Abstraction Mechanism on Neural Machine Translation Models for Automated Program Repair

by

Moshi Wei

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

© Moshi Wei 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Bug fixing is a time-consuming task in software development. Automated bug repair tools are created to fix programs with little or no human effort. There are many existing tools based on the generate-and-validate (G&V) approach, which is an automated program repair technique that generates a list of repair candidates then selects the correct candidates as output. Another approach is learning the repair process with machine learning models and then generating the candidates.

One machine learning based approach is the end-to-end approach. This approach passes the input source code directly to the machine learning model and generates the repair candidates in source code as output. There are several challenges in this approach such as the large vocabulary, high rate of out-of-vocabulary (OOV) tokens and difficulties on embedding learning. We propose an abstraction-and-reconstruction technique on top of end-to-end approaches that convert the training source code to templates in order to alleviate the problems in the traditional end-to-end approach. We train the machine learning model with abstracted bug-fix pairs from open source projects. The abstraction process converts the source code to templates before passing it to the model. After the training is complete, we use the trained model to predict the fix templates of new bugs. The output of the model is passed to the reconstruction layer to get the source code patch candidates.

We evaluate our approach by training the machine learning model with 470,085 bug-fix pairs collected from 1000 top python projects from Github. We use the QuixBugs dataset as the test set to evaluate the result. The fix of the bug in the QuixBugs is verified by the test cases provided by the QuixBugs dataset. We choose the traditional end-to-end approach as the baseline and comparing it with the abstraction model. The accuracy of generating correct bug fixes increase from 25% to 57.5% while the training time reduces from 5.7 hours to 1.63 hours. The overhead introduced by the reconstruction model is 218 milliseconds on average or 23.32%, which is negligible comparing to the time saved in the training, which is 4.07 hours or 71.4%. We performed a deep analysis of the result and identified three reasons that may explain why the abstraction model outperforms the baseline. Comparing to existing works, our approach has the complete reconstruction process which converts templates to the source code. It shows that adding a layer of abstractions increases the accuracy and reduces the training time of machine-learning-based automated bug repair tool.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Professor Lin Tan for the continuous support of my master study and research, for his patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my master study.

Besides my advisor, I would like to thank the rest of my thesis committee for their insightful comments.

Dedication

I dedicate this to my mother, father and my fiancee.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

2 Background and Related Work 4

2.1 Terminology 4

2.2 Generate and Validate Program Repair 5

2.3 Automated Program Repair with Deep Learning 6

3 Approach 8

3.1 Data Extraction 9

3.2 Abstraction 11

3.3 Neural Network Model 14

3.4 Template Generation 16

3.5 Reconstruction 18

3.6 Validation 20

4 Experimental setup 21

4.1 Dataset 21

4.2 Implementation 21

4.3 The Traditional Model Structure 22
4.4 Training 22
4.5 Hyperparameter Search 23

5 Experimental Results **24**
5.1 Comparison of Number of Correct Fixes Generated by Different Models . . 24
5.2 Distribution of Bug Types 26
5.3 The Execution Time of Reconstruction 29

6 Result Analysis **30**
6.1 The Clustering of Candidates 30
6.2 The Tracking of Variables 32
6.3 The Decrease of Out-of-Vocabulary Rate 34
6.4 The Unique Fix From Baseline 35
6.5 Semantically Equivalent Fix 36

7 Threats **37**
7.1 Multi-line Bugs 37
7.2 Random Hyperparameter Search 37

8 Conclusion **38**

9 Future Work **40**
9.1 Multi-line templates 40
9.2 Alternative Abstractions 40

References **41**

APPENDICES **50**

A Model configuration **51**
A.1 traditional model configuration details 51
A.2 abstraction model configuration details 53

List of Tables

3.1 table of reserved tokens 12

3.2 Example of template outputs 17

5.1 Comparison of the number of correct fixes 25

5.2 Comparison of ranks of shared correct fixes 25

5.3 Types of bugs in QuixBugs dataset 26

5.4 Comparison of type of bugs repaired 27

5.5 Comparison of time cost between models 29

6.1 Top-10 results of the bug in Quixbugs dataset with file name breadth_first_search, bug ID 1 31

6.2 Top-10 results of the bug in Quixbugs dataset with file name levenshtein, bug ID 16 32

6.3 Top-10 filtered results of the bug in file reversed_linked_list in Quixbugs dataset with bug ID 28 33

6.4 Top-10 results of the bug in Quixbugs dataset with file name reversed_linked_list, bug ID 28 34

6.5 Top-10 results of the bug in Quixbugs dataset with file name Subsequence 35

List of Figures

3.1 Workflow of Automated program Repair with Abstraction Technique . . . 8
3.2 Example of Parsing Git diff API Output 10
3.3 Abstraction process with example from training data 11
3.4 Convectional sequential learning model structure 15
3.5 Beam Search Ranking Example 18
3.6 Workflow of Reconstruction Algorithm 19

5.1 Comparison of type of bugs 28

Chapter 1

Introduction

As software scale grows in recent years, fixing software bugs becomes increasingly expensive for developers. The major part of the cost comes from human-hours in manual debugging. Developers have to spend a large amount of time on manually fixing software bugs[2] while this amount of time could have used to develop new features. Automated program repair tools are invented to address this problem. The goal of automated program repair tools is generating program repair patches with less human costs. There are many existing automated program repair tools with different structures and approaches [11, 19, 28, 37, 41, 46, 47, 52, 53, 55, 61, 64, 74, 84, 85].

The classic approach to automated program repair is generate-and-validate (G&V) method [19, 37, 41, 46, 61, 64, 84, 85]. The idea of this technique is that the G&V tools generate lists of candidate patches and validate each of the candidate patches using test cases. Although there are still challenges such as over-fitting patches in the G&V approach, the G&V technique helps developers fixing bugs easier and faster in some scenarios. Recently, there are researches on applying deep-learning techniques to automated program repair [76]. Tufano et al. perform an empirical study on automated program repair with neural machine translation (NMT) models [75]. Sequence-R [11] is an end-to-end machine learning-based program repairing tool. It fixes java bugs by learning the repair operation from the buggy code, the patches and the context using a machine learning model.

One of the challenges in applying end-to-end neural machine translation approaches to automated program repair is the large vocabulary size of programming language. NMT model requires an embedding dictionary that represents all possible tokens of a language. Unlike natural languages, programming languages contain tokens that are defined by users. One programmer can name a variable with any name they wish. This leads to the sparsity

of the programming language and also the large size of the vocabulary. Also, the vocabulary only covers the tokens in the training dataset and the actual vocabulary size is infinite due to the freedom in naming. These missing tokens are called the out-of-vocabulary tokens (OOV tokens) [5]. The OOV tokens have to be replaced by a place holder which makes the information of input sentence incomplete. The model may not understand the input sentence and produce meaningful output because of the missing information. Furthermore, learning the word embedding of tokens for NMT model is very challenging. The representations of the embedding of tokens are high dimensional vectors. Learning an ideal embedding requires a vocabulary with balanced characteristics. In the vocabulary of the traditional model, the majority of the tokens are user-defined variables which have similar characteristics. The tokens with other characteristics such as operators and symbols are rare comparing to user-defined variables. This imbalance in the distribution contributes to the difficulties in the learning of word embedding and leads to inaccurate patches generation.

To alleviates those challenges and improve NMT-based program repair patch generation, we proposed an abstraction technique in data pre-processing that replace the variables in the input to a general form. Abstraction aims to convert the buggy code to a template. The variable name strings in the input are replaced to the placeholder tokens. The model receives a buggy template as input and produces fixing template as output. We also proposed a reconstruction step to generate a list of patch candidates based on the fixing template produced in the previous step.

To evaluate our approach, we collected a dataset with over half a million lines of bug-fix pairs from GitHub [14]. Then, we apply the abstraction procedure to the bug-fix pairs to get the bug-fix template pairs. Next, we trained an NMT model with this dataset to learn the fix operation. After the model is trained, we use it to predict templates for new inputs. A reconstruction procedure is applied to the templates for transforming the templates to source code patches. We test the abstraction model and the traditional model with QuixBugs dataset (40 bugs) and compare the result [86]. The result shows that 60 percent of the bugs in QuixBugs dataset can be fixed by the abstraction technique while the traditional model can only fix 25 percent of the bugs. The fixes from abstraction model not only cover most of the fixes from the traditional model but also covers new types of bugs compared to the traditional model.

The contributions of this thesis are listed below:

- An abstraction-reconstruction technique on the traditional end-to-end automated program repair workflow that transfers source code to templates. The abstraction

process reduces the vocabulary size to 0.1% of the original size and reduce the training time of the machine learning model to 28% of the original training time. The reconstruction technique on the traditional end-to-end automated program repair workflow that transfers repair templates to source code candidates with an average time cost of 218 milliseconds. We are the first to propose a complete abstraction-reconstruction technique in NMT-based program repair.

- An empirical study on the accuracy, time consumption, and ranking of QuixBugs patch candidates generated by the abstraction model and the traditional model which identifyies the three reasons why the abstraction model outperformed the traditional model.

- A new machine learning model called the abstraction model that takes bug templates as input and generate repair templates as output. The evaluation on QuixBugs shows that we could fix 130% more bugs than traditional NMT-based approach.

Chapter 2

Background and Related Work

In this chapter, we first introduce the terminologies used in this thesis, then we show the background of auto-bug repairs.

2.1 Terminology

Neural Network: A neural network is an algorithm built by layers of nodes with non-linear activation functions [8, 62]. Neural networks are used for learning high-dimensional patterns from given datasets and make predictions based on the correlations it learns from the dataset. Neural networks layers process the inputs and pass the results to the next layer. The coefficients in each node, also called weights [33], are adjusted backward according to the difference between predictions and answers. The weights are adjusted iteratively using the training algorithm such as gradient descent [9] according to the input data. This process is commonly referred to as the training process. The training process stops when the result stops improving after many iterations.

Layer: A layer in a neural network is a group of nodes in the middle of a neural network [26]. The connection of nodes in a layer can be different depending on the task of the model. Complex deep learning models consists of stacks of layers[29].

Recurrent Neural Network(RNN): Recurrent network is a class of deep learning models that enables the possibility of processing and generating sequential data [66, 65]. The simple RNN model suffers from problems such as fixed sequence length and gradient vanishing problem [36]. Improvements such as recurrent net [70] and Long short-term memory unit [32] alleviate the above problems. RNN is strong in sequential data summary tasks such

as voice recognition, paragraph classification but not as good in grammar correction and translation tasks comparing to the Transformer model [77].

