DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Exploring the Usage of Neural Networks for Repairing Static Analysis Warnings

VINCENT PAUL LOHSE

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Exploring the Usage of Neural Networks for Repairing Static Analysis Warnings

VINCENT PAUL LOHSE

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: November 21, 2021

Supervisor: Zimin Chen
Examiner: Martin Monperrus
School of Electrical Engineering and Computer Science
Host company: DICE (EA Digital Illusions CE AB)
Swedish title: Utforsking av användningen av neurala nätverk för att reparera varningar för statisk analys

Abstract

C# provides static analysis libraries for template-based code analysis and code fixing. These libraries have been used by the open-source community to generate numerous NuGet packages for different use-cases. However, due to the unstructured vastness of these packages, it is difficult to find the ones required for a project and creating new analyzers and fixers take time and effort to create. Therefore, this thesis proposes a neural network, which firstly imitates existing fixers and secondly extrapolates to fixes of unseen diagnostics. To do so, the state-of-the-art of static analysis NuGet packages is examined and further used to generate a dataset with diagnostics and corresponding code fixes for 24,622 data points. Since many C# fixers apply formatting changes, all formatting is preserved in the dataset. Furthermore, since the fixers also apply identifier changes, the tokenization of the dataset is varied between splitting identifiers by camelcase and preserving them. The neural network uses a sequence-to-sequence learning approach with the Transformer model and takes file context, diagnostic message and location as input and predicts a diff as output. It is capable of imitating 46.3% of the fixes, normalized by diagnostic type, and for data points with unseen diagnostics, it is able to extrapolate to 11.9% of normalized data points. For both experiments, splitting identifiers by camelcase produces the best results. Lastly, it is found that a higher proportion of formatting tokens in input has minimal positive impact on prediction success rates, whereas the proportion of formatting in output has no impact on success rates.

Keywords

Automatic Program Repair, Neural Machine Translation, Static Analysis, Transformer Model, Formatting

Sammanfattning

C# tillhandahåller statiska analysbibliotek för mallbaserad kodanalys och kodfixering. Dessa bibliotek har använts av open source-gemenskapen för att generera många NuGet-paket för olika användningsfall. Men på grund av mängden av dessa paket är det svårt att hitta de som krävs för ett projekt och att skapa nya analysatorer och fixare tar tid och ansträngning att skapa. Därför föreslår denna avhandling ett neuralt nätverk, som för det första imiterar befintliga korrigeringar och för det andra extrapolerar till korrigeringar av osynlig diagnostik. För att göra det har det senaste inom statisk analys NuGet-paketen undersökts och vidare använts för att generera en datauppsättning med diagnostik och motsvarande kodfixar för 24 622 datapunkter. Eftersom många C# fixers tillämpar formateringsändringar, bevaras all formatering i datasetet. Dessutom, eftersom fixarna också tillämpar identifieringsändringar, varieras tokeniseringen av datamängden mellan att dela upp identifierare efter camelcase och att bevara dem. Det neurala nätverket använder en sekvens-till-sekvens-inlärningsmetod med Transformer-modellen och tar filkontext, diagnostiskt meddelande och plats som indata och förutsäger en skillnad som utdata. Den kan imitera 46,3% av korrigeringarna, normaliserade efter diagnostisk typ, och för datapunkter med osynlig diagnostik kan den extrapolera till 11,9% av normaliserade datapunkter. För båda experimenten ger uppdelning av identifierare efter camelcase de bästa resultaten. Slutligen har det visat sig att en högre andel formateringstokens i indata har minimal positiv inverkan på åndelen korrekta förutsägelser, medan andelen formatering i utdata inte har någon inverkan på åndelen korrekta förutsägelser.

Nyckelord

Automatisk programreparation, neural maskinöversättning, statisk analys, transformatormodell, formatering

Acknowledgments

A handful of people have helped in initiating and writing this thesis. I would specifically like to thank my examiner Professor Martin Monperrus for supporting me through the entire journey. This included connecting me with DICE, quickly and thoroughly answering every email I had regarding the core research methodology of this thesis and giving very useful advice whilst it progressed.

I would also like to thank my supervisors at DICE, Cristian Deri and Niklas Lindgren. Even though shouldering a heavy workload with the release of Battlefield 2042, Cristian regularly found the time to assist me in weekly meetings and was open to many scientific discussions. Not to forget, Cristian also allowed a very open approach to the task, which gave me a comfortable space of creative experimentation. Niklas, on the other hand, was always available for hands-on C# programming assistance, which was extremely helpful for me to avoid hitting walls.

A great aid in the field of neural machine translation for automatic program repair was my supervisor at KTH, Zimin Chen. Being able to discuss the machine learning technicalities with him gave this thesis a clear qualitative upgrade. Apart from this, other Master/PhD students at the ASSERT lab were always available for quick brainstorms, which made a significant difference in defining the scope of the problem. Here, I would specifically like to thank Nicolas Harrand, Javier Cabrera and Javier Ron.

Last, but not least, I would also like to thank my family for having supported me so far and without whom I would not have been able to put in the amount of time and effort to work on this thesis as I have.

Stockholm, November 2021
Vincent Paul Lohse

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

A3C Asynchronous Advantage Actor-Critic

APR Automatic Program Repair

AST Abstract Syntax Tree

CI Continuous Integration

CLI Command-line Interface

CWE Common Weakness Enumeration

DICE Digital Illusions Creative Entertainment

GAN Generative Adversarial Networks

GGNN Graph Gated Sequence Neural Network

GNN Graph Neural Network

GRU Gated Recurrent Unit

IDE Integrated Development Environment

LSTM Long-Term Short-Term Memory

NLTK Neural Language Toolkit

NMT Neural Machine Translation

NN Neural Network

PCA Principal Component Analysis

RL Reinforcement Learning

RNN Recurrent Neural Network

SDK Software Development Kit

seq2seq sequence-to-sequence

UML Unified Modeling Language

VS Visual Studio

Chapter 1

Introduction

1.1 Context

Static analysis is used on programming languages and checks code for bugs or bad practices without executing the code. This means code is read, perhaps also compiled, and checked for specific predefined patterns, which are labelled potentially problematic. In some cases, static analysis also provides fixes that change these patterns to remove the assumed problem.

Even though third-party vendors such as SonarQube, ReSharper or Coverity offer static analysis tools for various programming languages, C# already provides code analysis libraries created by Microsoft. Using these libraries and through the effort of open-source contributors, the landscape of free and application-specific analyzers has been able to grow in terms of NuGet packages. A NuGet is an open-source .NET package/library that can be installed with .NET's package manager also called NuGet.

The fixing mechanism in static analysis is a subset of Automatic Program Repair (APR), whereby code is changed in order to avoid unexpected behaviour. For APR, the usage of neural networks has become increasingly popular both in research and in industry. Often, datasets have been mined from bug-fixes in open-source repositories, which have then been used for neural training. Hereby, research in Neural Machine Translation (NMT) has been used to abstract the idea of fixing code to translating one language (buggy code) into another (fixed code). The most common research within this field has been sequence-to-sequence learning, whereby input and output of the neural network are code tokens. The types of network models vary from different Recurrent Neural Networks (RNNs) to the Transformer model.

1.2 Problem Statement

By supporting the distribution of third-party static analysis tools in C# through code analysis libraries, Microsoft has fostered the growth of a vast ecosystem of NuGet libraries. When searching for the keyword "analyzer" on nuget.org, 800 open-source NuGet packages appear. The resources have grown so abundant and intransparent, that even similar implementations of the same analysis have been documented.[1]

With the active development of the C# language, new analyzers are still being created for various specific use-cases. Examples of such specific NuGet packages are "WebOptimizer.Analyzers" or "AsyncUsageAnalyzers".[2][3] Additionally to the difficulty of understanding whether certain template-based fixes have already been written, writing them takes time and effort for human developers.

This thesis tackles these problems by proposing a neural network that can specifically imitate the fixing mechanism of existing static analyzers and also extrapolate to new fixes, given any diagnostic message. This aims to simplify the required code-base for static analysis fixing and eliminate the time required to create new fixers.

1.3 Research Questions

RQ1) What is the state of the art of open-source static analysis in C#?

When querying nuget.org for "analyzer", around 800 NuGets are shown. This thesis will explore what this number actually means and to what extent these packages can actually be used for static analysis. In relation to the following research questions, the question will also be used to answer whether this landscape of analyzer tools suffices to train a neural network to generate code fixes.

RQ2) To what extent can a neural network learn to imitate existing fixers?

This question aims at the problem statement of having a large and parcelled environment of open-source static analyzers with potential duplications. Is it possible to create one Neural Network (NN)-based static analyzer "to rule them all"? Furthermore, given that analyzers may also suggest incorrect fixes, can

[1] https://github.com/meziantou/Meziantou.Analyzer/blob/main/docs/comparison-with-other-analyzers.md [2] https://www.nuget.org/packages/LigerShark.WebOptimizer.Analyzers/ [3] https://www.nuget.org/packages/AsyncUsageAnalyzers/1.0.0-alpha003

a neural network use the "knowledge of the crowd" to even compensate for wrong fixes?

RQ3) To what extent can a neural network learn to extrapolate from existing fixers?

Given the problem statement of static analyzers being written on a regular basis and therefore continuously requiring engineering time, this question explores to what extent knowledge contained in existing fixers can be used to extrapolate to new fixes.

Furthermore, this RQ is related to research in APR, where unseen bugs are fixed "in the wild". Such research usually relies on bug-fix mining through git commits filtered by certain keywords, which however does not always guarantee that a data point is actually a fix. Therefore, this RQ will shed lights on whether static analysis fixes can be used for this task as well.

RQ4) How do formatting tokens influence sequence-to-sequence neural network learning?

As there are multiple analyzer NuGets available that implement code refactorings to change code formatting, this thesis further explores how formatting information can affect a NNs predictions. This is done to understand whether formatting tokens can be viewed as tokens with less contained data and are therefore easier to understand or predict for the given neural network architecture.

1.4 Contributions

The contributions of this thesis are as follows:

- A quantification of the landscape of NuGet packages related to static analysis
- A C# dataset of 25k fixes from 421 diagnostics on 18 open-source C# repositories, including diagnostic messages and tokenized formatting
- Novel results of a sequence-to-sequence learning neural network, that aims to imitate static analysis fixers and extrapolate to new diagnostics
- Empirical evidence that the same network can fix both syntactic and formatting changes.

