduced the vocabulary to 2,000 unique tokens.
- **Base Model with 2 line Context:** Base model with context. In addition to the buggy-line, 2 lines of context are also fed into the model for both training and testing. We hoped this context would allow the model to reason about the cause of the bug.
- **Base Model with 1 line Context:** Same as the above model, but only 1 line of context is fed in as input.

4.3 Evaluation

We considered two metrics in evaluating the models provided: BLEU and prediction accuracy. BLEU is used as a standard for various translation tasks. To measure prediction accuracy, we randomly chose 1,000 lines in our test set, and translated the buggy-lines to bug-free lines. We measured prediction accuracy by measuring the number of buggy-lines that were perfectly fixed (the translation was indistinguishable from the reference).

4.4 Final Model Results

Model	Beam Size	BLEU	Fix Accuracy
Base Model	4	77.9	53.3
Base Model, Beam Size 10	10	77.9	53.3
Base Model, with BPE (2k vocab)	4	82.3	54.6
Base Model, 1 line context	4	84.3	35.5
Base Model, 2 line context	4	84.8	33.5

4.5 Analysis of Results

To our surprise, the base model performed very well. At 100,000 steps, it was evaluated at 53.3% prediction accuracy. This result clearly indicates the Transformer model is able to learn about syntax and correctly fix errors unatainable by compilers and interpreters.

Another interesting result was that beam size had no impact on fix accuracy. For this reason, we continued to use a beam size of 4.

For the 1 line context model, we passed in the buggy line and 1 line above it. For the 2 line context model, we passed in the buggy line and 2 lines above and below it. We reasoned this context would allow the model to contextualize the bug. The results were poor for two reasons. First, the model needs to predict more text. Since all of this additional text is the same as the input, the BLEU score increases, leading the illusion of a better model. Second, the 1 and 2 lines of additional text was not enough context for the model to solve logical errors not already solved by the base model. The can be seen by the fact that the 1 line context model does even better than the 2 line one.

The best model was the base model with BPE. We restricted vocabalury size to 2,000 unique tokens using the SentencePiece public repository. We reason this model was best because it utilized the long-distance dependency properties with a smaller, more manageable vocabulary size.

Interestingly, for all models, when the model was not able to produce a fix, it exactly returned the input. This will be useful for further extensions of this project into actual developer tools because if the model is unable to make a decision, it can simply return the input and let the developer know.

4.6 Case Studies

This section will explore some the bugs at least one of our 5 models is able to fix, and those which none are able to fix. Input denotes the input to the model (the pre-PR line) and reference denotes the correct translation (the post-PR line).

Successful Translations

Input: @Exported (name = "str")
Reference: @Exported (name = "str" , inline = true)
Model Output: @Exported (name = "str" , inline = true)

Input: String txt = yytext ( ) ;
Reference: }
Model Output: }

Input: public static String getDefaultAlias (String aSourceName)
Reference: public static String getDefaultAlias (String sourceName)
Model Output: public static String getDefaultAlias (String sourceName)

Input: Map <String , DetectorNode >nodeMap = new HashMap <String , DetectorNode >( ) ;
Reference: Map <String , DetectorNode >nodeMap = new HashMap <>( ) ;
Model Output: Map <String , DetectorNode >nodeMap = new HashMap <>( ) ;

Input: ArrayList <T >list = new ArrayList <T >( ) ;
Reference: ArrayList <T >list = new ArrayList <>( ) ;
Model Output: ArrayList <T >list = new ArrayList <>( ) ;

Input: Setting . byteSizeSetting ("str" , new ByteSizeValue (32 , ByteSizeUnit . KB) , Property . NodeScope) ;
Reference: Setting . byteSizeSetting ("str" , new ByteSizeValue (64 , ByteSizeUnit . KB) , Property . NodeScope) ;
Model Output: Setting . byteSizeSetting ("str" , new ByteSizeValue (64 , ByteSizeUnit . KB) , Property . NodeScope) ;

Analysis of Successful Translations

These 4 error fixes presented are by no means the full extent of the model. A full file of translations is available in the repository. These examples do indicate the model is able to learn:

- **Import/Export Syntax:** The first example includes the use of the @Exported annotated type. The model is correctly able to recognize that this line is buggy because the designated function is an inline function. This bug fix will help developers write code that does not need to be compiled independently.
- **End of Function:** The second example reveals how the model is able reason that a line of code is unnecesary, and returns '}'. This ends control of the function.

- **Variable Naming:** The third example does not constitute a synatx or logical error, but instead a code-style error. The developer named the parameter variable 'aSourceName', which is not conventional. The model correctly returned 'sourceName'.

- **Constructor Initialization:** The next two examples reveal the model is able to reason about syntactic structures. In particular, it is able to correctly initialize two objects with constructors: a Map object and an ArrayList object.