Convolutional Layers: The nodes in a convolutional layer are connected to the local regions of the previous layer. A filter with a group of such nodes scans through the input volume to generate the output. A convolutional layer has several filters used to process the input independently [39]. The weight of one filter is shared in a scanning process and independent between each scan. The model we use in this work applies convolutional layers in the learning of the vector embeddings. Using convolutional layer improves the learning of local changes in sequential data [77].

Neural Machine Translation: Neural machine translation is a machine learning-based approach to language translation tasks [3]. The generative model is commonly used in NMT problems. We develop our approach to address the automated program repair problem from the NMT models used for NMT tasks due to the similarity between these two tasks.

Attention Mechanism: Attention mechanism proposed by Bahdanau et al. [3] introduces a weight factor α to the input sequence that prioritize the information relevant to the next output token. The weight factor learns the relevance between each element of the input sequence and output sequence. It is adapted as a standard component of NMT models. The weight factor learns the relevance between each element of the input sequence and output sequence. The attention mechanism emphasizes the correlation between the input layer and the output layer thus alleviates the bias of long-distance dependency problem in recurrent model [69].

2.2 Generate and Validate Program Repair

GenProg [41] is an automated C program repair tool using genetic programming techniques. It generates many repair candidates by mutating the input with a group of mutation operators. These mutation operators are written by human programmers with domain-specific knowledge. The selecting of the patch in GenProg is through measuring which patch makes the most passes of test cases. The GenProg also collected 69 defects in popular C programs as the benchmark dataset to test with. This dataset is accepted by other later works in automated program repair research.

RSRepair [61] is an automated program repair tool constructed by modifying GenProg. It applies random search instead of genetic programming to generate patches but uses the same mutants GenProg uses such as deletion and copy. RSRepair generates random

patches without fitness guidance. It also checks if the patch is valid before running the patch through test cases to save execution efforts on invalid candidates. The result shows RsRepair repairs more bugs and requires less time than GenProg.

Kim et al. [37] proposed several fix templates to address the randomness problem in GenProg. This work concludes several common fix operation templates from the manual investigation of six thousand developer patches in real bugs.

SPR [46] uses condition synthesis to generate repair patches. It replaces the buggy line to one of the hard-coded templates with a wild-card variable. Then the SPR test the wild-card variable with a set of values and record the output. Once SPR finds a variable in the context of the bug that simulates the bechavior of the fix. SPR apply the variable to the buggy line to generate the fix.

SemFix [55] uses symbolic execution techniques to address automated program repair problem. In the first step, the SemFix finds the bug location by using bug localization algorithm. Then, it synthesis the correct patch by solving the repair constraints. Angelix [53] also uses symbolic execution technique to fix bugs. Compare to SemFix, it is capable of fixing bugs in multiple locations which enables automated program repair on a larger scale. Direct-Fix [52] focus on generating simple fixes while considering bug localization and patch generation at the same time.

Nopol [84] focuses on repairing bugs related to the if-condition statement. It records the buggy condition statement and the expected output in the process of running test cases. Then, it solves the mapping from the expected output to the correct condition statement by using a satisfiability solver.

Prophet [47] extends SPR with a ranking system that builds on the debugging knowledges from developer patches. Prophet reuses the same bug localization and mutation algorithm from SPR. It adds a ranking layer on top of the testing phase to prioritize the patches that have higher chances fo repair the bug. The result shows that the Prophet can find repair patch faster compare to SPR.

2.3 Automated Program Repair with Deep Learning

DeepFix: [28] is a machine learning-based automated program repair tool focus on C program. Gupta et al. adapt the recurrent neural network (RNN) with attention model (Bahdanau, Cho, and Bengio 2014) [3] as their model to predict fixes in token level. Comparing to our model, Deepfix model use multilayer GRU (gated recurrent unit) as the

recurrent unit for the encoder and decoder while we use multi-stack convolutional layer for the encoder and decoder. The model uses Bahdanau attention [3] with the top-5 beam search result to produce candidate patches while we use multi-headed attention [77] with the top-220 beam search result for candidate patch generation. It replaces the value of variable names to general tokens based on its types. For example, it replaces all string variables to STR and all integers to NUM [28]. Comparing to our work, the Deepfix can not map the abstracted token back to the original value. We use a different abstraction strategy which enables the tracking the variable value and restores the abstracted tokens to its original value. Because of the reconstruction, our model has a better acceptance of a wide variety of inputs.

Sequence-R [11] is another work on end-to-end machine learning-based program repairing. The tool focuses on predicting java bug fixes with parallel data input. The model takes the original code, the abstracted buggy context and the buggy template as input. Comparing to our work, their abstraction strategy cannot track the value of variables so that they can not always fully recover the source code for the given input.

Tufano et al. [75] performed an empirical study on automated program repair using machine learning technique. They use LSTM (long-short-term-memory) multilayer encoder and decoder model with Badanau attention and beam search in their work. They also applied an abstraction layer on top of the model. Comparing to our model, they uses an abstraction strategy similar to us but without reconstruction. They can only generate fix templates instead of complete source code. They assume that for each input and output, the vocabulary of the input is a superset of the vocabulary of the output [75]. This means that they assume the output will not have tokens that are not in the input. This assumption makes the model unable to generate correct fixes for many real-world bugs since this assumption does not hold when it comes to fixing real bugs. They did not consider the reconstruction and the use of buggy context because of the previous assumption.

Chapter 3

Approach

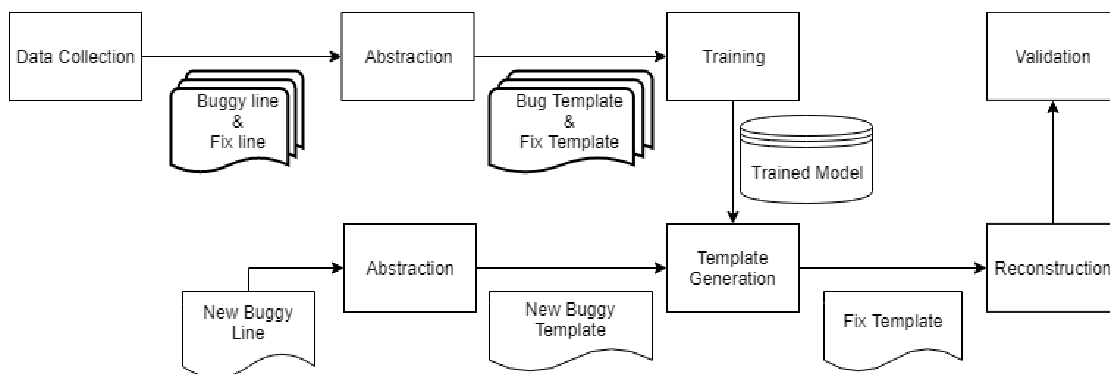

Figure 3.1: Workflow of Automated program Repair with Abstraction Technique

The approach has two parts which are training and template generation. The training takes the dataset which includes a list of bug-fix source code pairs as input and produces a trained model as the output. The template generation takes a single line of unseen (i.e., new to the trained model) buggy source code as input and produces the correct patch as the output.

The training begins with data extraction. We collect bug-fix pairs from open-source projects on GitHub. We choose GitHub as the source of data because it is one of the largest open-source community [14]. In the abstraction step, we transfer these source code pairs to bug template and fix template pairs. We use these datasets as the training dataset for the training of the model. Next, we generate templates for new bugs using the trained model. The model generates a list of fix templates after receiving a new bug template from

the abstraction step as input. The fix templates are converted to source code patches in the reconstruction step. In the end, we test the patches and select the correct patches in the validation step. Figure 3.1 shows the workflow of the training and template generation process. The arrows indicate the direction of the workflow. The texts in the boxes are corresponding to the major processes of the approach. The pages represent the result after each step.

Section 3.1 describes the data collection process. Section 3.2 shows the detail of the abstraction process. Section 3.3 explains how the neural machine translation model works. Section 3.4 shows the template generation output with an example. Section 3.5 describes the reconstruction workflow. Section 3.6 explains the validation method we use for generating correct patches.

3.1 Data Extraction

-+ We collect historical commit data from GitHub, which is one of the biggest open-source developer communities [31]. Among the popular programming languages such as Java, Python, C, and C++ for automated program repair research, we choose Python to study for the reason that Python has simpler syntax comparing to other programming languages such as Java and C++ [59]. We use single line bug-fix pairs as the training dataset for our machine learning-based automated program repair technique because single line bug-fix pairs are more likely to be bug fixes than multi-line changes [7].

In the first step, we select the top one thousand most popular python projects at 2019 Jan 7th as the source projects of the data. The popularity of a project is measured by the number of stars of the project. Next, we download the projects and use git API to get one line bug-fix pairs from the historical commits of these projects. We collected a total of 519,452 bug-fix pairs from these one thousand projects. For each project, we search through the entire commit history of the project and select commits with certain keywords. We manually investigate 100 commit messages and summarize a list of common keywords including “fix”, “bug” and “patch” as a template to get commits that related to program repair because these keywords commonly appear in the program repair commit messages [49]. From the python projects we selected, we apply the above template to filter unrelated commits and keep the buggy commits and its repair commits. We extract multiple bug-fix pairs of a commit by comparing the changes of the buggy version and fixed version.

A commit is selected into the dataset if one or more matches of keywords are found in the commit message. Also, on top of the result of this step, we remove instances

```
- field_names = map(str, field_names)
+ field_names = [str(x) for x in field_names]
  if rename:
      seen = set()
      for index, name in enumerate(field_names):
```

Figure 3.2: Example of Parsing Git diff API Output

from the result by going through the commit message again with a new set of keywords. The keywords used to filter commit contains "rename", "rewrite", "clean up", "refactor", "merge", "misspelling", "compiler warning" and "comment". These keywords are used for discarding false-positive commits from the result (i.e., commits with repair keywords but the content is not related to program repair) [49]. After we obtained the list of commits, We select one-line change pairs by using the git diff API, which is a tool to compare and highlight the difference between two versions of a file. The one-line change pairs are extracted by parsing the output of git diff API using a parser.

Figure 3.2 shows an example of bug-fix pairs. Firstly, the parser splits the output to multiple hunks by file names. Then, for each chunk, the parser scan through the code chunk and search for a pair of the removed line (start with "-" character) and added line (start with "+" character). We remove the "-" and "+" in the lines and attach each bug line and fix line pairs to a list. In the example below, the removed line is indicated by red color and the added line is indicated by green color. The data we get after this step is a list of bug and fix line pairs (bug-fix pairs). To remove data that unrelated to repair from the dataset, we discard the bug-fix pair if the pair contains comments or strings.