Chapter 2

Background

2.1 Types of Static Analysis Warnings

When analyzing code, static analyzers attempt to identify locations in the code that can lead to unexpected behaviour at runtime. The code entities that trigger these events are labelled code bugs. In the worst case, a code bug can also represent a software vulnerability, which is when an adversary is given more permissions than foreseen by the creators of the software. This can for instance be in terms data or machine access. Code smell on the other hand is a series of bad practices, that make the code more complex or difficult to understand. This can hint towards hidden code bugs. A simple method of addressing this problem are code refactorings, which are any changes made to the code that do not modify its behaviour. This does not remove remove bugs, but may make them more visible. Static analyzers may detect both code bugs and code smell, and thereby either provide fixes or refactorings.

2.2 Static Analysis in C#

Static analysis encompasses any method of inspecting code without executing it. This primarily stands in contrast to dynamic analysis, which can for instance mean executing test suites that run code and check for a predefined outcome. Static analysis can be used to identify code refactoring opportunities or point out direct code bugs/vulnerabilities. In the classical sense static analysis is rule-based, which is hard-coded by humans. Any programming pattern that can be defined by a rule can therefore also be identified by a static analysis rule. Static analysis further sees its application in linter programs or as Continuous Integration (CI) implementations. The former is similar to seeing

a colored underline due to a grammar mistake in natural language editors such as Microsoft Word.

In C#, static analysis is built upon the following four abstract classes:

- Microsoft.CodeAnalysis.Diagnostics.**DiagnosticAnalyzer**
- Microsoft.CodeAnalysis.CodeFixes.**CodeFixProvider**
- Microsoft.CodeAnalysis.CodeFixes.**FixAllProvider**
- Microsoft.CodeAnalysis.CodeRefactorings.**CodeRefactoringProvider**

These can be instantiated by the open-source community to build custom static analysis tools that integrate well into Visual Studio (VS). A Microsoft tutorial on this can be found in [1]. The following explanation of how these classes work together is accompanied by a simplified Unified Modeling Language (UML) diagram in Figure 2.1.

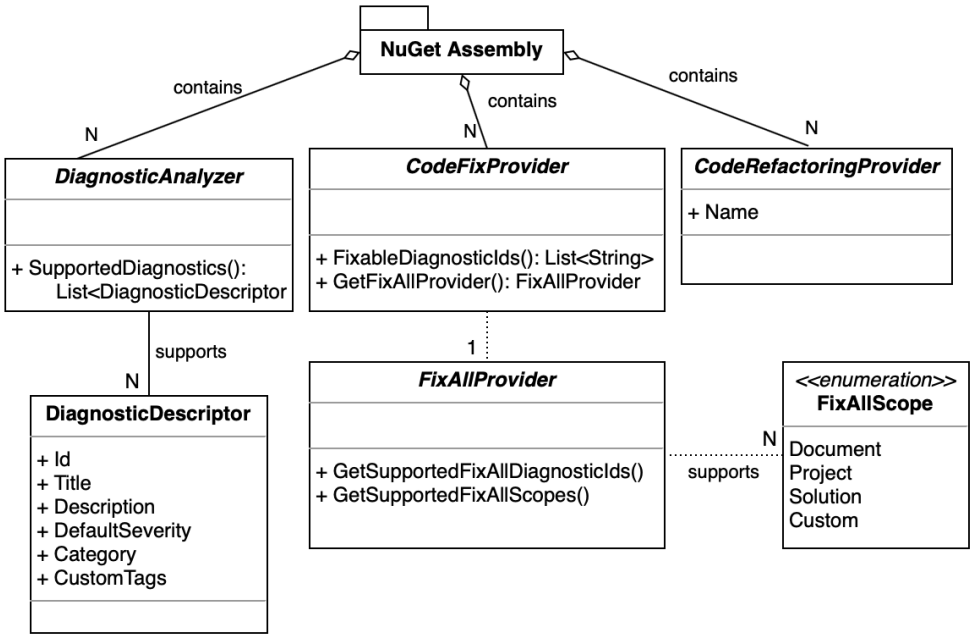

Figure 2.1: Static Analysis Classes in C#

Any C# assembly built along the lines of the tutorial [1], can include multiple instances of a DiagnosticAnalyzer, in short "analyzers", which in turn can "support" multiple "diagnostics". A diagnostic is a pattern the analyzer

will search for in the given code base and is identified by its ID. When integrated in VS, the areas where such patterns are detected will be highlighted by the Integrated Development Environment (IDE)'s linter.

On the other hand, the C# assembly can also contain multiple instances of a CodeFixProvider, in short "fixers", whereby one fixer can fix multiple different diagnostics. When instantiating a CodeFixProvider, a FixAllProvider can also be customized, which defines the fixers behaviour when attempting to fix all occurrences of a diagnostic in a given scope. Possible scopes can be restricted in the FixAllProvider to any set of combinations of "Document", "Project", "Solution" and "Custom".

Lastly, the "CodeRefactoringProvider" provides opportunities to manipulate code without the usage of analyzers or diagnostics. Since this class does not provide any title or description fields, it is largely ignored in this thesis. Furthermore, any code refactoring (see definition in Section 2.1) can be and has occasionally been implemented using CodeFixProviders as well, where documentation allows for better understanding of what the code manipulation does. Table 2.1 shows an example. Since part of this thesis is to quantify the landscape of open-source static analysis in C#, metrics of these findings will be shown in the results section.

Table 2.1: Example Code Diagnostic for Refactoring

NuGet Package	StyleCop.Analyzers.1.1.118
Diagnostic ID	SX1309
Diagnostic Title	Field names should begin with underscore
Diagnostic Description	A field name in C# does not begin with an underscore.
Default Severity	Warning

It is notable that static analysis built with these tools require the usage of C#'s compiler Software Development Kit (SDK) "Roslyn", which allows analysis to be done on an AST-level. Furthermore, Microsoft suggests that both "stateless" and "stateful" analysis can be achieved given these tools, whereby corresponding examples of these can be found in the documentation.[1] As anybody can however build and publish tools using the code analysis classes, there exist analyzers that detect similar patterns. Examples of such similar diagnostics can be found on GitHub.[2]

[1] https://github.com/dotnet/roslyn-sdk/tree/main/samples/CSharp/Analyzers/Analyzers.Implementation [2] https://github.com/meziantou/Meziantou.Analyzer/blob/main/docs/

2.3 Automatic Program Repair

According to Monperrus et al., APR is the "transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification" [2]. Monperrus continues classifying between Behavioural Repair, to which static analysis belongs, and State Repair. Whilst the former consists of changing code, either on source or on binary level, the latter describes changing program execution during runtime.

In this thesis, the usage of the term APR will be slightly widened. As the static analysis model in C#, described in Section 2.2, allows code refactorings and has in fact also occasionally been implemented in such a manner by open-source contributors, code refactorings will also fall under the scope of APR in this thesis.

2.4 Neural Machine Translation

NMT means using neural networks to translate a sequence of words from one language to another. Using NMT in APR takes the concept of translating one language to another, to translating buggy code to fixed code. This is done by mapping statistical dependencies between words in one language to words in another language and is most commonly done with RNNs [3]. RNNs are used when the input and/or output of a NN appear in a sequential manner and therefore one input/output will affect the next. Long-Term Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two important implementations of RNNs [4][5].

RNNs are notorious to struggle performance-wise when input and output length is large, as the effect of one input/output unit affecting the next, is that neural back-propagation results in a large chain of dependencies [6]. In recent years, the Transformer model by Vaswani et al. has surged in popularity as it avoids feeding input sequentially by giving input units a positional encoding [7]. Thereby, input units at the beginning of sequences can maintain just as strong dependencies to input units at the end of the sequence, as to consecutive ones. Consequently, back-propagation avoids the long dependency chain and the NN can therefore also learn faster.

`comparison-with-other-analyzers.md`

2.5 Neural Input Representations for Code

There are different strategies to describe code that will be inputted into a neural network. Since NMT describes sequence-to-sequence (seq2seq) learning, the inputted code is tokenized. Tokenizing describes the act of splitting text into a sequence of entities, whereby each entity defines a feature of the text. Generally, there exists a trade-off between using fine granularity tokenization (e.g. splitting by character) and low granularity (e.g splitting by white-space). This is because on the one hand, more input/output tokens force a NN to either understand or predict more sequential information, but more types of tokens also implies more information within one token.

Whilst tokenization in natural language may appear natural, since every word is separated by white-space, it is not that straightforward in code. One reason is that formatting such as white space may not necessarily be important to code semantics. Another reason is that identifiers are often a combination of different words, which generates an extremely high-dimensional space of words.

The idea of applying seq2seq learning strategies to APR has the major advantage that the models are language-agnostic, meaning that the same model can be used for learning to repair C# code as Python code. Furthermore, the model is stable to an extent that syntactically incorrect code can be fed into the NN. This is useful when the broken code in the input is supposed to be fixed by the NN.

However, there also exists the approach of utilizing codes' Abstract Syntax Trees (ASTs) for NN input. Using ASTs generally has the advantage that a graph places highly related nodes into near proximity, whereby they might be far away in source code. Furthermore, any programmatic analysis, such as Control Flow Analysis, can be further embedded into a graph using additional edges between nodes. This approach mostly uses specific neural network implementations, such as Graph Neural Networks (GNNs), Graph Gated Sequence Neural Networks (GGNNs) and Graph Attention Networks [8][9][10]. However, in this thesis, neither ASTs are used for the input, nor are any graph-related NN architectures used.

Chapter 3

Related Work

3.1 Neural Automatic Program Repair Approaches

Related to sequence-to-sequence learning is the the work of Chen et al., namely SequenceR [11]. Using LSTMs buggy context (with explicit tokens wrapped around the buggy location) were fed as input and a one-line fix was generated as output. Trained upon real-life GitHub bug-fix commits, it was able to fix 14/75 one-line fixes of the Java bug test-suite benchmark Defects4J. It is especially worth noting that SequenceR did not restrict its input size to the buggy method, but towards the entire parent class, giving the NN more context to understand from. Furthermore, Chen et al. utilized a copy mechanism to avoid the out-of-vocabulary issue of NMT [12].

Gupta et al. architect "DeepFix" - a NN that both localizes compiler errors and fixes them by suggesting one-line changes. A compiler/oracle is used to determine whether the code has been finished fixing and can forward it back to the NN. They thus enable iterative multi-line fixing across entire files. [13]

Building on top of DeepFix is DrRepair by Yasunaga et al. whereby the main difference is that the compiler diagnostic is included in the NN input [14]. This presents a clear similarity to the work done in this thesis, where a message is combined with code to predict a change in code. In contrast to this thesis however, DrRepair feeds every line into a separate LSTM and programmatically creates a graph of hidden LSTM states with every word in the message that corresponds to a token in the code. It is thereby trying to both capture the full meaning of every line (using one LSTM per line), but also establish clear relationships between lines (using the graph attention mechanism), which essentially tries to encounter long-range

dependency problems in sequence-to-sequence NNs. Furthermore, they pre-train their NN with a corpus of self-corrupted files.

