Automatic Patch Generation System Using Deep Learning Techniques

[1]S. Ashok Kumar, [2]D. Arvind, [2]S. Dharanikumar, [2]S. Premkumar

[1]Assistant Professor, Department of Computer Science & Engineering, Manakula Vinayagar Institute of Technology, Puducherry,

[2]Student, Department of Computer Science & Engineering, Manakula Vinayagar Institute of Technology, Puducherry

Abstract- Many software and applications are being developed on a daily basis; millions of lines of code are being written daily, thousands and thousands of bugs are fixed daily. Bugs are the common enemies for developers, the larger the code base of software the more troublesome they are. The possibility is very high that a bug. Bug fixing normally takes a lot of human resources and time to fix it. A developer is said to spend a substantial amount of time in fixing a bug. Though there are sites like Stack Overflow, a developers community which is like an encyclopaedia for developers, doesn't help the developers in finding the right answers all the time, and mostly the amount of time a developer spends in finding a fix for the bugs are a lot. So a tool which can suggest a fix for the bugs in the program for the programmers can be of a great value. The easiest way is to manually record the bugs and fix used for it, by classifying them can make much sense but it is a tedious process to save the bug and the fix used and finding the right fix may take a solid amount of time So we are suggesting a model, **BUGCHER** which can learn the program transformations of code which are done by some programmer to fix an issue by producing a patch for a bug. The model takes the line which has the bug as input and produces a fix which may be able to fix the bug with some tweaks.

And another use case of our idea is to make a automotive grading system for programming tasks. It is not a cake walk thing to grade or correct the mistakes in the programming tasks by single individual, the more the number of students the more the time it takes for the tutors to grade the assignments. Moreover the tutor can't correct the mistakes and be there for all the students to help them find the right answer. Thus we are suggesting a way by which we can use our model to help the students. The approach is, at first the tutor feeds our model with possible mistakes the students may make in their assignments and the corresponding fix for it. These sets of mistakes (i.e. bugs) and their fixes can be retrieved from previous assignments and also from the tutor's years of experience.

1. INTRODUCTION

1.1 Why patch generation?

Software systems have entered every corner of our life. We use software in our desktop PC to work, store our photos to cloud system, and plan our dinner with applications in our phone. Software defects are pervasive in software systems and can cause undesirable user experience, denial of service, or even security exploitation. Generating a patch for a defect is a tedious, time-consuming, and often repetitive process. Human developers have to diagnose the defect from the collected bug reports, understand its root cause, craft the patch to correct the defect, and validate the patch with regression tests and code reviews.

As another example, in programming courses student submissions that exhibit the same fault often need similar fixes. For large classes such as massive open online courses (MOOCs), manually providing feedback to different students is an unfeasible burden on the teaching staff.

Automatic patch generation holds out the promise of automatically correcting software defects without the need for human developers to diagnose, understand, and correct these defects. Given a set of test cases, at least one of which exposes a defect in the software, the goal of automatic patch generation system is to produce correct patches to eliminate the defect.

1.2 Why learning from human patches?

As human beings, we wrote a lot of code. The number of open source projects in GitHub just reached 12 million at 2015 and it is still counting. we believe the large amount of code provides not just challenges but opportunities. It enables many

potential data-driven techniques that were not possible before.

Our observation is that when fixing software errors human developers often unconsciously consider only those patches that follow certain software engineering code patterns and that exhibit certain characteristics of successful patches. The goal of our projects is to use a combination of cutting-edge machine learning and program analysis techniques to learn those human knowledges from successful human patches widely available over the web.

1.3 Challenges:

Building an automatic patch generation system is very challenging. The given test cases are always incomplete and there could be many plausible patches that produce correct output for all given test cases but produce incorrect output for some other cases. A successful automatic patch generation system has to rely on additional information other than given test cases to rank correct patches ahead of such plausible but incorrect patches.

The main challenge of example-based learning lies in abstracting concrete code edits into classes of transformations representing these edits. For instance, Fig. 1 shows similar edits performed by different students to fix the same fault in their submissions for a programming assignment. Although the edits share some structure, they involve different expressions and variables. Therefore, a transformation should partially abstract these edits

2. APPROACH

2.1 Deep learning:

Deep learning is a function of artificial intelligence that mostly imitates the workings of the human brain in handling and processing of data and creating patterns for the purpose of decision making. Deep learning is a subset of machine learning which is again the subset of artificial intelligence (AI) that has networks that are capable of learning unsupervised data which is either unlabeled or unstructured. Also known as deep neural network deep or neural learning. Deep learning utilizes a hierarchical level of artificial neural networks to carry out the process of machine learning, a self-adaptive algorithm that gets increasingly better analysis and patterns with experience or with newly added data. The artificial neural networks are loosely built much like the human brain, with nodes similar to biological neurons that are connected together like a web. Traditional programs follow a linear pathway where the data is analysed and built-in such manner whereas deep learning follows a non-linear approach through building a hierarchical function of networks.