We apply two constraints for data cleaning. The first constraint is filtering out comments from the bug-fix lines. We discard the sample if the sample is editing of comments, such as fixing a comment typo or changing the function description. The second constraint is removing samples that contain string value (e.g. a =" some string"). Since the focus of the model is on repairing source code bugs, predicting the variation of the string value is less interesting. Also, including string values will increase the volume of the vocabulary of input dramatically and make the template generation a harder task. For each bug-fix pair from the previous step, if the line matches one or more of the filters, the pair is discarded. To check if the data collected by this approach is a good representation of bug-fix data, we picked 100 random samples from the bug-fix pairs and manually inspect each of them. 93 out of 100 samples are related to repair operation [49].

3.2 Abstraction

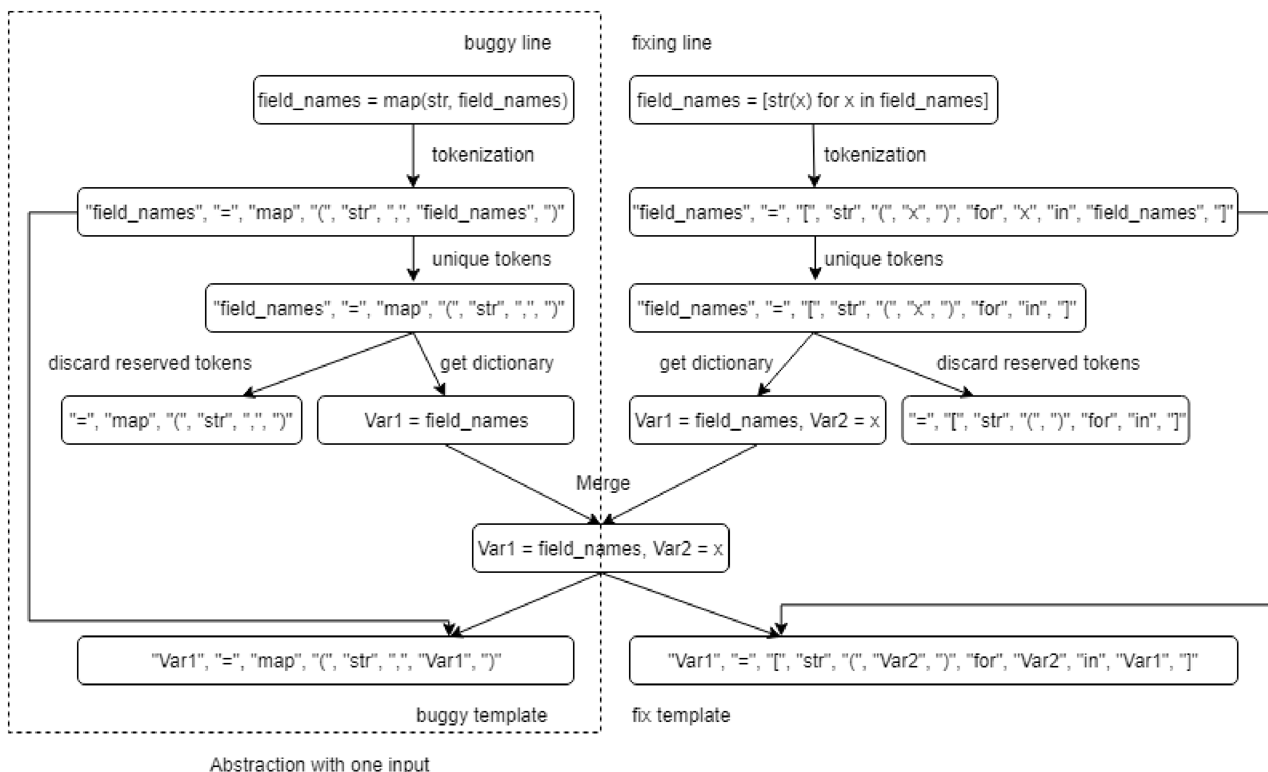

Figure 3.3: Abstraction process with example from training data

We propose an abstraction technique that converts the source code to template to alleviate the OOV token problem in traditional NMT model for program repairing mentioned in Chapter 1. Figure 3.3 describes the process of abstraction. In this graph, we use an example to demonstrate the abstraction process. The example we use is a buggy commit from python project boltons[1]. We used the python built-in tokenizer and an abstraction script to transfer tokens in the input sequence to get the bug template and fix template. We keep the duplicates after this step for two reasons. the first one is that duplicates of codes are common in software engineering. The second one is that different bug-fix pairs may have to same repair pattern after the details of the code is replaced in the abstraction step. Keeping the duplicates of bug-fix templates may emphasize the pattern of repairing operation.

[1] File path: root/boltons/namedutils.py; commit hash: bdfba346113b6d31a169e70c05449813d2e-06750

The abstraction starts from a bug-fix pair from the dataset we obtain in data collection. In the first step, we tokenize the bug-fix pair using built-in python parser. After the tokenization completes, we record the unique tokens from the bug-fix pair to a list. Then, we further remove tokens from the list if the token is in the reserved token list. The reserved keywords list contains tokens that related to python language syntaxes such as operators, data types, built-in methods, and common attributes. Table 3.2 shows all the reserved tokens used in the approach.

After the above two steps of removal operations, the remaining tokens are mostly relevant to the context of the bug such as variable names and methods names. We need to remove these tokens from the input and replace these tokens to placeholders. The rationale behind this is that the original token before abstraction provides details that are related to the context of the bug. Replacing these tokens with variable tokens (VTs)

,	(	)	:	.	=
*	]	[	-	<	>
+	!	/	{	}	%
\	|	;	&	∧	”
’	False	None	True	and	as
assert	break	class	continue	def	del
elif	else	except	finally	for	from
global	if	import	in	is	lambda
non	local	not	or	pass	raise
return	try	while	with	yield	self
abs	min	dict	help	setattr	all
dir	hex	next	slice	any	divmod
id	object	sorted	ascii	enumerate	input
oct	staticmethod	bin	eval	int	open
str	bool	exec	isinstance	ord	sum
bytearray	filter	issubclass	pow	super	bytes
float	iter	print	tuple	callable	format
len	property	type	chr	frozenset	list
range	vars	classmethod	getattr	locals	repr
zip	compile	globals	map	reversed	import
complex	hasattr	max	round	delattr	hash
memoryview	set				

Table 3.1: table of reserved tokens

removes the context dependency of the input. With these tokens replaced to their generic form, the model focus more on predicting the programming language syntax rather than the variable names. The purpose of the abstraction technique is to remove details that are unnecessary for program repairing based on the belief that the fixing operation of the buggy line is on syntax instead of the details such as the name of variables. In other words, the modifications which can potentially fix the bug are on the syntax of the code rather than the naming of variables. Motivated by existing study [80], changing variable names do not affect the syntax of the code and the execution of the code. Also, the naming of variables and methods names are project-specific, which means that these variable names do not represent the syntax of the programming language. The abstraction technique will replace these tokens to its generic form so that the input of the model represents more of the syntax of the bug rather than the naming methodology of the project that has the bug.

Next, we build a dictionary from the remaining tokens of the input line. We use VTs ("Var" followed by a number) as the keys of the dictionary where hashtags represent the indices of the tokens in the dictionary. In some cases, the VTs in output have lower indices than VTs in the input. In this case, we reorder the dictionary to make sure all input tokens are before the output tokens. For example, given a dictionary {Var1 : input_func_name_one Var2 : output_string_one, Var3 : input_func_name_two}. Var1 and Var3 have a gap, which is Var2. We believe this gap between indices is a hint to the answer that can potentially interference the template generation process of the model. We remove this gap to make sure of the isolation of the answer to the template generation process. To do so, we sort the input token and output token independently and concatenate the output dictionary after input dictionary. The reason for sorting two dictionaries separately is to make sure that the abstraction of the input line is consistent between the training phase and the template generation phase. If the two dictionaries are not sorted separately, there could be a mismatch of indices between tokens from the output and tokens from the input. This mismatch will make the abstraction of only the input tokens inconsistent. For example, If the dictionary of the abstraction of the input line and the output line is dict1 = {Var1 = a, node = b, Var3 = c, Var4 = d} where token c belongs to output only. The abstraction of input line by itself will be dict2 = {Var1 = a, Var2 = b, Var3 = d}. This creates the inconsistency between {Var3 = c , Var4 = d} in dict1 and {Var3 = d} in dict2. By using the above method, we keep track of each key-value pairs so that the variable dictionaries are consistent in the abstraction stage and the reconstruction stage.

The abstraction process in the template generation for new inputs is slightly different from the abstraction in training. In the process of generating training data, we have the input buggy line and the fixed-line. Meaning that the correct fix of the buggy line is known. However, in the template generation for new inputs, the dictionary from the input side does

not merge to the output side because the fixing line is missing. Because the correct fix of a new buggy input is unknown, we need to tokenize the buggy line without the information from the fixing line. In this project we only consider single line bugs because the input length is short and the fix is concise. For multi-line bugs, we could transform the buggy lines to a long one-line input sequence and use this as the input of the model.

The abstraction in the template generation process begins with a buggy line. The buggy line is tokenized and filtered by using the same algorithm from the abstraction in the training phase. The missing of the fix line dictionary does not change the ranking of VTs because the two dictionaries are sorted in isolation. At the end of the abstraction process, the new buggy lines transfer to the buggy templates.

3.3 Neural Network Model

The model we used to learn the bug-to-fix operation is the convolutional sequence to sequence model [22]. The initial purpose of this model is to apply machine learning technique on natural language translation tasks. We choose this model because the model performs better on end-to-end program repairing tasks comparing to other models such as LSTM-based sequence to sequence [49]. In model selection, we evaluate the performance of the model by comparing the validation accuracy across models. This model preforms well on a variety of natural language processing tasks as well [23]. The author show how the convolutional layers contributes to the improvement of the model in the paper [23]. The model learns the translation operation by adjusting its weight and minimize the loss function according to a pair of given input and output, which are two sentences in two languages with the same meaning. We believe that the model is capable of learning the repairing operation from a piece of buggy source code to a piece of correct code because the translation and repairing are both transformations between input and output. Similar to the learning of natural language translation, in our approach, we use this model to learn the transformation from the buggy lines to the fix lines.