Harer et al. approach the fixing of vulnerabilities with a Generative Adversarial Networks (GAN) setup [15]. Having pre-trained the LSTM-generator as an auto-encoder of masked code sequences, they implement a discriminator, which discriminates on a token-basis. Final training is ultimately performed on the SATE IV Juliet Test Suite of artificially introduced vulnerabilities for C/C++ functions [16][17].

With the advent of combining Reinforcement Learning (RL) with neural networks in fields such as gaming (including classical games such as Go), APR has also seen its application in this direction [18]. Whilst Li et al. take first steps in generating paraphrases for natural languages, thus utilizing RL in a sequence-to-sequence learning setting, Gupta et al use the Asynchronous Advantage Actor-Critic (A3C) method to let an actor navigate along code with a cursor (up, down, left, right) and then pass edit actions to the environment as soon as it has arrived at a buggy location [19][20]. The latter strategy is specifically helpful in the field of RL, where minimizing the action space can be of high interest for efficient learning [21]. Somewhat more simplistic is the approach of Barriga et al., which have combined the Bellman Equation in Q-Learning with neural networks [22]. Contrary to Gupta et al, they pre-define a set of very specific actions, such as "Rename class", which may be hard to scale across a larger action space [23]. Further strategies of how to utilize RL in sequence-to-sequence learning is well discussed in the work of Keneshloo et al. [24].

Related to the approach taken in this thesis, is DeepCodeReviewer by Gupta [25], which uses a multi-encoder LSTM to determine whether a code review will lead to a code change given the respective piece of code. Whilst the network was trained on 30.5k code and review pairs from C# GitHub pull requests, the review was parsed separately from the code using the Neural Language Toolkit (NLTK)'s tweet lexer [26]. The end-to-end evaluation pipeline that Gupta proposes, chooses a code review from a repository of code reviews, based on the cosine-similarity of the encoded code nearby. If the review passes a threshold of the NN's classifier, the review is labelled as actionable. In contrast to this thesis, DeepCodeReviewer's NN thereby does not generate any actual code change, but does on the other hand make the implicit decision whether a "diagnostic" *should* generate a change or not. The latter is strongly related to the area of false positives of static analyzers.

Within the scope of leveraging code ASTs, the work of Allamanis [27], Alon [28] and Tarlow [29] is especially worth mentioning. Whilst the task

that Alon et al. applies is related to code summarization rather than program repair, it is unique in the methodology of choosing random paths between end-nodes in the AST, encoding these via LSTMs and then applying an attention mechanism between them. Allamanis et al. go a different route by applying Li et al's GGNNs to ASTs, and describe a classification problem of variable misuse and a variable naming prediction task [9]. Of specific interest is their methodology of adding edges between nodes to foster further information flow through graph propagation. Building on top of [27] is the Graph2Diff approach by Tarlow et al, which again bridges the gap between GNNs and APR. Similarly to this thesis, Graph2Diff implements it's own logic regarding a the representation of diffs, named Graph2Topoco. In contrast to the model used in this thesis, their model works with a combination of pointers in the input graph and copy expressions. The dataset Tarlow et al use, stems from Google's internal build logs including compiler diagnostics, amounting to approximately 500k data points.

3.2 Neural Networks & Static Analysis

Related work of using neural APR with static analysis can generally be divided into two attempted tasks. On the one hand, researchers use analyzers to train NNs to detect vulnerabilities, usually resulting in a classification problem. These works are covered in the Section 3.2.1. On the other hand, researchers have worked with static analysis to train NNs to learn to apply code transformations. Of the two, the latter is more related to the work in this thesis and is described in Section 3.2.2.

3.2.1 Neural Vulnerability Detection

Tanwar et al. trained a neural network to classify whether one of 5 types of Common Weakness Enumeration (CWE) vulnerabilities is present in C/C++ functions, whereby the vulnerabilities in the dataset were previously identified by static analyzers. [30] Even though the NN is both evaluated and manually fine-tuned with a proprietary dataset of historical bugs at Cisco, the fine-tuning still incorporates labelling by the same static analysis tools [31].

Russell et al. similarly combined the dataset SATE IV Juliet Test Suite with another custom dataset of mined GitHub commits, which were labelled by static analyzers. By mapping all static analyzer rules to CWEs, and discarding all data not representing vulnerabilities (e.g. code refactorings),

149 static analysis rules were utilised in the data generation. Their NN-Random Forrest implementation was thereby trained to classify whether any CWE vulnerability was present in a given code function [32].

Zheng et al. takes a unique approach to generate a dataset using static analyzers, which they label "Differential Analysis" [33]. Mining real life commits over multiple versions through git history, they determine whether static analysis warnings have been removed over time, assuming that these have represented real bugs. Most importantly, this dataset also encompasses static analysis warnings that have not been removed and therefore treated as false positives. Once again, the task at hand is to identify vulnerabilities, and not to fix them - this time however addressing the issue of false positives. Unlike [31] and [32], which utilized the C/C++ static analyzers Clang [34], Cppcheck [35] and Flawfinder [36] to label their dataset, Zheng et al. used Facebook's static analyzer Infer [37].

3.2.2 Neural Representation of Program Edits

Highly influential towards this thesis was the work by Yin et al., whereby an auto-encoder NN was built to learn and re-apply code edits [38]. Whilst the input would be both the code before and after the edit, the NN was expected to encode the "edit representation". In an intermediary step, the initial code would again be fed to the NN and the output was expected to be the code after the change. To encounter the NN simply memorizing the code after the edit, the dimensionality of the encoding representation vector was reduced to a minimum. Whilst the NN was trained on a large set of C# GitHub edits, it was evaluated on a dataset of C# static analyzer fixes, whereby one code edit representation (essentially a rule-based static analysis fix) would be applied to a different context. In contrast to their paper, where a fix representation would be determined by the code change itself, this thesis represents the code change purely through the diagnostic message of the static analyzer, making it easier to create a new edit representation. A clear distinction between the datasets used by this thesis and their paper is later described in Section 3.3.

Also using static analysis tools with C# is the work of Qureshi et al.[39]. Comparing the AST-based approach code2vec by Alon et al. with a LSTM and a bag-of-words model, they attempt to classify edits that had previously been generated by 10 different diagnostics stemming from the Roslynator analyzers [40]. In the comparison, the LSTM performs best.

Further related to the scope of code transformations with static analysis is the work of Allamanis [41], named Naturalize, that however generates a static

rule set given the available codebase. Similar to this work, Naturalize focuses on formatting and naming conventions and thereby requires the formatting in the code to be explicitly tokenized. Additionally to creating the "diagnostics", Naturalize will also provide code changes. It is notable that Allamanis et al. also analyze commit and review messages in GitHub's top 10 most popular Java repositories and find that 38% of review messages are related to coding conventions and 9% to formatting, thereby giving the field of formatting changes an empirical weighting in the field of APR.

Building on the work of Allamanis [41] is the tool "Styler" by Loriot et al. Once again, formatting is explicitly tokenized, but the dataset is generated by violations against static analysis Checkstyle rules, which are defined in 2.56% of Java repositories they mine [42]. By only feeding formatting tokens between identifiers and Java keywords to an LSTM, the NN is trained to identify the token which represents the irregularity and can output the same sequence with the fixed token. Instead of applying Checkstyle rules to the corresponding original repositories, the violations are generated by randomly changing only the formatting tokens, which creates the dataset of fixes [43].

Last, but not least, the tool TFix by Berabi et al. has most similarity to our work [44]. It also attempts to fix code given a diagnostic message and uses both an alike textual encoding and the Transformer model to predict code changes. The main difference between our work lies in the dataset generation and the programming language to fix. Berabi et al. utilize ESLint to check whether diagnostic messages are removed between commits on JavaScript repositories on GitHub [45]. Whilst this was considered for our work, the open-source C# tools inspected did not allow for a *quick* single-file analysis without compiling the entire C# project. Furthermore, the given open-source tools in C# already provided fixers to diagnostics. Thereby, their work attempts to learn fixes from human behaviour, whilst we attempt to learn fixes from hard-coded behaviour.

3.3 Datasets for Static Analysis in C#

In relation to this thesis, two types of datasets are relevant. Firstly, any agglomeration or list of C# static analyzers is required and secondly, any dataset which shows identification/fixing of code smell/bugs in C# is of interest. The former appears to be mostly limited to listings on READMEs of GitHub repositories[1] [2]. These help to understand the landscape of open-

[1] https://github.com/analysis-tools-dev/static-analysis/blob/5bc2594291b664d93c6a835238a962d869aae73a/README.md [2] https://github.com/DotNetAnalyzers

source static analysis, but are neither extensive nor particularly detailed, as do they do not give quick insights into any analyzers' assemblies.

Regarding datasets of applied static analysis in C#, resources appear to be limited as well. Most closely related to the "fixer dataset" in this thesis is the one established by Yin et al. [38]. This paper aggregates 2878 data points, in which each consists of both the source and target version of the AST-node that has been modified in a code-, token- and AST-form. Further included is the preceding and the succeeding context of this node.

The prime similarity to our dataset, in fact being the factor that influenced this thesis greatly, is that it also applies C# code fixers to a set of C# GitHub repositories and documents the changes. However, in contrast to this thesis, Yin et al. limit their dataset to 16 diagnostics, disregard the diagnostic messages and strip all formatting. Furthermore, since their dataset is built with Roslyn, the changes are represented on the node and not the line level. However, their dataset also includes AST representations, which this thesis' dataset does not. Lastly, the 54 C# GitHub repositories utilized in this thesis to apply the fixes on, originated from a list described in the paper by Yin et al.

As mentioned in the previous section, the work of Qureshi et al. also use C# static analyzers to generate a dataset of fixes [39]. Even though the dataset contains more than 250 open-source C# repositories, the number of diagnostics is limited to 10, amounting to 12,784 code edits altogether. The dataset is not used in this thesis, since the data points neither contain a diagnostic message nor file context. Nonetheless, Qureshi et al. also argues that finding labelled bug datasets are difficult to find.

Further related C# datasets are the C# Vulnerability Test Suite by Stivalet [46] and the Juliet Test Suite for C# [47], released by the Static Analysis Tool Exposition (SATE) project, whereby the latest SATE V report was headed by Delaitre et al. The two test suites are similar as they both label CWE vulnerabilities in C# code. As other papers have done [32][17], both these datasets can be used to train NNs to identify vulnerabilities. In this thesis paper however, the goal is to learn repair strategies.