- **Common Logical Errors:** The last example is a surprising result. The model is able to fix common logical changes, like changing a set length from 32 bytes to 64 bytes. On would think this is a fix that requires more context, but the model recognizes how often this change is made in the dataset, and accordingly adjusts.

Failure Modes

Input: private void checkIfClosed ( ) throws IOException {
Reference: private void checkIfClosed ( ) {
Model Output: private void checkIfClosed ( ) throws IOException {

Input: addMessage (new Message (MessageKind . INFORMATION , model . getModelUID ( ) + "str" + (genOK ? "str" : "str"))) ;
Reference: addMessage (new Message (genOK ? MessageKind . INFORMATION : MessageKind . WARNING , model . getModelUID ( ) + "str" + (genOK ? "str" : "str"))) ;
Model Output: addMessage (new Message (MessageKind . INFORMATION , model . getModelUID ( ) + "str" + (genOK ? "str" : "str"))) ;

Input: return 1 ;
Reference: return curSelection = = null ? 0 : 1 ;
Model Output: return 2 ;

Analysis of Failed Translations

These three errors are not comprehensize of all issues with the models presented, but highlight some key failure modes:

- **Deleting Code:** Consistently, we notice our model fails to delete code which is deemed unnecesary by the developer. We think this may be because the majority of code changes during a pull request involve code addition, not deletion.

- **Complex Logical:** Currently, even with a Transformer model, we are not able to fix long, complex logical statements. These issues may require a deeper architecture, a larger, more diverse training dataset, or both.

- **Return Logic:** Return statements come at the very end of a function, method, or program. They represent the overarching logic behind the written code, and to reason about them will likely require deeper, more advanced architectures.

5 Conclusion and Future Work

This work clearly shows that Transformer models are well suited for code patching. Their ability to reason with long-distance dependencies seemed promising in theory, and these experiments prove they indeed work in practice.

In addition to mere syntactic fixes, these models are able to fix some style and even basic logical errors. We hope those who come after us will be able to further refine these models to solve increasingly complex bugs.

Finally, we would like to express our gratitude to our principle advisors, Professor Premkumar Devanbu and PhD Candidate Vincent Hellendoorn, without whom this work would not have been possible.

6 Bibliography

[1] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Premkumar Devanbu, Alberto Bacchelli, Zhaopeng Tu, *On the Naturalness of Buggy Code*

[2] Rahul Gupta, Soham Pal, Aditya Kanade, Shirish Shevade, *DeepFix: Fixing Common C Language Errors by Deep Learning*

[3] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, Denys Poshyvanyk, *On Learning Meaningful Code Changes via Neural Machine Translation*

7 Figures

Hyperparameter	**Value**
# of training steps	100,000
# of hidden layers	6
# of heads	8
batch size	1024
learning rate	0.2, with noam decay
relu dropout	0.1
attention dropout	0.1

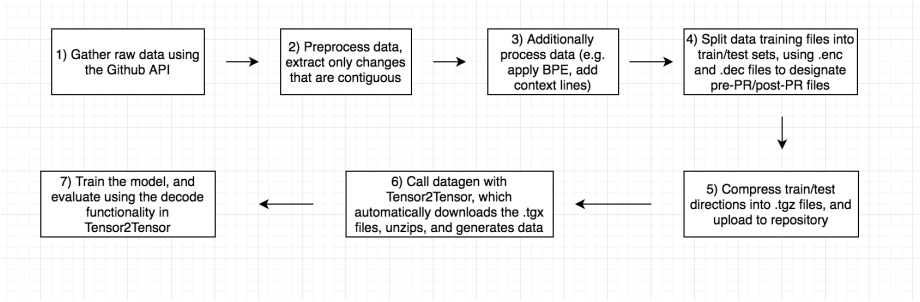

Figure 1: The overall dataflow, from raw data acquisition to training the model and evaluating results.

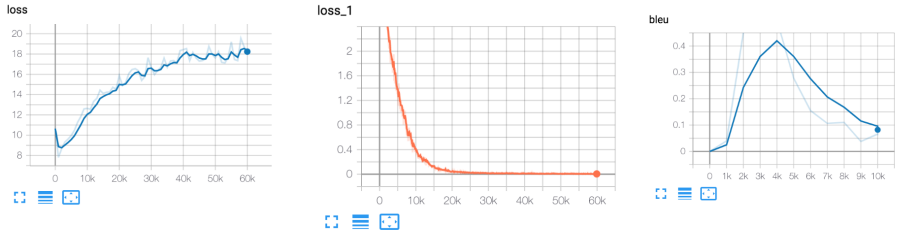

Figure 2: Small model results. 'loss' represents training loss and 'loss_1' represents testing loss. Clear overfitting seen in this figure.