The structure of the convolutional sequence to sequence model contains three major parts, which are the encoder, decoder, and multi-step attention [22]. The encoder is a neural network structure that contains many layers of one-dimensional convolution blocks. Each block $\mathbf{z}_l$ takes the output of the previous block $\mathbf{z}_{l-1} = (z_{l-11},...z_{l-1m})$ as input and output $\mathbf{z}_l = (z_{l1},...\ z_{lm})$ except the first layer of encoder block. Each input block takes the input sequence $\mathbf{x} = (x_1,...\ x_m)$ as input. The decoder structure is similar to the encoder. It takes the fixing part of the input sequence and outputs a hidden state $\mathbf{h}_l = (h_{l1}\ ,...\ h_{ln})$. The last part of the model structure is multi-step attention. The traditional attention

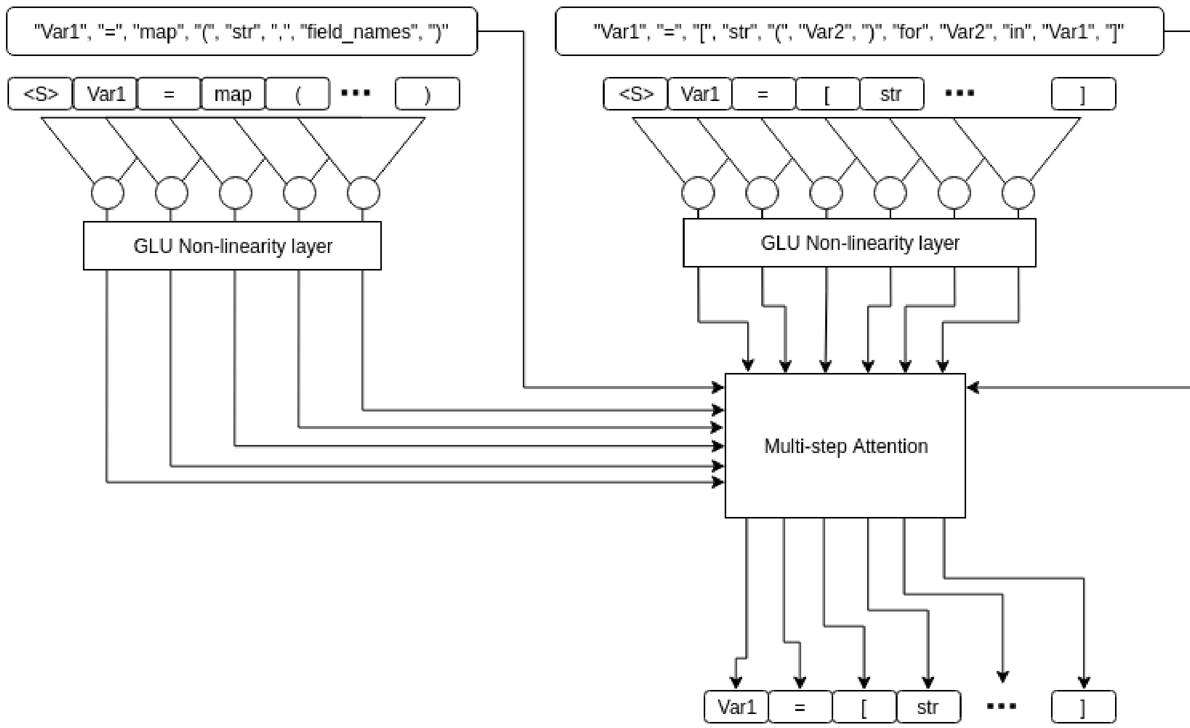

Figure 3.4: Convectional sequential learning model structure

layer is simply a dot product of the last input hidden states and last output hidden states. However, in this model, the initial input representation and output representation are also involved in the calculation of the attention. On the encoder side, the representation of the input of each encoder convolutional layer is added to the calculation of decoder hidden states as a residual connection. On the decoder side, the output representation of each decoder layer is added to the output of the attention.

As soon as the data is abstracted and tokenized, we split the data into two parts, which are training data and validation data, to train the model. We use 10-fold cross-validation in this step. It is common to split training and validation datasets in this way [38]. The training of the model is minimizing the error between output and label by adjusting its weights with multiple iterations (also known as epochs) of training data. The initial parameter is weight W$\in R^{2dx2d}$, bias $b_w \in R^{2d}$ and input x $\in R^{k*d}$. After several epochs of training, the model contains a certain set of weight parameters which generates the most promising repair templates. After the model is trained, The training part of the workflow is finished.

3.4 Template Generation

Once the model is trained, we use the trained model to generate repair patches for new bug instances. This process is called the template generation. The template generation uses the trained model to obtain the fix of a new buggy input (i.e., a bug the trained model has never seen before).

To begin with, the template generation process requires one buggy line, the context of the buggy line and the trained model. Firstly, we abstract the buggy line to the bug template using the abstraction script. Note that the abstraction in predicting phase is slightly different than the abstraction in the training phase. In the abstraction of training data, we transfer both the input and output to templates. In the abstraction of the predicting phase, we only transfer the input because the output (repaired line) is unknown.

Next, We feed the bug template as input to the trained model to predict the patch template. In this step, the output of the model is a set of templates instead of source code. After the model finishes reading the input sequence, the generation begins with the initial token <START> and then generates tokens one at a time and concatenates tokens to the output sequence. In each step of the token generation, the model outputs a new token based on the token sequence generated in previous steps and concatenates the new token to the output sequence. In one iteration, the model uses the beam search

rank	fix template
0	return 1 + Var4(Var1[1:], Var2[1:])
1	return 1 + Var4(Var1[1:], Var2[1:])
2	return 1 + Var4(Var1[1:], Var2[1:]]
3	return 1 + Var4(Var1[1:], Var2[1:]])
4	return True + Var4(Var1[1:], Var2[1:])
5	return 0 + Var4(Var1[1:], Var2[1:])
6	return 1 + Var4(Var1[1:], Var2[1:]
7	return 1 + Var1(Var2[0:], Var3[1:])
8	return 1 + Var1(Var2[1:], Var3[1])
9	return 1 + Var1(Var2[1:], Var3[1:]]
10	return Var4(Var1[1:], Var2[1:])

Table 3.2: Example of template outputs

algorithm to decide which token to select as the output token. In the next iteration, the model takes an incomplete sequence from previous steps as the input and produces the next token based on the current incomplete input sentence. After several iterations, the model finally outputs an <END> token and that ends the template generation process. The beam search algorithm is a searching algorithm that generates output candidates by searching through the token candidates' space and picking the best candidates according to the probability based on the current incomplete input. The beam search algorithm is an alternative to the default greedy search algorithm. The greedy algorithm returns the token with the most probability at each generation step. On the other hand, the beam search algorithm returns top K result for each generation step and ranks the complete output by the total likelihood score in T generation steps. The algorithm has two parameters, beam width and searches depth which corresponds to the above variable K and T respectively.

Table 3.2 shows the ranking of the beam search result. We use the result of bug Levenshtein from QuixBugs (bug_ID 16) as an example. In this bug from QuixBugs dataset, the buggy code is "return 1 + levenshtein(source[1:], target[1:])" and the fix code is "return levenshtein(source[1:], target[1:])" In this example, the rank of the correct fix template is ten. The number beside the token is the negative log of the probability of the token. The ranking of this example is measured under the beam width 220 and search depth 220. The beam width is the number of candidate tokens considered in each iteration of predicting the next token. we set the beam size and the number of best candidates to 220 for both models. We choose 220 because the vocabulary size is 220 tokens and the beam width can not go over this limit. The rank is calculated by the sum of the likelihood

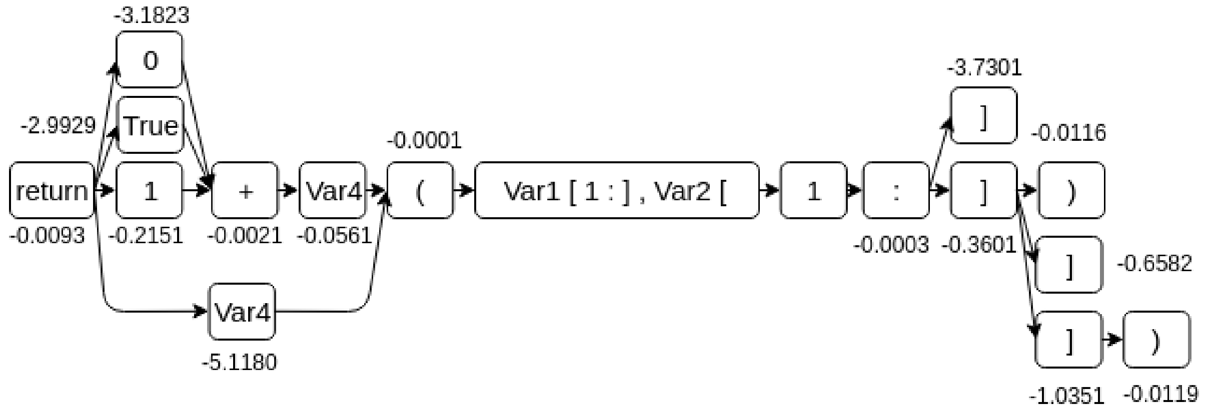

Figure 3.5: Beam Search Ranking Example

score for each sequence and sorted in descending order(i.e., from high to low). Figure 3.5 shows an example of the beam search scores, in the second iteration, the score of token “0” is lower than the score of token “True”. In the current sequence (from the beginning to the second token), the score of sequence with “True” is lower than the score of sequence with“0”. In the generation of the second last token, token “)” (-0.0116) has a higher score than token“]” plus token“)” (-1.0351+0.0119). The generation process stops at the second last token because of that. At the end of the generation, the output of the beam search algorithm is a ranked list of top 220 fixing template candidate.

3.5 Reconstruction

After obtaining the fix templates from the previous step, we run the reconstruction algorithm for each fix templates to get the candidate patches. The goal of the reconstruction process is to transfer each fix templates to a set of source code patch candidates. To begin with, this process requires a template of the buggy input, the context of the bug and also the fix template candidates from the model. Figure 3.6 shows the workflow of the reconstruction process. In the first step, we take one fix template candidate and replace the VTs back to its original value according to the input dictionary generated in the abstraction process.