Chapter 4

Experiment Design

4.1 Overview

RQ1 evaluates the state-of-the-art of static analysis in C# by quantifying open-source C# analyzers in form of NuGet packages. This is important to understand what can be learnt from them, in addition to opening up drawbacks or advantages that the C# code analysis framework offers. The findings of RQ1 are used in an intermediate step to create a dataset, which optimally describes as many fixing behaviours as possible inherit to the examined static analyzers.

An overarching goal of both RQ2 and RQ3 is to quantify how much a NN can learn from existing static analyzers. The specific aim is to learn repair strategies from static analyzers and not the actual code analysis. This means that the given NN does not decide what about the code is generally wrong, but rather how to change the code given a certain diagnostic message. This can be compared to a code review, where the review has already been done, a comment is added, and now the code has to be changed accordingly.

The difference between RQ2 and RQ3 is that RQ2 attempts to measure how many repair strategies from existing diagnostic messages can be copied by a NN and RQ3 explores how a NN performs on entirely new diagnostic messages. Hence, the experiments are split between "Imitation" for RQ2 and "Extrapolation" and for RQ3. Both of these experiments are performed with the same dataset and only the manner of partitioning between training, validation and test set is changed.

Lastly, findings to RQ4 are a side-product to the experiments made for RQ2 and RQ3. To the best of our knowledge, there is no previous work that has shown repair strategies of formatting and syntax simultaneously, so that the findings made here may be interesting for future work in the area of code

refactorings using neural networks.

The properties and the implementation of the dataset are described in a separate chapter. The same applies for the architecture of the neural network. Furthermore, all code and corresponding datasets created for this thesis can be viewed on https://github.com/olapiv/neural-repair-static-analysis.

4.2 RQ1: Mining Static Analyzers

The collection of static analyzers includes multiple steps. Following the Microsoft Documentation[1], the NuGet REST API is queried for all packages related to the keyword "analyzer". This results in a list of 782 NuGet packages.

When installing all NuGet analyzer packages into one directory using the nuget Command-line Interface (CLI), NuGet packages install their dependencies into the same folder. Many NuGet packages related to the word "analyzer" (further called "analyzer packages") have other analyzer packages as dependencies. Thereby in the following, it is differentiated between "source packages", meaning packages that support/fix their own diagnostics, and "host packages", meaning packages that reference other analyzer packages, but may not support/fix any diagnostics of their own. To understand the extent of these cross-dependencies between the NuGets, every installed NuGet directory is analyzed recursively and checked whether it has any dependencies related to the initial list of analyzer packages. This generates a dependency tree which is saved to a JSON file.

An example of the structure in the JSON file is shown in Listing 4.1. Since the host packages usually do not reference the newest version of the source packages and therefore multiple versions of source packages are regularly installed, the structure in the listing gives a good insight to which host packages install the older source package versions.

After all packages are installed and dependencies are examined, C# Reflection is used to extract all data related to static analysis from the corresponding assemblies. The output is saved to a csv file. Since one NuGet package can reference an outdated version of another analyzer package, the csv file is initially prone to contain duplicate data from different NuGet versions. These are however eliminated in the analysis.

Both the csv file and the JSON file are quantified using Python's Pandas library, whereby the results are shown in Section 7.1 [48]. These calculations

[1] NuGet API Documentation: https://docs.microsoft.com/en-us/nuget/api/search-query-service-resource

Listing 4.1: Examples of Analyzer Dependencies

```
"Infeeny.Dotnet.Analyzers__2.0.0": {
    "codecracker.CSharp__1.1.0": {},
    "SonarAnalyzer.CSharp__8.7.0.17535": {},
    "StyleCop.Analyzers__1.1.118": {},
    "ConfigureAwaitChecker.Analyzer__3.0.0": {}
}

"Ustilz.Analyzers__1.0.0": {
    "Microsoft.CodeAnalysis.FxCopAnalyzers__2.9.8": {
        "Microsoft.CodeQuality.Analyzers__2.9.8": {},
        "Microsoft.NetCore.Analyzers__2.9.8": {},
        "Microsoft.NetFramework.Analyzers__2.9.8": {},
        "Microsoft.CodeAnalysis.VersionCheckAnalyzer__2.9.8":
            {}
    },
    "StyleCop.Analyzers__1.1.118": {},
    "Microsoft.CodeAnalysis.PublicApiAnalyzers__2.9.8": {},
    "Text.Analyzers__2.6.4": {}
},
```

are used to answer RQ1.

4.3 RQ2: Imitating Static Analyzers

This experiment, conveniently named "Imitation", trains a seq2seq Transformer model, meaning that input and output are sequences of tokens. The dataset to train the model comprises of C# static analysis fixes from different diagnostics in various file contexts. The architecture of the network is described in detail in Chapter 5.

The experiments is divided between two datasets, which differ in tokenization method and are named "standard" and "camelcase" tokenization. There also exist minor differences in dataset size. The different tokenization methods are a measure of optimization and the motivation behind this is once more explained in Chapter 6. However, the evaluation of the network therefore also includes comparisons between these two methods/datasets.

To accompany neural learning, the dataset is randomly split up between training, validation and test dataset. The neural network is trained on the training dataset and is evaluated on the validation dataset every n steps. This

is done in order to observe whether the dataset is overfitting on the training set. Overfitting defines the point where training may still decrease loss, but the evaluation loss on unseen data is increasing. As final evaluation, the test dataset is introduced. This is the dataset that the outcomes in the results section are based on.

The proportions of how the dataset is split are 60/20/20 for train, validation and testing respectively. All existing data points are randomly mixed, so that this experiment will only have a few diagnostics that have not been provided in the training set, thus providing a good measure of whether a NN can learn to imitate observed behaviour.

The method of evaluation in the results section is to first count how many predictions have been identical to the output in the test dataset, using the corresponding input. It is important to note that "success rate", "accuracy" and "perc_correct" are used interchangeably and are equivalent to this definition. They show how many entire sequences/data points were predicted correctly in the test dataset and are not equal to the loss of the NN. Further evaluations are then run to calculate the success rates of single diagnostics, and thus an average success rate normalized over all diagnostics is measured. The normalized success rate and total success rate would achieve the same result of x% if for example each diagnostic only had one data point and x% data points were predicted correctly.

The Pearson two-tailed p-value is applied to any data for which a causality between the variables is not clear. A Pearson p-value threshold of 10% is chosen to determine whether the data is random or not. If it is greater than 10%, it is defined random.

This experiment is run with a Nvidia GeForce GTX 1080 and 64GB RAM, provided by Digital Illusions Creative Entertainment (DICE). To show whether the NNs converge, the loss curves are shown for all steps that are run. As described later in Section 5.4, the network is trained until Early Stopping is activated. The weights used for evaluation are the ones reached at the end of the loss curve.

4.4 RQ3: Extrapolating from Static Analyzers

This experiment, named "Extrapolation", follows a similar methodology to RQ2 and uses the same neural network architecture. The difference of methodology lies within the dataset.

Instead of splitting the data points proportionately between the train, validation and test dataset as in Imitation, this is now done with the diagnostic

ids. This means that we divide diagnostic ids into train, validation and testing dataset, so that we ensure that different sets have mutually exclusive diagnostic ids. For a more conservative estimation of extrapolation, identical diagnostic ids from different NuGet packages are assumed identical diagnostics. This avoids observing successful extrapolation when the diagnostic has in fact already been seen in training. The proportion to split the dataset is 70/20/10 for train, validation and testing respectively. This therefore guarantees the opposite of Imitation - specific diagnostics will not have been seen in training, so the network can be evaluated on how well it performances on entirely new diagnostic messages. In contrast to RQ2, this experiment is run with a Nvidia GeForce RTX 2080 Ti, provided by the ASSERT research lab at KTH ("Tiramisu").

4.5 RQ4: Measuring the Impact of Formatting Tokens on Neural Repair

The datasets for the experiments described for RQ2 and RQ3 include refactorings, which are often changes in formatting. As later described in Chapter 6, formatting is therefore included as explicit tokens. To learn to apply these changes, the neural network must understand the concept of formatting as well. This research question measures to what extent including formatting inhibits or even fosters neural learning. To answer the question, the same results as for RQ2 and RQ3 are taken and inspected from the perspective of formatting. Specifically, the best results of either Imitation / Extrapolation will be used here.

This experiment will examine how the percentage of formatting tokens influences the success rates of the neural network. The percentage of formatting tokens is chosen since the raw number of formatting tokens is highly related to the raw number of total tokens. As both long input and output sequences generally represent large amounts of data, making them difficult to understand or predict for a neural network, high numbers total tokens will by default perform worse. Thereby, raw numbers of formatting tokens are not suitable for this analysis.

Chapter 5

Neural Network

The goal is to train a neural network to predict the edit representation of one or more diagnostic messages given a code snippet and the location of the diagnostics in the code. It is important to note that it should not develop its own understanding of "good"/"bad" code, but rather output changes that fit the surrounding file context and satisfy the fixing of the diagnostics. In the following, the model used to tackle this task, its input and output representation, the framework used to implement it and the corresponding configuration is described.

5.1 Model

The neural network that is chosen is a sequence-to-sequence Transformer model, provided by OpenNMT [7][49]. The following section explains the reasoning behind this model and why a textual sequence encoder was specifically chosen over an AST-encoder.

Since the goal for RQ2 and RQ3 is to learn repair strategies from fixers *given a diagnostic message*, some sequential input encoder is required at any point. However, when analyzing the fixes done by analyzers, it is found that many fixers actually implement code refactorings, whereby code formatting plays a crucial role. Table 5.1 shows an example of such a fixable diagnostic. Whilst including formatting in textual tokens is straightforward, it is not that simple when using Roslyn to generate the AST. There, it is labelled "Trivia" and is difficult to separate from actual tokens, as each span of Trivia is assigned

to a token.[1] Although Yin et al. provide code[2] to serialize source code changes in C# into AST representations, it strips all formatting and performs further transformations which do not suit the dataset tokenization properties described in Section 6.4 [38].

Table 5.1: Example Code Diagnostic to refactor Code Formatting

NuGet Package	StyleCop.Analyzers.1.1.118
Diagnostic ID	SA1516
Diagnostic Title	Elements should be separated by blank line
Diagnostic Description	Adjacent C# elements are not separated by a blank line.
Default Severity	Warning

As mentioned previously in Section 3.1 with related work from Allamanis [27], a strong argument for using ASTs is adding further edges to the graph originating from other program analysis methods. Since in this thesis the analysis has already been done, culminating in the diagnostic message, it does not appear necessary to generate the AST once more. To our knowledge, none of the fixers applied any analysis themselves, but relied on this having been done previously by the analyzers.