2.2 Recurrent Neural Networks:

Recurrent Neural Network (RNN) is a type of Neural Network where the output from previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.

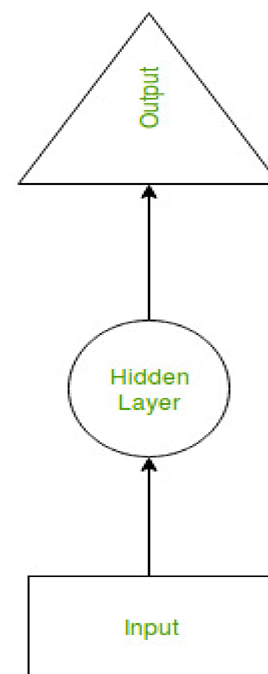

Fig 1: RNN Block Diagram

RNN have a “memory” which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

2.3 Transformer Model:

Recurrent Networks were, until now, one of the best ways to capture the timely dependencies in sequences. However, the transformer paper proved that architecture with only attention-mechanisms without any RNN (Recurrent Neural Networks) can improve on the results in translation task and other tasks. One improvement on Natural Language Tasks is presented by a team introducing BERT: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

The Transformer – a model that uses attention to boost the speed with which these models can be trained. The Transformers outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. It is in fact Google Cloud's recommendation to use The Transformer as a reference model to use their Cloud TPU offering. So let's try to break the model apart and look at how it functions. The Transformer was proposed in the paper Attention is All You Need. A TensorFlow implementation of it is available as a part of the Tensor2Tensor package. Harvard's NLP group created a guide annotating the paper with PyTorch implementation. In this post, we will attempt to oversimplify things a bit and introduce the concepts one by one to hopefully make it easier to understand to people without in-depth knowledge of the subject matter.

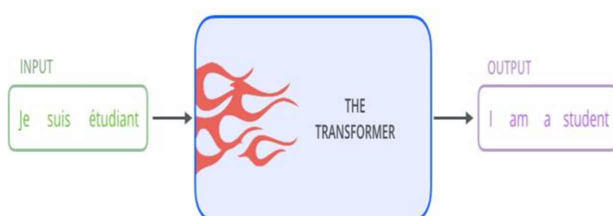

Fig 2: Transformer Model Block Diagram

The Encoder is on the left and the Decoder is on the right. Both Encoder and Decoder are composed of modules that can be stacked on top of each other multiple times, which is described by n times in the Figure 3. The modules consist mainly of Multi-Head Attention and Feed Forward layers. The inputs and outputs (target sentences) are first embedded into an n-dimensional space since we cannot use strings directly.

One slight but important part of the model is the positional encoding of the different words. Since we have no recurrent networks that can remember how sequences are fed into a model, we need to somehow give every word/part in our sequence a relative position since a sequence depends on the order of its elements. These positions are added to the embedded representation (n-dimensional vector) of each word.

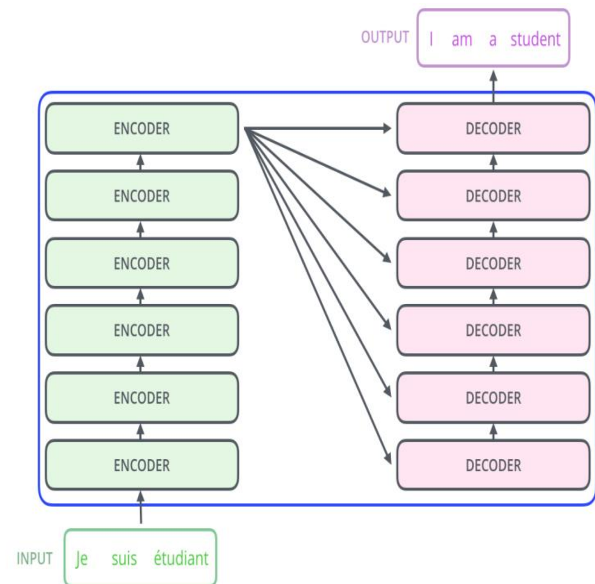

Fig 3: Encoder and Decoder structure inside the Transformer Model

It's not completely possible to mimic a programmer as a program which can be used to fix a bug, even if we train the machines, they are always going to be machines.