After replacing VTs to its original values, there are two scenarios to consider. The first one is that all VTs in the source code are mapped to its value and the second one is that there are still VTs remains unconverted in the output. For example, if the patch template

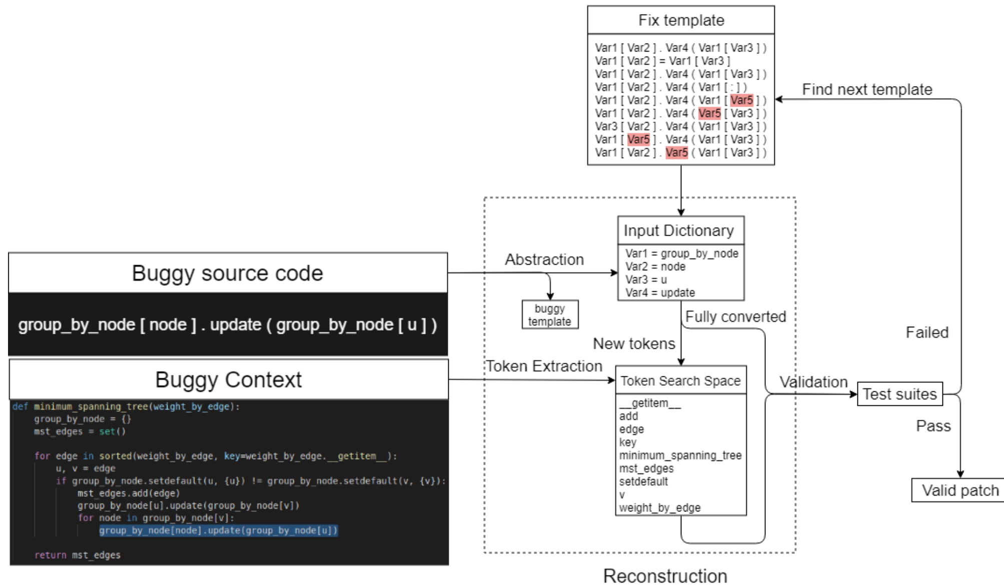

Figure 3.6: Workflow of Reconstruction Algorithm

contains Var1 to Var5 and the input dictionary only contains Var1 to Var4, token Var5 is not in the input dictionary and therefore cannot be converted. If the source code is fully converted, that line of source code is considered as fix line and ready to proceed to the next step. On the other hand, if there are tokens that are not in the input dictionary, we need to map each remaining VTs to a value. because the code is modified in the fixed version. Normally, some tokens in the fixed version are not in the buggy version. This causes the output templates of abstraction model sometimes contain VTs that do not appear in the input. We further investigated the context of the bug and we found that most of the missing tokens are in the context of the bug. We randomly select 10 thousand samples from our database. Out of 7405 parsable instances, 94.2% or 6975 instances have every new token in the context with a range of 5 lines of codes above and below the buggy line. Based on our experiment, the new tokens are very likely to be in the context of the bug. In this case, we create a search space from the context of the inputs and fill in the new token by trying every possible user-defined tokens from the search space. Since the context size of samples in QuixBugs is small, we use the full file as the scope of the context. For new inputs, the size of context is five lines before and after the buggy line. In our approach, we generate a search space of candidate tokens from the context of the bug. Then we generate a fix line candidate for each token or each combination of tokens in search space. In the case we can not find any candidates that pass the test suite successfully, we conclude that

we are not able to generate correct fix for this particular input.

3.6 Validation

The next step is the validation of patches. We take the list of fix line candidates and overwrite the buggy line with each fix line candidates to generate patch candidates. In the end, we run the test cases on each of the patch candidates to check if the patch is correct. In automated program repair, using future data to predict past data should be avoided in the generation of patches [71]. We address this problem by picking an independent test set instead of taking a slice from the training data to make sure that our validation process does not suffer from this problem.

We pass the buggy line as the input to the model and receive a list of patches as outputs. Then, we run related test cases for each output in the list and mark one patch as a correct patch if the patch passes the test suite. Next, we manually investigate the correct patches to check if the correct patch is overfitted [68]. For evaluation purpose, the overhead of the reconstruction is measured by the additional testing time introduced by the reconstruction. Note that the testing time is case-specific, meaning that the run time of testing depends on the complexity of test cases. For this reason, we calculate the run time for each patch and compare the average of run-time per patch.

Chapter 4

Experimental setup

4.1 Dataset

We collected 519,452 bug-fix pairs from top 1000 popular python GitHub projects at 2019 Jan 7th. After applying the filtering process described in section 3.1, there are 423,075 bug-fix pairs remains for training and 470,10 bug-fix pairs for validation. The training samples provide to the two models are the same. The difference between these two models is that the abstraction model has an abstraction layer in pre-processing. The abstraction model takes the abstracted code template as input and the traditional model trains directly on the source code.

We test both models with the QuixBugs [43] dataset which is an independent dataset. The QuixBugs dataset does not overlaps with the training dataset. Thus, the experiment does not suffer from the data contamination problem [71] described in section 3.6.

4.2 Implementation

The tool is implemented using python. The python packages used are pytorch, collections, difflib, io, matplotlib, tokenize, copy, itertools, json, numpy, os, pandas, pprint, random, re, statistics, sbuprocess, sys, time, and types. The model we used is the fconv model from Facebook AI research lab which is an NMT model based on pytorch framework[57].

4.3 The Traditional Model Structure

We implemented the traditional version of the model as a baseline to evaluate the effectiveness of the abstraction model. This section explains how the traditional model is constructed. The traditional model uses the same machine learning model structure but without the abstraction step in the prepossessing and thus do not require reconstruction. We feed the buggy source code line directly to the model without mapping it to a template. The model is trained with source code instead of the template. The output of the model is source code as well.

We do not have the reconstruction process after the template generation because the output is source code already. Then, after we get the candidate patches, we test all patches with the test suite and finally output the patches that pass all test cases. We consider the patch as a correct patch if and only if the patch is an exact match to the developer patch or semantically equivalent to the developer patch.

4.4 Training

To make sure the comparison between the two models is fair, we apply the same training process to the traditional model and the abstraction model. We train both models 12 epochs with 423,075 samples and validate the models using 470,10 samples.

The traditional model has an encoder dictionary with 215,664 tokens, a decoder dictionary with 224,608 tokens and 234,660,654 model parameters. The training of the traditional model completes in 20548.9 seconds (5.7 hours). The average training time for each epoch is 28.5 minutes.

The abstraction model has an encoder dictionary with 220 tokens, a decoder dictionary with 220 tokens and 11,852,562 model parameters. The training of the abstraction model completes in 5888.3 seconds (1.63 hours). The average training time for each epoch is 8.1 minutes. The above numbers are the key feature of the two models. The configuration details of both models are listed in Appendix A. Both models are trained on 56 Intel Xeon E5-2695 CPUs and a single NVIDIA Xp GPU.

4.5 Hyperparameter Search

We perform the hyperparameter search on both models for 100 iterations. As previous work mentioned [4], hyperparameter tuning should be performed specifically for models with different learning tasks to fairly compare the performance. In our work, we consider learning the bug-fix operations with and without abstraction as two different learning tasks. We perform random hyperparameter search on both models separately for 100 times within the following search space: size of embedding vectors form 50 to 500, convolution filter shape $128 * (1 - 5)^2$, stack of convolution filters from 1 to 10, drop out rate from 0 to 1, clip normalization parameter from 0 to 1, learning rate from 0 to 1, and momentum from 0 to 1. we choose the hyperparameter set that gives the best result from the 100 random parameter sets as the final hyperparameter set.

Chapter 5

Experimental Results

In this section, we present the result of our experiment. We compare the performance of the abstraction model and the traditional model in three aspects, including numbers of bug fixes, types of bug fixes and execution time of generating bug fixes. In our experiment, we define the reconstructed code as a candidate patch. We consider a candidate patch as a correct fix of one bug if the reconstructed code is the same as developer patch or semantically equivalent to the developer patch. Otherwise we consider the candidate patch as a failed attempt.

5.1 Comparison of Number of Correct Fixes Generated by Different Models

Table 5.1 shows the number of bugs correctly fixed by the traditional model and the abstraction model in the QuixBugs dataset. We get the result by running the test scripts on both model outputs for each bug in the QuixBugs dataset. In the total amount of 40 bugs in the QuixBugs dataset, the traditional model fixes 10 bugs and the abstraction model fixes 23 bugs. Column "All" shows the total number of fixes from each model. Column "Top-one" shows the number of bugs fixed correctly by the first patch generated for each model. Column "unique" shows the number of bugs fixed by one model but cannot be fixed by the other model. In the traditional model, there are nine out of ten fixes has the top-one rank. We consider a fix to be the top-one rank if the fix is the first correct fix generated by a model. In the other model, 22 fixes out of 23 bugs fixed are ranked as top-one. Its top-one ratio is close to the top-one ratio in the traditional model. Meaning

that the overfitting level of the abstraction model is as low as the traditional model. The abstraction technique does not rise the overfitting level in the traditional model. In the ten fixes of the traditional model, the abstraction model can fix nine of them as well. Meaning that the abstraction model fixes the majority (90%) of the bugs the traditional model fixes. The remaining 15 fixes in the abstraction model are unique to the traditional model. The above comparison shows that the abstraction model fixes not only 90% of the bugs the traditional model fixes but also 15 additional bugs the traditional model does not fix.

Fixes	All	Top-1	Unique
Original	10	9	1
Abstraction	23	22	14

Table 5.1: Comparison of the number of correct fixes

The abstraction model increases the number of total fixes by 140%. The fixes from the abstraction model cover most of the original fixes and also maintains a similar top-one ratio. This result shows that the abstraction model fixes more bugs than the traditional model without increasing the overfitting level. The rational and analysis of why the abstraction model fixes more bugs are described in chapter 6.

Index	Algorithm	traditional_rank	Abstraction_rank
5	find_first_in_sorted	4	5
7	flatten	1	44
8	gcd	111	28
10	hanoi	110	1
13	knapsack	1	3
20	mergesort	107	36
24	pascal	6	5
29	rpn_eval	133	46
34	sieve	3	3

Table 5.2: Comparison of ranks of shared correct fixes

In the bugs that both model fix, we compare the rank of fixes of both models. Recall that the rank of fixes is the rank of the successful patch among all patches. For example, in a list of patches generated from one model for one task, if the second patch repairs the bug,

Type	Name	abbreviation
A	Incorrect Assignment operator	IncAssignOp
B	Incorrect variable	IncVar
C	Incorrect comparison operator	IncCompOp
D	Missing condition	MisCond
E	Missing/added+1	MisOne
F	Variable swap	VarSwap
G	Incorrect array slice	IncArrSlc
J	Incorrect method called	IncMethCal
K	Incorrect field dereference	IncFldDeref
L	Missing arithmetic expression	MisArithExp
M	Missing function call	MisFuncCal
N	Missing line	MisLine

Table 5.3: Types of bugs in QuixBugs dataset

the rank of the fix for that task is two. Table 5.2 shows the rank of fixes in different models. From the table, we can see that the abstraction model reduces the rank dramatically in the case that the original rank is over 100. In algorithm gcd, Hanoi, mergesort, and rpn_eval, the rank of fixes in the traditional model are 111, 110, 107, and 133. After applying the abstraction technique, the rank of fixes decreases to 28, 1, 36, and 46 respectively. In the case that the original rank of the fix is less than ten, the rank of fix in the abstraction model stays at the same level. In algorithm find_first_in_soted, knapsack, Pascal, and sieve, the original rank of fixes are four, one, six, and three. The abstraction rank of the above bugs stays at a similar level, which is five, three, five and three. There is an exception to the above observation. In the algorithm flatten, the original rank is 1 and the rank with the abstraction model is 44. In general, the abstraction model increases the rank in the case that the rank is lower than 100. Also, in most cases, the abstraction model generates patches with ranks similar to the original rank if the original rank is relatively high.