Lastly and more trivially, as to the best of our knowledge, no other paper has attempted to use static analyzer messages to fix code with neural networks (in contrast to compiler diagnostics). The aim is thereby not to score higher than any benchmark using a particularly complicated AST-based NN. In the contrary - the goal is to provide a simple, but effective proof-of-concept. Nonetheless, the Transformer model is chosen over conventional LSTMs due to its successes in learning an order of magnitude quicker for translating natural languages [7].

5.2 Input/Output Representation

The input consists of a code snippet, one or more diagnostic messages and the diagnostic locations within the code snippet. The output consists of a diff, which should be applied to the code to remove the diagnostics. To

[1] Documentation to the class is here: `https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.syntaxtrivia`

[2] Code by Yin et al.: `https://github.com/microsoft/msrc-dpu-learning-to-represent-edits`

obtain an understanding of the representation, an example data point is shown. Whilst Listing 5.1 shows an exemplary input for the NN, Listing 5.2 being its readable counterpart, Listing 5.3 shows the corresponding output with Listing 5.4 being the readable version. The terms "code snippet", "file context" and "file content" are used equivalently. The following sections describe how the input and output is structured.

Listing 5.1: Example of tokenized Input

```
LINE 2 MESSAGE convert WHITESPACE static WHITESPACE method
   WHITESPACE call WHITESPACE to WHITESPACE extension
   WHITESPACE method WHITESPACE call FILE_CONTENT " ; NEWLINE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE literalType WHITESPACE += WHITESPACE
   string . Join ( " , WHITESPACE " , WHITESPACE keyValues .
   ToArray ( ) ) ; NEWLINE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   DbLinqMemberInfoExtensions . SetMemberValue ( memberInfo ,
   WHITESPACE owner , WHITESPACE literalType ) ; NEWLINE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE } NEWLINE NEWLINE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE internal WHITESPACE
   EnumType ( object WHITESPACE owner , WHITESPACE MemberInfo
   WHITESPACE memberInfo ) NEWLINE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE
```

Listing 5.2: Humanly readable Input for 5.1

```
"parsed_src": {
    "diagnostic_occurances": [
        {
            "diagnostic_line": "2",
            "diagnostic_message": "convert static method call
                to extension method call"
        }
    ],
    "file_context": [
        "\";",
        "            literalType += string.Join(\", \",
            keyValues.ToArray());",
```

```
        "            DbLinqMemberInfoExtensions.SetMemberValue(
            memberInfo, owner, literalType);",
        "        }",
        "",
        "        internal EnumType(object owner, MemberInfo
            memberInfo)",
        "    "
    ]
}
```

Listing 5.3: Tokenized Output for 5.1

```
REPLACE SOURCE_LOCATION 2 TARGET_LINES WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   WHITESPACE WHITESPACE WHITESPACE WHITESPACE WHITESPACE
   memberInfo . SetMemberValue ( owner , WHITESPACE literalType
    ) ; NEWLINE
```

Listing 5.4: Humanly readable Output for 5.3

```
"parsed_tgt": {
    "diff_type": "REPLACE",
    "source_location": [
        "2"
    ],
    "target_lines": [
        "            memberInfo.SetMemberValue(owner,
            literalType);"
    ]
}
```

5.2.1 Input

The input to the neural network is the file context, diagnostic messages and the diagnostic locations. The file context is the source code we want to change, the diagnostic message(s) describe what should be changed in the code and the diagnostic location point to a line in the code where this change should be applied to.

The file context is constructed by using two components: firstly, all lines that will change, which we call the "required lines", and secondly, tokens around the required lines until a defined limit of total tokens is reached.

Listing 5.5 exemplifies this strategy. This is done in order to understand the environment in which the diagnostic messages are placed. Data points that contain a "too many" tokens in the required lines will be excluded by this technique. As later described in Section 6.4.3, one dataset applies camelcase tokenization, whilst the other preserves identifier names. For the latter case, the number of file context tokens is defined as 100, whilst for the dataset with camelcase tokenization 115 are defined.

Listing 5.5: Required Lines in Orange and Additional Tokens in Yellow

```
    return _serverTask.Value;
  }
}

<<<< DIAGNOSTIC: remove async suffix from method name >>>>
public ValueTask DisposeAsync() => StopAsync();

private async Task StartCore()
{
  _listener
```

Regarding the diagnostics in the input, each diagnostic message available, including its location in the file context, is prepended to the file context in the input, similarly to the design of DrRepair for compiler diagnostics [14]. In contrast to DrRepair, this model allows attention to be applied between any word in the diagnostic and any word in the code. DrRepair represents a regularized version of this NN, since it only allows this for keywords / identifiers, which appear both in code and diagnostic. Even though this is an interesting approach, it is not shown that it is more effective.

5.2.2 Output

The output can be one of three "diff actions" ADD, REMOVE, REPLACE. This is done by examining the git diffs resulting from fixes and only extracting the diff hunks (consecutive lines in a diff). All code that does not represent a change is removed. Repeating code context in the output is not necessary if the diff action also defines at what line it should be inserted. Whilst Listings 5.6 and 5.7 show basic code changes of adding and removing lines respectively, Listing 5.8 shows an example where a deletion is followed by an addition, therefore being a replacement.

This technique is used because longer sequences in output increase the number of possible predictions and they become more difficult to predict.

Therefore the output is constructed so that a minimal amount of tokens are required to be used. Even though the dataset generated in this thesis only includes single diff actions, thereby coming short of changes to non-consecutive lines, this technique also allows multiple diff actions to be appended to each other.

Listing 5.6: Example Diff Action ADD

```
"ActionType": "ADD",
"Action": {
    "PreviousSourceLocation": 6,
    "TargetLines": [
        "\n"
    ]
}
```

Listing 5.7: Example Diff Action REMOVE

```
"ActionType": "REMOVE",
"Action": {
    "SourceLocationStart": 7,
    "SourceLocationEnd": 8
}
```

Listing 5.8: Example Diff Action REPLACE

```
"ActionType": "REPLACE",
"Action": {
    "SourceLocations": [
        3
    ],
    "TargetLines": [
        "        System.DateTimeOffset System.IConvertible.
            ToDateTime(System.IFormatProvider? provider) { throw
             null; }\n"
    ]
}
```

5.3 Framework

The framework used to apply the Transformer model is OpenNMT, which calls itself an "ecosystem for neural machine translation and neural sequence

learning" [49]. It supports multiple models for learning NMT, of which the Transformer model by Vaswani et al. is the default one [7]. It is a project that has two implementations, one in PyTorch and one in Tensorflow [50][51]. These are both also Python frameworks, which are built to simplify not only the creation of NNs, but also their debugging and evaluation. Due to initial considerations of using a multi-encoder for diagnostic and code and having this only implemented in the Tensorflow version, this thesis uses the Tensorflow version. The multi-encoder is however not applied to be sure that the self-attention mechanism is more effective between common words in the diagnostic message and code.

5.4 OpenNMT Configuration

The default OpenNMT-tf configuration settings for the Transformer model can be found in the documentation. The settings that are overwritten are shown in Table 5.2. As a standard procedure for NN learning, the trained NN is regularly benchmarked using the validation dataset. Here, this is done every 5,000 steps and the benchmark is the loss of the results. Early Stopping describes the mechanism in which the NN stops training if the benchmark has not improved for the last defined number of validation runs. The maximum amount of input/output tokens is limited to avoid outliers.

Table 5.2: Neural Network Configuration

Max Tokens Input	200
Max Tokens Output	150
Share Input/Output Vocab	True
Validation Steps	5000
Early Stopping Steps	4
Early Stopping Minimum Improvement	0.01

Chapter 6

Dataset

6.1 Overview

The following chapter highlights different aspects of the dataset that was generated for the experiments in RQ2, RQ3 and RQ4. It describes the underlying source of the dataset, the implementation details of mining and the methodology of fine-tuning it. The dataset consists of a set of fixes with corresponding diagnostic messages. Since the format of the dataset, described in detail in Section 5.2, are sequences of tokens, the fine-tuning in this chapter concentrates on the tokenization process. Lastly, this chapter also gives insights to the metadata of the final dataset.

6.2 Creating Raw Dataset

A major tool utilized in creating the "raw" dataset is Roslynator.[1] Roslynator is a C# repository, which amongst others, contains analyzers, fixers and most importantly, a CLI-tool to apply analyzers and fixers to C# projects or solutions. As the name suggests, the CLI-tool is powered by Roslyn. The two commands of interest, *analyze* and *fix*, allow the user to specify the analyzer package, plus optionally a specific diagnostic. Whilst *analyze* will return diagnostics with locations[2], *fix* will apply a fix, therefore generating a diff in the project/solution to be fixed. These two elements are what the raw dataset consists of.

[1] https://github.com/JosefPihrt/Roslynator

[2] A sample of this analysis is stored here: https://github.com/olapiv/neural-repair-static-analysis/blob/main/sample_roslynator_analysis.xml

The initial set of repositories to which the analyzers and fixers are applied to stem from the work of Yin et al., which used these C# repositories to mine GitHub edits across commits and chose the repositories based on popularity [38]. However, due to time reasons, only a random subset of 28 repositories is selected for these experiments, which is displayed in Table 6.1.

Table 6.1: C# GitHub Repositories

CodeHubApp/CodeHub
shadowsocks/shadowsocks-windows
dotnet/runtime
PowerShell/PowerShell
dotnet/roslyn
mxgmn/WaveFunctionCollapse
StackExchange/Dapper
NancyFx/Nancy
SignalR/SignalR
Wox-launcher/Wox
ShareX/ShareX
mono/mono
AutoMapper/AutoMapper
restsharp/RestSharp
hbons/SparkleShare
Microsoft/BotBuilder
OpenRA/OpenRA
JamesNK/Newtonsoft.Json
ServiceStack/ServiceStack
MahApps/MahApps.Metro
MonoGame/MonoGame
icsharpcode/ILSpy
MaterialDesignInXAML/MaterialDesignInXamlToolkit
reactiveui/ReactiveUI
Microsoft/msbuild
Microsoft/VFSForGit
chocolatey/choco
aspnetboilerplate/aspnetboilerplate

Listing 6.1 shows a simplified abstraction of the code, in which Roslynator is applied iteratively to the list of repositories. Since the script is executed on a 10-core CPU machine from DICE, it is parallelized across 10 repositories

Listing 6.1: Generating the Raw Dataset

```
for repo in [28]repos:
    for solution in repo.c_sharp_solutions:
        for analyzer in [442]nuget_analyzers:
            # Creates xml file:
            roslynator.analyze(solution, analyzer)
            for diagnostic in analyzer:
                # Creates diff:
                roslynator.fix(solution, analyzer, diagnostic)
                save_diff_to_file()
                # Discard changes
                git restore
```

using Powershell 7 and run for 4.5 days. Due to the multitude of different C# SDKs the different projects are targeted towards, additional to encountering various compiler errors, debugging this script takes a considerable time and presents a large workload within this thesis. Instead of using Roslynator's NuGet CLI tool, Roslynator is also built locally, since errors can occur when the project to analyze is not built with the same MSBuild version as Roslynator is built with itself. Furthermore, the script is specifically run with Powershell, since the cross-platform .Net Framework implementation Mono does not work as reliably in these experiments as the standard .Net Framework SDKs on Windows.