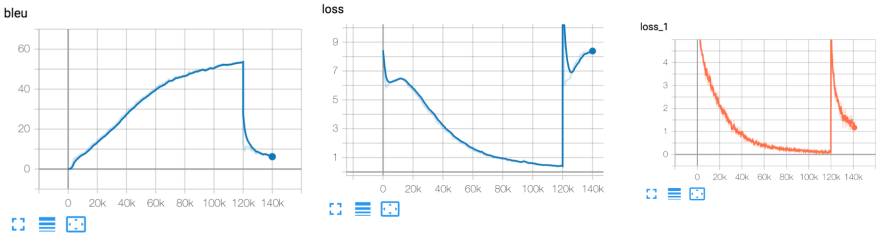

Figure 3: Pretraing results. All metrics (BLEU, loss) initially are healthy, but after pretraining ends at 120k steps, degrade rapidly.

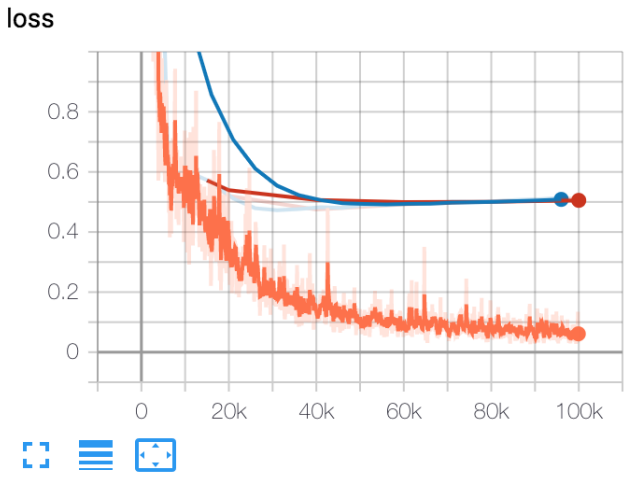

Figure 4: Train/test losses from final base model. We used two different eval/test sets, so a total of 3 curve are shown

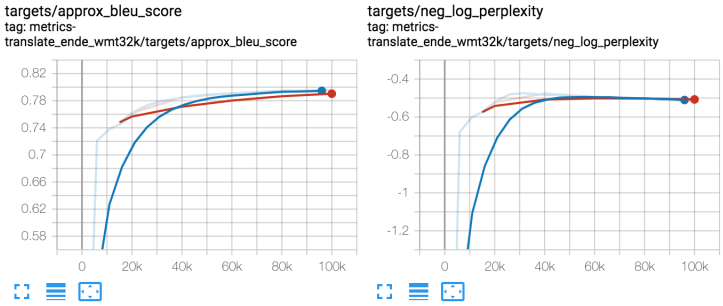

Figure 5: BLEU scores and perplexity from the final base model

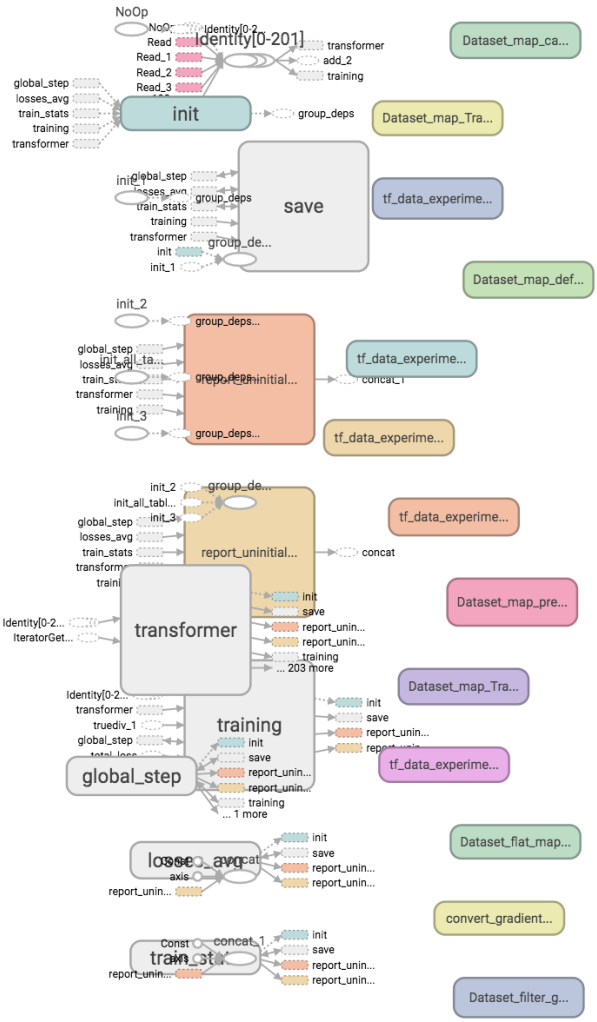

Figure 6: Overall graph structure of the final base model.