5.2 Distribution of Bug Types

In our work, we analyze and compare the difference of the types of bug fixes from the two NMT-based models and also other existing models. According to this paper [86], the QuixBugs dataset contains 12 kinds of bugs. We use the same labeling and classification from this work for ease of discussion. Table 5.3 shows all the types of bugs defined by the

index	algorithm	type	Our	Baseline
0	bitcount	A	Yes	
1	breadth_first_search	D	Yes	
2	bucketsort	G	Yes	
5	find_first_in_sorted	C	Yes	Yes
7	flatten	J	Yes	Yes
8	gcd	F	Yes	Yes
9	get_factors	G	Yes	
10	hanoi	B	Yes	Yes
11	is_valid_parenthesization	B	Yes	
12	kheapsort	G	Yes	
13	knapsack	C	Yes	Yes
16	levenshtein	E	Yes	
17	lis	L	Yes	
20	mergesort	C	Yes	Yes
21	minimum_spanning_tree	B	Yes	
23	next_permutation	C	Yes	
24	pascal	B	Yes	Yes
25	possible_change	D	Yes	
27	quicksort	C	Yes	
29	rpn_eval	F	Yes	Yes
30	shortest_path_length	K	Yes	
31	shortest_path_lengths	F	Yes	
34	sieve	J	Yes	Yes
36	subsequences	N		Yes

Table 5.4: Comparison of type of bugs repaired

paper. The twelve bug classes in the paper are A: Incorrect Assignment operator (IncAssignOp), B: Incorrect variable (IncVar), C: Incorrect comparison operator (IncCompOp), D: Missing condition (MisCond), E: Missing/added+1 (MisOne), F: Variable swap (VarSwap), G: Incorrect array slice (IncArrSlc), J: Incorrect method called (IncMethCal), K: Incorrect field dereference (Inc FldDeref), L: Missing arithmetic expression (MisArithExp), M: Missing function call (MisFuncCal), and N: Missing line (MisLine).

In all 40 bugs from QuixBugs dataset, there are ten instances in class IncVar, five instances in class IncCompOp, IncArrSlc and MisLine, four instances in class VarSwap, three instances in class MisCond, two instances in class MisOne and IncMethCal and one instance for the class in IncAssignOp, Inc, MisArithExp, and MisFuncCal. Table 5.4 shows all bugs fixed by the models. From the table, we can see that the traditional model fixes

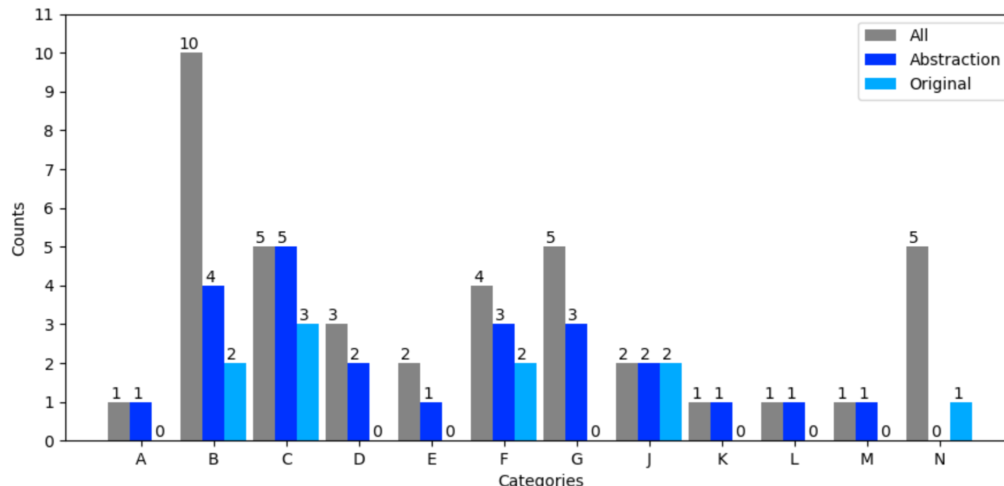

Figure 5.1: Comparison of type of bugs

three bugs in class IncCompOp, two bugs for each of class IncVar and VarSwap, and one bug in class IncMethCal. The abstraction model fixes all bugs in the table except bug 36 (subsequences).

Figure 5.1 shows the comparison of numbers of fixes for both models. The bars in gray color represent the total bugs in each class in the QuixBugs dataset. The bars in dark blue color represent the fixes in the abstraction model. The bars in light blue color indicate the bugs fixed by the traditional model. In all twelve types of bugs in the QUixBugs dataset, the abstraction model can fix all bugs in class IncAssignOp, IncCompOp, IncMethCal, FldDeref, MisArithExp, and MisLine, while class IncMethCal is the only class that the traditional model can fix every bug in that type. Comparing the numbers of fixes from two models, the abstraction model fixes two more bugs in class IncCompOp (next_permutation and quicksort), two more bugs in class IncVar (is_valid_parenthesization and minimum_spanning_tree), three bugs in class IncArrSlc , two bugs in class MisCond , and one bug of each type in class IncAssignOp, MisOne, MisArithExp, MisFuncCal, and IncFldDeref.

In terms of the coverage of different types of bugs, The traditional model fixes bugs in class IncVar, IncCompOp, VarSwap, IncArrSlc, and MisLine. In contrast, the abstraction model fixes all the bugs in the above categories except class MisLine. Also, the traditional model fixes bugs in class IncAssignOp, MisCond, MisOne, IncArrSlc, IncFldDeref, Mis-

ArithExp, and MisFuncCal, which cannot be fixed by the traditional model. This result shows the abstraction model not only covers more types of bugs than the traditional model but also fixes more bugs than the traditional model in the types of bugs that they both fix.

5.3 The Execution Time of Reconstruction

To measure the execution time of the reconstruction, we run the test suite on the end-to-end patch generation approach and the patch generation with reconstruction. Note that the rank of the correct patch can be different in the abstraction model and the traditional model. For each task, we calculate the average execution time of the patch using the execution time of that task divided by the number of total patches. For example, if task A has five candidate patches before the correct patch and the execution time to find the correct patch is ten seconds. The average patch execution time of task A is ten divided by five, which is 2 seconds per patch. Table 4 shows the model ranking, total execution time and execution time per patch for all samples fixed by both models in the QuixBugs dataset. Comparing with the end-to-end approach, the reconstruction needs extra 23.32% of total execution time. This percentage can be different in the bugs with a larger scope of context.

model		traditional			abstraction	
algorithm	rank	total (sec)	per patch	rank	total (sec)	per patch
find_first_in_sorted	4	2.15	0.54	14	10.45	0.75
flatten	1	0.048	0.048	58	2.51	0.043
gcd	111	6.80	0.06	10	0.40	0.04
mergesort	107	5.13	0.048	181	6.9	0.038
pascal	6	0.32	0.053	23	0.89	0.039
hanoi	110	4.19	0.038	3	0.24	0.08
rpn_eval	133	5.85	0.044	153	7.35	0.048
sieve	3	0.20	0.07	10	0.43	0.04

Table 5.5: Comparison of time cost between models

Chapter 6

Result Analysis

In this chapter, we analyze all bugs from the unique fixes of the abstraction model. We find some insights that explain why abstraction model performs better than the traditional model. After we compare the top ten candidates from the traditional model outputs with the top ten candidates from the abstraction model outputs, we found 4 reasons which could explain the difference in performance between these two models. Section 6.1 explains the structural differences in the output of the template model and the traditional model. Section 6.2 explains why synonyms affect the result and how the abstraction model alleviates this problem. Section 6.3 describes how the reduction of the out-of-vocabulary token rate contributes to the increase of bug fixes. Section 6.5 shows why the model has the potential of finding semantically equivalent fixes.

6.1 The Clustering of Candidates

The output of the abstraction model is a list of templates. Each template can be transferred to a list of source code candidates. Comparing to the original approach where each output is one source code candidates, the abstraction model outputs have a higher density of pattern representation. For example, table 6.1 shows the top-10 outputs for both models for bug one in Quixbugs, in bug one (breadth first search), the Var1 in the output is a new token to the traditional model input, indicating that Var1 does not exist in the input. We define these tokens as variable tokens (VTs) and the outputs with such tokens as templates. the VTs are placeholders for the real variables and will be replaced by some variables in the search space of the input context. Recall that the reconstruction process maps the VTs to each value in the context search space. For example, given an output template “while Var2.Var1:” and

a search space S={ update, successors, startnode }, The output of reconstruction is Out={ “while update.successors:”,“while successors.update:”,“while successors.startnode:”,“while startnode.successors:”,“while startnode.update:”,“while update.startnode:” }

In table 6.1, the candidate at rank one in the abstraction model (red-colored cell in column abstraction) has the same pattern with the candidate at rank three and rank seven in the traditional model (red-colored cell in column original). The abstraction model uses one position for each candidate template and fills in the VTs later. The traditional model needs two positions for these two results, which is inefficient.