6.3 Merging Diagnostics with Diffs

Since the raw dataset consists of separate analysis files and diff files, it is necessary to merge them into single data points. As mentioned previously, one diff stems from fixes of a single type of diagnostic - however, this diagnostic may have appeared multiple times and may therefore have triggered multiple changes. The assumption taken here is that a diagnostic in one file will not only lead to a change in the same file, but also to one in the immediate line proximity of the diagnostic location. As the output representation of the NN splits diffs into diff actions of ADD, REPLACE and REMOVE, the challenge is to map diagnostic locations to these diff actions.

A shortcoming of this process is that changes which are in non-consecutive lines are not treated as a single diff action and are therefore split into different data points. Furthermore, in some cases, multiple diagnostic occurrences may

have generated one diff action. An example would be two occurrences in the same line, but at different characters. If the line was deleted in the diff, one cannot tell which diagnostic occurrences caused this to happen. Therefore, these occurrences are bundled.

To generate a more heterogeneous dataset, a maximum of 3 data points are generated per patch, whereby one patch is a subset of a diff and represents all changes within one file. This is because some files may be very large and contain repetitive structures (such as Enum listings) and so specific analyzers will generate many identical diagnostics for similar contexts.

6.4 Tokenization

The tokenization of the NN's dataset naturally takes a major influence on what the NN will learn. Generally, large and heterogeneous datasets are best suited to train NNs as they make it difficult for the NN to overfit to specific use-cases. Since learning from a fixed amount of static analyzer diagnostics already represents a restraint in this direction, the aim is to at least encompass as many different diagnostics in the dataset as possible. This is a major design goal and has large implications on the dataset's tokenization. The following sections describe specific properties of the applied tokenization. To support these features a custom regex lexer using the Python library Pygments is used [52].

6.4.1 Formatting

First and foremost, both input and output representations include explicit tokenization of formatting in the form of WHITESPACE, TAB, NEWLINE. As mentioned previously, a number of code fixers implement code refactorings, whereby a change may only be visible if formatting is available. Furthermore, by particularly using NEWLINE, multi-line input file context and multi-line output diffs can be represented.

6.4.2 Comments, String Literals and Documentation

Related to the property of tokenizing formatting, this model includes full comments, documentation and string literals in input and output. Since a fraction of diagnostics point towards missing documentation, such as the ones from NuGet package Documentation.Analyser[1] or towards using string

[1] https://www.nuget.org/packages/Documentation.Analyser/

interpolation[1] and the fixes thereby operate in this scope, natural language segments are tokenized as well as code. Once more the goal is to encompass as many different fixes as possible.

6.4.3 Identifiers

Since a multitude of fixers apply changes to identifiers, the diagnostic in Table 2.1 being an example, they play an important role in tokenization. Identifiers are often a combination of English words and therefore represent a larger space than the English language itself. Therefore, it becomes difficult to generate a dataset where identifier names appear multiple times. This also makes it more difficult for a NN to understand identifier names as it may have not seen them before. Therefore, the tokenization is enhanced by splitting identifier names by camelcase, and thereby halving the number of vocabulary words in the dataset. Identifiers that are different, but contain identical phrases are thereby given a semantic relation. An example of splitting an identifier by camelcase is separating `EclipseRCPExt` to `Eclipse`, `RCP` and `Ext`. The idea is that the NN can thereby understand more of the code environment and also predict changes in identifier names more easily. As this feature is experimental, a dataset with camelcase-splitting is generated and one is generated without. The latter is named "standard tokenization".

6.5 Dataset Statistics

This section gives a quantification and setup of the final dataset. Following the tokenization, this dataset has been lastly filtered on the following properties:

- ASCII-only
- Maximum of 500 input/output tokens
- No duplications

Statistics regarding the final distribution of diagnostic types and tokens in input/output are listed in Table 6.2. The total number of data points is small in relation to other datasets meant for neural learning. This means that learning can converge quickly, but also that it may not be as effective. The final composition of the two resulting **test** datasets, on which the neural network is

[1] CC0048 from NuGet package codecracker.CSharp.1.1.0

evaluated on for RQ2 and RQ3, can be seen in Table 6.3. All dataset variants that the NN is trained, validated and tested on are uploaded to the GitHub code repository of this thesis.

Table 6.2: Dataset Statistics

	Standard Tokenization	**Split by Camelcase**
Data points	24,435	24,622
Diagnostics	421	421
Repositories	18	18
NuGets	113	113
Input		
Max tokens	347	362
Avg tokens	121.7	137.5
Std of tokens	13.2	14.3
Output		
Max tokens	473	487
Avg tokens	41.0	43.5
Std of tokens	37.0	38.4

Table 6.3: Test Dataset Metadata for Standard Tokenization

	Imitation	**Extrapolation**
Data Points		
Total	4887	1217
Diagnostic in Train	4856	0
Diagnostic not in Train	31	1217
Diagnostics		
Total	324	41
In Train	301	0
Not in Train	23	41

Chapter 7

Results and Analysis

7.1 RQ1: State of the Art open-source C# Static Analysis

Of the initial list of 782 analyzer packages which were installed, 456 of these turned out to be source packages, 44 to be host packages and the rest was unrelated to static analysis. The list of all source packages is uploaded to the GitHub repository[1] and exemplary host packages are shown in Table A.1. Interestingly, the host packages also referenced 8 source packages, which were not in the initial list of analyzer packages and are shown in Table 7.1. This was partially due to the usage of the word "analysis" instead of "analyzer", which the packages were queried for, or the fact that these were System.Runtime packages and were therefore not available on NuGet.org. It furthermore turned out that it was a small subset of 36 source packages, which was regularly referenced by host packages. Led by StyleCop.Analyzers, the most important ones are listed in Table 7.2.

When creating own analyzers and fixers with the C# environment, the diagnostic ids can be chosen freely. Therefore, duplicate ids exist across analyzer packages to a great extent. After having filtered out all old versions of source packages, there still existed 889 duplicate supported diagnostics and 490 duplicate fixable diagnostics. As is visible in the two examples in Table 7.3, these duplicates arose for different reasons, which were hard to quantify. Since the NuGet package "LazyMixin" is likely to be an updated version of "LazyMixinAnalyzer", the diagnostic "AccessViaProperty" is likely

[1] List of source packages: https://github.com/olapiv/neural-repair-static-analysis/blob/main/nuget_packages_relevant_sources.txt

Table 7.1: Missed NuGet Packages with Analyzers

NuGet Package	**Reasoning**
Desktop.Analyzers	*Unknown*
MADE.Analyzers	*Unknown*
Microsoft.AnalyzerPowerPack	*Unknown*
Microsoft.CodeAnalysis.CSharp	Word "analyzer" missing
SmartAnalyzers.ExceptionAnalyzer	*Unknown*
System.Runtime.Analyzers	System packages are not on NuGet.org
System.Runtime.InteropServices.Analyzers	System packages are not on NuGet.org
TestableFileSystem.Interfaces	Word "analyzer" missing

Table 7.2: Source Packages with more than two References

NuGet Analyzer Package	**References**
StyleCop.Analyzers	24
Microsoft.CodeQuality.Analyzers	18
Microsoft.NetCore.Analyzers	18
Microsoft.CodeAnalysis.Analyzers	16
Microsoft.NetFramework.Analyzers	17
Microsoft.CodeAnalysis.VersionCheckAnalyzer	13
SonarAnalyzer.CSharp	4
Text.Analyzers	4
codecracker.CSharp	4
Microsoft.VisualStudio.Threading.Analyzers	4
RefactoringEssentials	3
Roslynator.Analyzers	3

to be identical between them. However, it seems more unlikely that CA1052 is identical between the NuGet packages Gu.Analyzers and Microsoft.Analyzer.PowerPack. If all duplicates would be identical, the average diagnostic were supported by 1.17 and fixable by 1.25 different NuGet packages.

There are in total 6213 unique supported diagnostic ids and 2235 unique fixable diagnostic ids. However, when assuming all diagnostics with identical diagnostic ids, but different NuGet packages are unique, there are altogether 7192 supported diagnostics and 2728 fixable diagnostics. Furthermore, there are in total 232 refactorings.

Table 7.3: Sample of Duplicate Diagnostics

Diagnostic ID	NuGet Analyzer Package
AccessViaProperty	LazyMixin.2.6.1
AccessViaProperty	LazyMixinAnalyzer.2.3.0.0
CA1052	Gu.Analyzers.1.8.5
CA1052	Microsoft.AnalyzerPowerPack.1.1.0
CA1052	Microsoft.CodeAnalysis.NetAnalyzers.5.0.3
CA1052	Microsoft.CodeQuality.Analyzers.3.3.2

Relating the supported and fixable diagnostics towards each other, assuming that a diagnostic is both supported and fixed in the same NuGet package, the union of diagnostics is 7214 and the intersection is 2540. The latter number is specifically of interest, since it suggests how often a code-base can be analyzed/fixed when creating a fixer dataset. However, some identical diagnostics from different NuGets may nonetheless be semantically equivalent, and so grouping by diagnostic ids leaves 2092 unique diagnostics. This can be seen as a conservative benchmark to describe the number of different diagnostics in a fixer dataset and is also the technique used to describe the dataset in this thesis.

Continuing on unions and intersections, there are 4548 diagnostics, which are supported, but not fixable and 126 diagnostics, which are fixable, but not supported. The latter displays an odd case, since the diagnostic title and description is also only available in the analyzer and not the fixer. Therefore, it is not known, what use-case these diagnostics actually fix. Nonetheless, the two examples in Table A.2 show again that it is not a clear assumption that the same NuGet supports and fixes a diagnostic as both CS1570 and CS1105 (only) have a fixer in another NuGet.