2.4 Dataset:

A project's popularity is determined by computing the sum of z-scores of its forks and watchers. We kept only bug commits that contain only single statement changes and ignore stylistic differences such as spaces or empty as well as differences in comments. We also attempted to spot refactoring such as variable, function, and class renaming, function argument renaming or changing the number of arguments in a function. The commits are classified as bug fixes or not by checking if the commit message contains any of a set of predetermined keywords such as bug, fix, fault etc. We evaluated the accuracy of this method on a random sample of 100 commits that contained SStuBs from the smaller version of the dataset and found it to achieve a satisfactory 94% accuracy. This method has also been used before to extract bug datasets (Ray et al., 2015; Tufano et al., 2018) where it achieved an accuracy of 96% and 97.6% respectively.

The bugs are stored in a JSON file (each version of the dataset has each own instance of this file) Any bugs that fit one of 16 patterns are also annotated by which pattern(s) they fit in a separate JSON file each version of the dataset has each own instance of this file).

The dataset contain the following fields:

Field Name	Description
bugType	The bug type(16 possible values)
commitSHA1	The hash of the commit fixing the bug
commitFile	Path of the fixed file
patch	The diff of the change
projectName	The oncatenated repo owner and repo name separated by a '.'
lineNum	The line in which the bug exists
nodeStartChar	The character position at which the affected ASTNode starts
before	The affected AST node text before the fix
after	The affected AST node text after the fix

Dataset Statistics

Pattern Name	Instance (small)	Instance (large)
Change Idenfier Used	3265	22773
Change Numeric Literal	1137	5447
Change Boolean Literal	169	1841
Change Modifier	1852	5010
Wrong Function Name	1486	10179
Same Function More Args	758	5100
Same Function Less Args	179	1588
Same Function Change Caller	187	1504
Same Function Swap Args	127	612
Change Binary Operator	275	2241
Change Unary Operator	170	1016
Change Operand	120	807
Less Specific If	215	2381
More Specific If	175	2381
Missing Throws Exception	68	206
Delete Throws Exception	48	508

3. EXPERIMENTAL RESUTS

In our evaluation, BUGCHER generated patches for 42 out of 60 scenarios. In 35 (58%) scenarios, the generated patch applied the same edits applied by developers, whereas, in the other 26 scenarios, the transformations applied more unwanted edits resulting in a bad path.

We can infer from our Experimental that automatic bug fix is highly possible and achievable with greater accuracy in the near future.

4. CONCLUSION

We presented BUGCHER, a technique for synthesizing program transformations from examples. Given a set of examples consisting of program edits, BUGCHER synthesizes a transformation that is consistent with the examples. Our model builds on the state-of-the-art program transformation mechanism, Transformer Model.We evaluated BUGCHER on two applications: transformations of programs for developers to fix bugs and the transforming the programmatic assignments that describe how students can “fix” them. As future work, we plan to make our model to efficiently learn the program transformations. Although BUGCHER is capable of fixing student bugs automatically, it lacks the domain knowledge of a teacher and can therefore generate functionally correct but stylistically bad fixes. To address this issue the tutor can give feedback on the fix suggested by BUGCHER and a ranking system to use the right fix for the bug are to be implemented.

5. REFERENCES

[1] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, “A graph-based approach to API usage adaptation,” in Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, ser. OOPSLA ’10. New York, USA: ACM, 2010.

[2] C. Goues, S. Forrest, and W. Weimer, “Current challenges in automatic software repair,” Software Quality Journal, vol. 21.

[3] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, page 26. ACM, 2014.

[4] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering, pages 364–374. IEEE Computer Society, 2009.

[5] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In ACM SIGPLAN Notices, volume 47, pages 431–450. ACM, 2012.

[6] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 289–299

[7] D. Edge, S. Gulwani, N. Milic-Frayling, M. Raza, R. Adhitya Saputra, C. Wang, and K. Yatani, "Mixed-initiative approaches to global editing in slideware," in Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. ACM, 2015, pp. 3503–3512.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 770–778, 2016.

[9] R. Robbes and M. Lanza, "Example-based program transformation," in Model Driven Engineering Languages and Systems, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5301.