By applying the abstraction technique, the model aggregates the output pattern of candidates three and candidates seven in candidate one, which reduces the space usage from two to one. By contrast, the traditional model spends two or more positions matching tokens for candidates with the same pattern. With the clustering property, the abstraction compresses the candidates in templates, which increase the possibility of finding fixes templates.

model	abstraction	traditional
input	while True:	while True:
output	while Var1:	while queue:
rank 1	while self.Var1:	while True:
rank 2	while Var1:	while not self._closing:
rank 3	while True:	while self.alive:
rank 4	while not self.Var1:	while not self._closing.is_set():
rank 5	while Var1 is not None:	while not self.exit_flag.is_set():
rank 6	while Var2.Var1:	while self.alive is not None:
rank 7	while self.Var2.Var1:	while self.closed:
rank 8	while self.Var1 is None:	while not self.exit_flag :
rank 9	while not Var2.Var1:	while len(lSendWaiting)>0:
rank 10	while self.Var1.Var2 :	while not self.shouldquit:
Context: Var1,Var2 $\in \{update, successors, startnode, queue, popleft$, nodesseen, node, goalnode, extend, deque, collections, append, add, Queue}		

Table 6.1: Top-10 results of the bug in Quixbugs dataset with file name breadth_first_search, bug ID 1

model	abstraction	traditional
input	return 1+Var1(Var2[1:], Var3[1:])	return 1+ levenshtein(source[1:],target[1:])
output	return levenshtein(source[1:],target[1:])	return levenshtein(source[1:],target[1:])
rank 1	return 1+levenshtein(source[1:],target[1:])	return 1+affineGapDistance(source[1:],target[1:])
rank 2	return 1+Var4(source[1:],target[1:])	return 1+levenshtein(source[1:],target[1:])
rank 3	return 0+levenshtein(source[1:],target[1:])	return 1+affineGapDistance(source[0:],target[1:])
rank 4	return levenshtein(source[1:],target[1:])	return 1+affineGapDistance(source[1:])
rank 5	return 1+levenshtein(source[1:],Var4[1:])	return 1+affineGapDistance(source[1:],target[2:])
rank 6	return Var4+levenshtein(source[1:],target[1:])	return 1+affineGapDistance(source[1:],targets[1:])
rank 7	return 1+levenshtein(source[0:],target[1:])	return 0+affineGapDistance(source[1:],target[1:])
rank 8	return 1+levenshtein(source[1:],target[1])	return 1+affineGapDistance(source[1:],target[0:])
rank 9	return 1+levenshtein(source[1:],target[1:]]	return 2+affineGapDistance(source[1:],target[1:])
rank 10	return True+levenshtein(source[1:],target[1:])	return 1+affineGapDistance(source[2:],target[1:])
Context: Var1 = levenshtein, Var2 = source, Var3 = target, Var4 $\in$ {}		

Table 6.2: Top-10 results of the bug in Quixbugs dataset with file name levenshtein, bug ID 16

6.2 The Tracking of Variables

In deep learning models, each word in the vocabulary is represented as vectors. The word embedding of the vector can be either learned by the model or imported from another model. In some cases, the vector distance between a word vector and its synonym vector is so close that these two vectors are used interchangeably in the process of template generation. The synonyms problem makes the template generation inefficient because the model wastes resources in generating similar patches with synonyms. The abstraction model alleviates this problem by tracking the variables. In the input sequences of the abstraction model, each user-defined variable is replaced by VT followed by an index. The mapping from user-defined variables to VTs is saved as a dictionary to replace the VTs to its original variable later. This mapping tracks the variables and avoids the confusion of synonyms.

For example, the table 6.2 shows the top-ten results of bug Levenshtein, in the original output of Bug Levenshtein, the traditional model generates fixes with the synonymous token "affineGapDistance" instead of the correct token "Levenshtein", which is a method of calculating string differences (red-colored cells in the original column). In the training process, the traditional model learns the similarity of the token "Levenshtein" and its synonyms "affineGapDistance", which is another algorithm for calculating string differences. The confusion of these two wards contributes to the failure of the fix generation in the end-to-end approach. However, this issue is being addressed in the abstraction model by replacing the user-defined tokens to VTs and labeled by index. In the reconstruction step,

the VTs are replaced to its original variable according to the mapping dictionary. Since the reconstruction only searches for user-defined tokens that are in the context of the input, this replacement removes the synonyms because the synonym “affineGapDistance” is not in the context of the input. The abstraction technique increases the chance of finding the fix by avoiding the matching of synonyms in the template generation.

The abstraction model not only saves the model from generating outputs with irrelevant tokens but also increase the diversity of output templates. The other approach, which simply filters out the irrelevant tokens from the original output will not receive good results. To prove this, we filter out the outputs that contain tokens outside the context search space from the traditional model and compare this filtered output to the output of the abstraction model to check if the filtered output performs as good as the abstraction model output. Table 6.3 shows the top-ten original output examples after the filtering process. We keep the candidate fixes that only contains tokens in the input or the bug context. Only 20 out of 220 candidates remain after the filtering. From the result, we found that the correct fix is not in the 20 filtered results. This is because the traditional model focuses on matching every detail of the output which results in the exhaustion of beam search potentials. With a level of abstraction to the source code, the model focuses on predicting the fixed template instead of a perfect detailed fix.

Rank	Filtered Original Output
Input	return 1+levenshtein(source[1:],target[1:])
Output	return levenshtein(source[1:],target[1:])
2	return 1 + levenshtein(source[1:],target[1:])
27	return 1 + levenshtein(source[1:])
39	return 1 + levenshtein(source[1:],target[1:],target)
45	return 1 + levenshtein(source,target[1:],target[1:])
46	return (1 + levenshtein(source[1:],target[1:]))
50	return 1 + levenshtein(source[1:]+target[1:])
53	return 1 + levenshtein(source[1:],target,target[1:])
67	return 1 + levenshtein(sources[1:],target[1:])
81	return 1 + levenshtein(target[1:],target[1:])
94	return 1 + levenshtein(source[1:],target)

Table 6.3: Top-10 filtered results of the bug in file reversed_linked_list in Quixbugs dataset with bug ID 28

model	abstraction	traditional
input	node = Var1	node = nextnode
output	Var3 = node	<<unk>>= node
rank 1	node = Var3	node = cached_requirements
rank 2	Var3 = nextnode	node = nextnode
rank 3	node = nextnode	node = get_original_host()
rank 4	node = Var3.nextnode	node = self._xtickmarkers
rank 5	node = self.Var3	node = []
rank 6	node = Var4.Var3	node = self.MockClass
rank 7	node = self.nextnode)	node = self.LONG_SLEEP_INTERRUPT_TIMEOUT
rank 8	node = nextnode()	node = self._existing_object
rank 9	node = Var3()	node = LazyInstanceClassName
rank 10	node = 1	node = self.cache1_md5
Var1 = nextnode, Var2 = node, Var3,Var4 $\in \{prevnode, reverse_linked_list, successor\}$		

Table 6.4: Top-10 results of the bug in Quixbugs dataset with file name reversed_linked_list, bug ID 28

6.3 The Decrease of Out-of-Vocabulary Rate

In the traditional model, some of the tokens in the developer patch are out-of-vocabulary (OOV), thus being replaced by token <<unk>> (red-colored cell in the original column), which makes the output syntactically incomplete (i.e., missing information). As a result, the traditional model is unable to predict such output because the correct token is not in the vocabulary of the model. We pick 47010 random samples as a validation set and check the OOV rate. With the aid of the abstraction technique, the rate of unknown token reduces from 2.7% in the traditional model to 0.00021% in the abstraction model. The abstraction model does not require all the tokens in the developer patch to exist in the vocabulary of the model. This is because the abstraction model predicts the pattern first, then searches for the missing tokens in its context search space which is additional to the model vocabulary. Moreover, the abstraction technique alleviates the OOV token problem and thus accepts a broader variation of inputs compared to the traditional model. For example, table 6.4 shows the top-ten result of bug 28, the correct output is “prevnode = node”. The token “prevnode” is in replacement of token <<unk>>because it is not in the vocabulary of the traditional model. The traditional model cannot fix this bug because the buggy line does not have token “prevnode” and the traditional model cannot generate tokens that are not in the model vocabulary. However, the abstraction model is capable of

fixing this bug because the token "prevnode" is in the context search space of the input, therefore it can be generated by the reconstruction process.

Comparing to the traditional model, the abstraction model extracts tokens from the context of the bug. If we provide the context information to the traditional model,i.e., inject the context tokens to the dictionary of the traditional model, this additional information still cannot be used directly in the traditional model. The reason is that if we add these context tokens to the dictionary of the traditional model, the traditional model has zero chance of generating candidate fixes that contains these context tokens because the model never saw any training data that has these tokens, otherwise these tokens should already exist in the model dictionary. Recall that the model dictionary is a collection of all tokens in the training dataset. If one context token occurs in the training dataset, the token will be collected by the model dictionary.

6.4 The Unique Fix From Baseline

For the bug that only fixed by the baseline, We compared the patch list generate by the baseline and the abstraction model. Table 6.5 shows the comparison of top-10 result of the patch lists. For bug subsequence, the input of the model is return [ ] for both models. The output of the model is return [[ ]] for both the baseline and the abstraction model.

model	abstraction	traditional
input	return []	return []
output	return [[ ]]	return [[ ]]
rank 1	return [ ]	return [ ]
rank 2	return { }	return Var1 ( )
rank 3	return [ ],[ ]	return { }
rank 4	return None	return Var1.Var2 ( )
rank 5	return set ( )	return None
rank 6	return ( )	return Var1
rank 7	defer.returnValue([ ])	return [ ],[ ]
rank 8	return { },[ ]	return Var2.Var1 ( )
rank 9	return list([ ])	return ( )
rank 10	return list({ })	return ( ),[ ]

Table 6.5: Top-10 results of the bug in Quixbugs dataset with file name Subsequence

From the table we can see that, the input and the output are identical for both baseline and the abstraction. Variable tokens are not exist in the input nor output. In this case, the abstraction is equivalent to not being applied to this bug. Based on our analysis to this bug, we think the absence of variable tokens contributes to the non-successful repair to this bug.

6.5 Semantically Equivalent Fix

There are instances that the fix generated by the abstraction model is not the same as the developer patch but semantically equivalent to it. For example, In bug 11, the developer patch is return depth == 0 and the patch generated by the abstraction model is return not depth. These two patches are semantically equivalent in python3 because they both triggered the same root function __nonzero__. Another example is bug 23. The developer patch of bug 23 is if perm[i] <perm[j]: and the model output is if perm[j] >perm[i] :, which is semantically equivalent to the developer patch. In the testing result of QuixBugs, the semantically equivalent fixes are normally ranked at top-one, which means the model is capable of generating semantically equivalent fixes without introducing overfitted patches.

Chapter 7

Threats

7.1 Multi-line Bugs

In the data collection process, we collected 519,452 bug-fix pairs from top 1000 popular python GitHub projects. We select only the bugs that contain single line fixes out of all kinds of bugs. In practice, Single line fixes is a small part of all types of bugs. In result, the dataset does not include multi-line bugs. So, there is no instance represents the fixing operations that only exist in multi-line bugs in the collected dataset. Since the dataset is the only data source that is used to train the model, the model learns only the fixing operations of single-line bugs. The pattern of multi-line bugs and corresponding fixes can be different from single-line bugs. Even if the fixing operation of single-line bugs is will-learned, the model may not perform well on multi-line bug fixing.

7.2 Random Hyperparameter Search

In the tuning process, we randomly generate 100 sets of hyperparameters and run the experiment 100 times with each set of hyperparameters for both the traditional model and the abstraction models. We pick the set of hyperparameters that returns the best result for both model. Although we tried to make the comparison as fair as possible, the best result of two sets of 100 random hyperparameters does not guarantee the same level of tuning performance.