The average and median fixer/analyzer numbers per NuGet are shown in Table 7.4. The discrepancy between average and median numbers shows that few NuGets contain a large fraction of supported/fixable diagnostics and that the rest of the landscape is strongly fragmented between the other NuGets. The average number of refactorings is relatively high, since it only takes into account NuGets that contained code refactorings in the first place.

Lastly, the average configurations for FixAllProviders regarding fixable diagnostics is shown in Table 7.5. The scope "solution" is important, since it was the scope upon which Roslynator was applied to the open-source C# repositories. Diagnostics that did not support this scope could not be used for

Table 7.4: Numbers per NuGet

	Average	**Median**
Supported and Fixable Diagnostics	21.8	4
Only Supported Diagnostics	15.8	2
Only Fixable Diagnostics	8.9	2
Refactorings	23.2	2

the dataset.

Table 7.5: Average FixAllProvider Configurations for fixable Diagnostics

FixAllProvider defined	0.834
Solution Scope supported	0.797
All Scopes supported	0.795

The full dataset of mined diagnostics that can be supported and fixed is uploaded to the code repository of this thesis[1]. The other dataset documenting dependencies amongst analyzers can be found in the GitHub repository [2].

When querying Nuget.org for "analyzer", 456 NuGet packages contained source code with C# code analysis. Assuming that diagnostics are both supported and fixed in the same NuGet package, the intersection of supported and fixable diagnostics is 2540. This number is significant for the creation of any dataset of fixes. Due to non-unique diagnostic identifiers, it is however not guaranteed that there are no semantic overlaps between identical diagnostics across NuGets. The unique number of diagnostic ids within the 2540 diagnostics is 2092. This is conservative measure of how many diagnostic ids a fixer dataset can possibly have.

[1] https://github.com/olapiv/neural-repair-static-analysis/blob/main/analyzer_package_details.csv

[2] Dependencies amongst analyzers: https://github.com/olapiv/neural-repair-static-analysis/blob/main/nuget_deps.json

7.2 RQ2: Imitating Static Analysis Fixers

Convergence for training loss was reached after around 3k steps, as seen in Figure 7.1. The dataset with camelcase tokenization, having 14,774 training points that span 396 diagnostics, achieves its best validation results after 10k steps (6h50min), leaving 20k steps to run before Early Stopping is activated. The results for validation loss are depicted in Figure 7.2. Standard tokenization on the other hand, containing 14,661 training points with 378 diagnostics, has its best validation results at 5k steps (3h25min) and therefore stops training at 25k steps. As described in 5.4, the validation loss is only labelled "better", if it improves by a certain threshold.

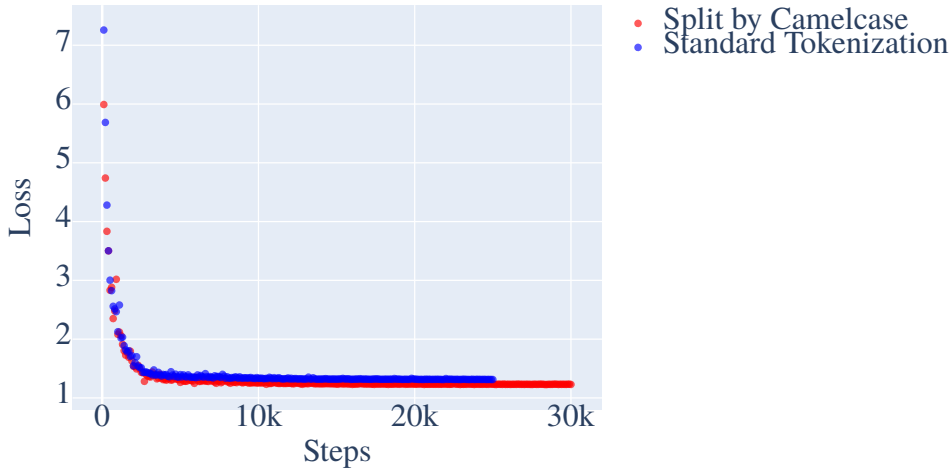

Figure 7.1: Training Loss for Imitation

From the beginning to the end of both training and validation loss, camelcase tokenization performs better than standard tokenization, suggesting that a given fraction of diagnostics perform better with camelcase tokenization. These results are undermined by the results on the test dataset in Table 7.6. Normalized by the number of diagnostics, the average success rate is 4.3% higher for the dataset split by camelcase, which is similar to the amount of diagnostics related identifier changes. Generally, since only less than 31 and 16 data points have diagnostics which are not in the training dataset, the extrapolated results are assumed negligible.

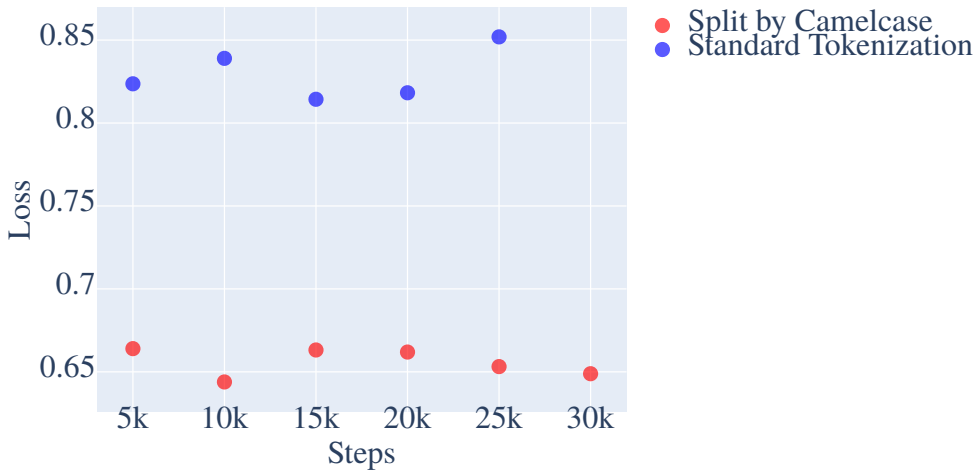

Figure 7.2: Validation Loss for Imitation

Table 7.6: Percentage Results of Test Dataset

	Standard Tokenization	**Split by Camelcase**
Success Rate		
All Data Points	47.5% (/4887)	**55.4% (/4923)**
Diagnostic in Train	47.6% (/4856)	55.5% (/4907)
Diagnostic not in Train	19.6% (/31)	12.5% (/16)
Normalized Success Rate by		
Diagnostic	42.0% (324 diagnostics)	**46.3% (299 diagnostics)**
Diagnostic in Train	43.8% (301 diagnostics)	48.0% (284 diagnostics)
Diagnostic not in Train	17.4% (23 diagnostics)	13.3% (15 diagnostics)

Since this experiment is primarily measuring the extent the NN can imitate, Figures 7.1, 7.2 and 7.4 show samples of diagnostics, which are also included in the training set. Furthermore, since the experiment using camelcase tokenization performed better as well, the examples are also restricted towards these data points.

Listing 7.1: Correctly Predicted Example with Camelcase Tokenization of Diagnostic *SA1513* with 132 Data Points in Train and 100% Accuracy in Test

```
 }
          scopeExpression.Where[whereIndex] = whereClausole;
<<<< DIAGNOSTIC: closing brace should be followed by blank line
    >>>>
      }
+
      if (scopeExpression.Where.Count > 0)
      {
```

Listing 7.2: Example of Correct Solution with Camelcase Tokenization of Diagnostic *SA1307* with 6 Data Points in Train and 50% Accuracy in Test

```
   tree; allowing access by visible index.
        partial class SharpTreeNode
        {
                /// <summary>The parent in the flat list</
                    summary>
<<<< DIAGNOSTIC: field 'listParent' should begin with upper-
   case letter >>>>
-               internal SharpTreeNode listParent;
+               internal SharpTreeNode ListParent;
                /// <summary>Left/right nodes in the flat list
                    </summary>
                SharpTreeNode left, right;

                internal TreeFlattener treeFlattener;
```

Listing 7.3: Incorrect Prediction for Data Point shown in Figure 7.2

```
   tree; allowing access by visible index.
        partial class SharpTreeNode
        {
                /// <summary>The parent in the flat list</
                    summary>
<<<< DIAGNOSTIC: field 'listParent' should begin with upper-
   case letter >>>>
-               internal SharpTreeNode listParent;
```

```
+               internal SharpTreeNode Parent;
                /// <summary>Left/right nodes in the flat list
                    </summary>
                SharpTreeNode left, right;

                internal TreeFlattener treeFlattener;
```

Listing 7.4: Example of Correct Solution with Camelcase Tokenization of Diagnostic *SA1502* with 20 Data Points in Train and 0% Accuracy in Test

```
      [Obsolete("Delete Wox.Plugin.Features using directive, "
         +
                "and use Wox.Plugin.Feature.IContextMenu
                   instead, " +
                "this method will be removed in v1.4.0")]
<<<< DIAGNOSTIC: element should not be on a single line >>>>
-     public interface IContextMenu { }
+     public interface IContextMenu
+     {
+     }
  }
```

Listing 7.5: Incorrect Prediction for Data Point shown in Figure 7.4

```
      [Obsolete("Delete Wox.Plugin.Features using directive, "
         +
                "and use Wox.Plugin.Feature.IContextMenu
                   instead, " +
                "this method will be removed in v1.4.0")]
<<<< DIAGNOSTIC: element should not be on a single line >>>>
-     public interface IContextMenu { }
+         public interface IContext
+         {
+             get { return "km"; }
+         }
  }
```

The code snippets show various mechanisms the neural network has successfully and unsuccessfully learnt. Whilst Listing 7.1 indicates that the network can understand to apply basic formatting changes with high accuracy, Listing 7.2 shows an use-case where an identifier change has been applied

incorrectly. Since the neural network still technically outputs a correct result in the latter case, the result can nonetheless be rated positively. A lot more data would have been necessary for the neural network to map every lowercase word with its uppercase version. A potential fix for this mechanism could be character tokenization, which runs the risk of creating too large source and target lengths. Lastly, the fix in Listing 7.4 shows some understanding of the diagnostic message, but a clear tendency to overfit. The suggested fix appears entirely unrelated to the targeted interface or the context around it. This could point towards a lack of diversity for the given diagnostic in the training dataset.