Chapter 8

Conclusion

Program repairing is a critical and time consuming part in day-to-day software development cycle. Automated program repair tools are created to reduce the cost of program repair. The goal of these tools is generating program patches with less or without human effort in the process of repairing. The classic approach to Program repair is Generate and Validate approach. These type of tools generates a lot of patch candidates by mutates the buggy code with a list of mutants[41]. This approach suffer from the over-fitting problem[68]. Another new approach is to use a neural machine translation model to generate patches. The limitation of this method is that training takes a long time, and if the size of the training data set is large, accuracy cannot be guaranteed.

In this thesis, we propose an abstraction technique on top of the end-to-end Neural machine translation approach to alleviate the problems in the end-to-end approach and improve the result of the fix generation. We collected 519,452 bug-fix pairs and convert each pair to templates using the abstraction algorithm. We train the f-conv model using 423,075 bug-fix templates and validate the model using 470,10 samples. We use the trained model to predict the outputs and convert the outputs back to the source code using the reconstruction process. We train the model 12 epochs with 100 random sets of hyperparameters and choose the model with the best validation result.

We choose the traditional end-to-end approach which does not have the abstraction and reconstruction as the baseline and compare it with the abstraction model. The accuracy of the fix generation increased form 25% (10 out of 40) in the traditional model to 57% (23 out of 40) in the abstraction model. The result of the abstraction model covers 9 out of 10 fixes of the traditional model. The type coverage of bugs increased from five types in the traditional model to eleven types in the abstraction model The remaining fifteen fixes from

abstraction models are unique to the traditional model. The training time is reduced from 5.7 hours in the traditional model to 1.63 hours. The size of the encoder dictionary reduced from 215,664 tokens in the traditional model to 220 tokens in the abstraction model. The size of the decoder dictionary reduced from 224,608 tokens to 220 tokens.

The experimental results show that our method significantly reduces the time cost and the vocabulary size of training. Since our approach is an overlay of pre-possessing and post-processing to the existing neural machine translation rather than modifying the model, our approach can be applied to other existing NMT-based automated program repair tool.

Chapter 9

Future Work

9.1 Multi-line templates

The Single-line bug is one simple scenario among many different bug scenarios. There are many other kinds of bugs we do not take into consideration. To support multi-line program repair, We need to further analysis multi-line bug scenarios and find a better way to represent the bug rather than simply flatten multiply buggy lines to a one-dimensional sequence.

9.2 Alternative Abstractions

In our abstraction technique, we abstract user-defined values and variable names to variable tokens (VTs). There are many other ways to consider in the abstraction from source code to templates. We will explore more on different abstraction mappings and compare the effectiveness. Hopefully, it will give us more insight to get the best abstraction design and also answer the question that what information is more important in fixing software bugs.

References

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

[2] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

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

[4] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *International conference on machine learning*, pages 199–207, 2013.

[5] Issam Bazzi. *Modelling out-of-vocabulary words for robust speech recognition.* PhD thesis, Massachusetts Institute of Technology, 2002.

[6] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70. IEEE, 2018.

[7] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[8] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995.

[9] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

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

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

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

[13] Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. Npefix: Automatic runtime repair of null pointer exceptions in java. *arXiv preprint arXiv:1512.07423*, 2015.

[14] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286. ACM, 2012.

[15] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org, 2017.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

[18] Tobias Domhan. How much attention do you need? a granular analysis of neural machine translation architectures. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1799–1808, 2018.

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

[20] Sergey Edunov, Myle Ott, Michael Auli, David Grangier, and Marc'Aurelio Ranzato. Classical structured prediction losses for sequence to sequence learning. *arXiv preprint arXiv:1711.04956*, 2017.

[21] Steven Fortune. A note on sparse complete sets. *SIAM Journal on Computing*, 8(3):431–433, 1979.

[22] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org, 2017.

[23] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *Proc. of ICML*, 2017.

[24] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[25] Oded Goldreich. Computational complexity: a conceptual perspective. *ACM Sigact News*, 39(3):35–39, 2008.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[27] Linda Grandell, Mia Peltomäki, Ralph-Johan Back, and Tapio Salakoski. Why complicate things?: introducing programming in high school using python. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 71–80. Australian Computer Society, Inc., 2006.

[28] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[29] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

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

[31] Brandon Heller, Eli Marschner, Evan Rosenfeld, and Jeffrey Heer. Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th working conference on mining software repositories*, pages 223–226. ACM, 2011.

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

[33] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[34] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. *arXiv preprint arXiv:1708.09492*, 2017.

[35] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. *arXiv preprint arXiv:1905.13334*, 2019.

[36] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[37] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[38] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[40] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8each.InSoftwareEngineering(ICSE), 201234thInternationalConferenceon, pages3--13.IEEE, 2012.

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

[42] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

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

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

[45] Fan Long and Martin Rinard. Prophet: Automatic patch generation via learning from successful human patches. 2015.

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

[47] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, pages 298–312. ACM, 2016.

[48] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78. IEEE, 2015.

[49] Thibaud Lutellier, Lawrence Pang, Viet Hung Pham, Moshi Wei, and Lin Tan. Encore: Ensemble learning using convolution neural machine translation for automatic program repair. *arXiv preprint arXiv:1906.08691*, 2019.

[50] Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. Towards an automated approach for bug fix pattern detection. *arXiv preprint arXiv:1807.11286*, 2018.

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

[52] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 448–458. IEEE Press, 2015.

[53] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.

[54] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, volume 2, page 4, 2016.

[55] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[56] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 902–912. IEEE, 2015.

[57] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[58] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[59] Tim Peters. The zen of python. In *Pro Python*, pages 301–302. Springer, 2010.

[60] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

[61] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[62] Brian D Ripley and NL Hjort. *Pattern recognition and neural networks*. Cambridge university press, 1996.

[63] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 648–659. IEEE, 2017.

[64] Ripon K Saha, Hiroaki Yoshida, Mukul R Prasad, Susumu Tokumoto, Kuniharu Takayama, and Isao Nanba. Elixir: an automated repair tool for java programs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 77–80. ACM, 2018.

[65] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[66] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[67] Aliaksei Severyn and Alessandro Moschitti. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 373–382. ACM, 2015.

[68] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[69] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[70] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[71] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[72] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 471–482. IEEE Press, 2015.

[73] Hung Viet Phan Moshi Wei Lin Tan Thibaud Lutellier, Lawrence Pang. Encore: Ensemble learning using convolution neural machine translation for automatic program repair. 2019.

[74] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. *arXiv preprint arXiv:1901.09102*, 2019.

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

[76] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. *arXiv preprint arXiv:1812.10772*, 2018.

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

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

[79] Song Wang. Leveraging machine learning to improve software reliability. 2019.

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

[81] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. *arXiv preprint arXiv:1707.04742*, 2017.

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

[83] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2016.

[84] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair

of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

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

[86] He Ye, Matias Martinez, and Martin Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *arXiv preprint arXiv:1805.03454*, 2018.

[87] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *arXiv preprint arXiv:1712.07804*, 2017.

APPENDICES

Appendix A

Model configuration

This appendix provides the configuration details of the traditional model and the abstraction model to help other researcher reproducing the result. The difference of parameters between two models are highlighted in red.

A.1 traditional model configuration details

arch=fconv
bucket_cap_mb=150
clip_norm=0.4828638760240562
criterion=label_smoothed_cross_entropy
data=[../fairseq-data/bin]
ddp_backend=c10d
decoder_attention=True
decoder_embed_dim=290
decoder_embed_path=None
decoder_layers=[(512\7)] * 8
decoder_out_embed_dim=197
device_id=0
distributed_backend=nccl
distributed_init_method=None
distributed_port=-1
distributed_rank=0

distributed_world_size=1
dropout=0.31636707239399064
encoder_embed_dim=290
encoder_embed_path=None
encoder_layers=[(512\7)] * 8
fix_batches_to_gpus=False
fp16=True
fp16_init_scale=128
keep_interval_updates=-1
label_smoothing=0.0
left_pad_source=True
left_pad_target=False
log_format=None
log_interval=1000
lr=[0.5774274889402233]
lr_scheduler=reduce_lr_on_plateau
lr_shrink=0.1
max_epoch=12
max_sentences=32
max_sentences_valid=32
max_source_positions=1024
max_target_positions=1024
max_tokens=1000
max_update=0
min_loss_scale=0.0001
min_lr=0.0001
momentum=0.6168128191126693
no_epoch_checkpoints=False
no_progress_bar=False
no_save=False
optimizer=nag
optimizer_overrides={}
raw_text=False
reset_lr_scheduler=False
reset_optimizer=False
restore_file=checkpoint_last.pt
save_dir=../fairseq-data/model

save_interval=1
save_interval_updates=0
seed=1
sentence_avg=False
share_input_output_embed=False
skip_invalid_size_inputs_valid_test=False
source_lang=None
target_lang=None
task=translation
train_subset=train
update_freq=[1]
upsample_primary=1
valid_subset=valid
validate_interval=1
weight_decay=0.0

A.2 abstraction model configuration details

arch=fconv
bucket_cap_mb=150
clip_norm=0.4553322458425517
criterion=label_smoothed_cross_entropy
data=[../fairseq-data/bin]
ddp_backend=c10d
decoder_attention=True
decoder_embed_dim=312
decoder_embed_path=None
decoder_layers=[(256\6)] * 2
decoder_out_embed_dim=469
device_id=0
distributed_backend=nccl
distributed_init_method=None
distributed_port=-1
distributed_rank=0
distributed_world_size=1
dropout=0.22343933879238076

encoder_embed_dim=312
encoder_embed_path=None
encoder_layers=[(256\6)] * 2
fix_batches_to_gpus=False
fp16=True
fp16_init_scale=128
keep_interval_updates=-1
label_smoothing=0.0
left_pad_source=True
left_pad_target=False
log_format=None
log_interval=1000
lr=[0.7896441745393431]
lr_scheduler=reduce_lr_on_plateau
lr_shrink=0.1
max_epoch=12
max_sentences=32
max_sentences_valid=32
max_source_positions=1024
max_target_positions=1024
max_tokens=1000
max_update=0
min_loss_scale=0.0001
min_lr=0.0001
momentum=0.7680298739590937
no_epoch_checkpoints=False
no_progress_bar=False
no_save=False
optimizer=nag
optimizer_overrides={}
raw_text=False
reset_lr_scheduler=False
reset_optimizer=False
restore_file=checkpoint_last.pt
save_dir=../fairseq-data/model
save_interval=1
save_interval_updates=0
seed=1

sentence_avg=False
share_input_output_embed=False
skip_invalid_size_inputs_valid_test=False
source_lang=None
target_lang=None
task=translation
train_subset=train
update_freq=[1]
upsample_primary=1
valid_subset=valid
validate_interval=1
weight_decay=0.0