To obtain an idea whether the amount of data points per diagnostic impacted the success rate for the particular diagnostic, Figure 7.3 is depicted. Each point on the plot represents a diagnostic. However, there appears to be no clear correlation and the Pearson p-value is 0.254, signifying that the data is random by the definition given in the experiment design. This would suggest that generating many data points for single diagnostics may not be effective in improving the success rate of the neural network's ability to copy. To understand this result better, an example of high accuracy with little training data in Figure 7.6 and an example of low accuracy with significant training data in Figure 7.7 is shown.

Listing 7.6: Example of Correct Solution with Camelcase Tokenization of Diagnostic *WTG1007* with 0 Data Points in Train and 100% Accuracy in Test

```
  "failureMode"></param>
          public virtual void SubmitChanges(ConflictMode
              failureMode)
          {
<<<< DIAGNOSTIC: do not compare bool to a constant value. >>>>
-             if (this.objectTrackingEnabled == false)
+             if (!this.objectTrackingEnabled)
                  throw new InvalidOperationException("Object
                      tracking is not enabled for the current
                      data context
```

Figure 7.6 shows a noteable example, as it appears to understand an ambiguous diagnostic message and presents an appropriate solution. However due to its complexity, it is unlikely that a similar diagnostic has not previously been in training set. However, even if this were so, it would emphasize the advantage of using a neural network for static analysis fixing - as many fixers are similar, duplicate template-based codebase is handled by "one fixer to rule them all".

Listing 7.7: Example of Correct Solution with Camelcase Tokenization of Diagnostic *SA1116* with 30 Data Points in Train and 0% Accuracy in Test

```
          /// <param name="builderContext"></param>
          /// <returns></returns>
<<<< DIAGNOSTIC: the parameters should begin on the line after
    the declaration, whenever the parameter span across multiple
     lines >>>>
-         public ColumnExpression RegisterColumn(
    TableExpression table,
+         public ColumnExpression RegisterColumn(
+             TableExpression table,
```

Listing 7.8: Incorrect Prediction for Data Point shown in Figure 7.7

```
          /// <param name="builderContext"></param>
          /// <returns></returns>
<<<< DIAGNOSTIC: the parameters should begin on the line after
    the declaration, whenever the parameter span across multiple
     lines >>>>
-         public ColumnExpression RegisterColumn(
    TableExpression table,
+         public TableExpression RegisterColumn(
+             TableExpression table,
```

Figure 7.7 also shows an interesting example, as the network appears to understand the diagnostic message, but is unable to reproduce the identifiers correctly. One can therefore argue that stating that this diagnostic has a 0% accuracy is an underestimation, since the fix is mostly correct. Consequently, Figure 7.3 would shine a strict light upon the results, even though diagnostics with low success rates could in fact perform better given a looser metric. There is therefore reason to assume that generating more data for single diagnostics may be more effective than Figure 7.3 implies.

When examining the diagnostic messages of the diagnostics with high success rates, it appears that the neural network is capable of repairing several different types of diagnostics. Tables A.3, A.4, A.5, A.6 & A.7 show a list of diagnostics with 100% success rate, whereby behavioural changes display the largest amount of varying diagnostics. Behavioural changes are defined as changes that may change the execution of the code. The diagnostics in these tables are however filtered by hand, as diagnostics with similar diagnostic titles are removed for readability.

Altogether, the results look promising that a neural network can in fact

learn to imitate the combination of unknown code contexts with natural language comments and a structural fix, whilst also taking into account the style of the environment in terms of formatting. Specifically for the limited amount of less than 25k data points, these results are encouraging for future work in similar contexts. It also should be mentioned that the dataset only encompasses 421 diagnostics of the total of 2092 unique, fixable diagnostics found in Section 7.1, which leaves room for improvement to diversify the fixes the network can learn from. Lastly, as a downside to this experiment, related to the gathering of the dataset, is the observation that most fixes do not actually target code bugs, but more code style. It is possible that the neural network could fail fixing as effectively if the file context was syntactically incorrect.

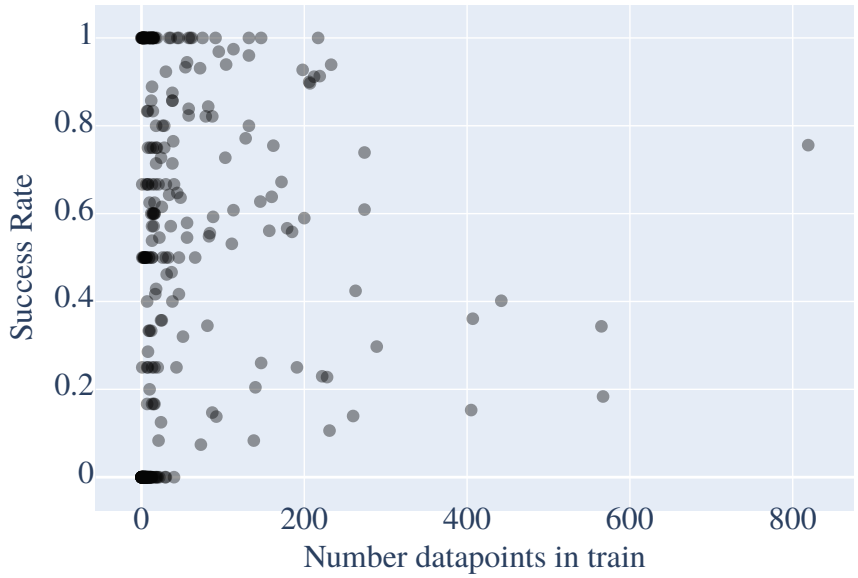

Figure 7.3: Impact of Training Data per Diagnostic on Accuracy for Imitation using Camelcase Tokenization

The training for this experiment is performed with a Nvidia GeForce GTX 1080.

Using a dataset of around 25k data points with 421 diagnostics, the best results were achieved by splitting identifiers by camelcase, whereby the neural network could successfully imitate 55.4% of data points. Normalizing these results by diagnostic, the success rate was 46.3%. The neural network was capable of predicting formatting changes, identifier changes and behavioural changes correctly.

7.3 RQ3: Extrapolating from Static Analysis Fixers

Training loss is shown in Figure 7.4 and validation loss in Figure 7.5. Interestingly, here the dataset with standard tokenization had its best validation results after 10k steps (4h 40min), whilst the dataset with camelcase tokenization achieved best results after 5k steps (1h 20min). However, like in the previous experiment, both training and validation loss are continuously lower for camelcase tokenization.

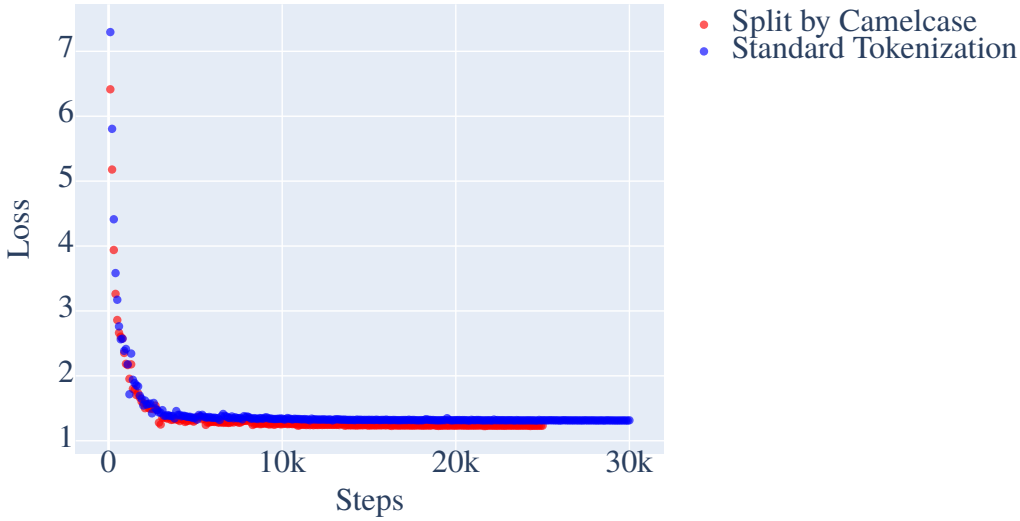

Figure 7.4: Training Loss for Extrapolation

Table 7.7 shows the aggregated results of the experiment. As mentioned in the experiment design and in contrast to the imitation experiments, none of diagnostics applied in test had been used in the training or validation dataset. This neural network was at best only capable of fixing 9.0% of data points. When normalizing the weighting per diagnostic for camelcase tokenization, this success rate rose to an average of 11.9%, but nonetheless leaving a significant gap towards the 46.3% in imitation. Generally, the results between standard and camelcase tokenization show a minimal difference,

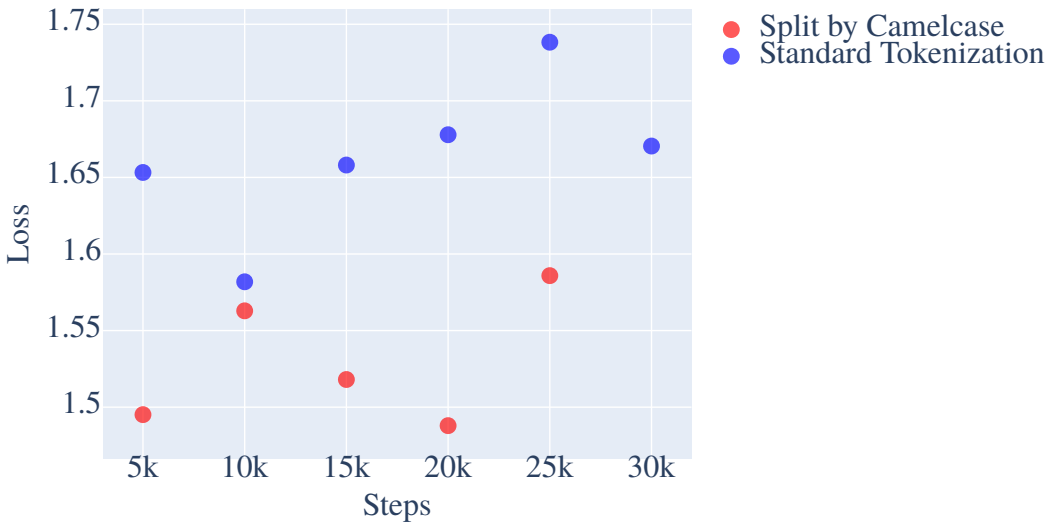

Figure 7.5: Validation Loss for Extrapolation

which indicates that identifier changes perform better when the diagnostic / edit representation has been seen before.

The following Figures 7.9, 7.10 and 7.12 show examples of how some data points were predicted. In contrast to the previous experiment, these examples show results from the dataset with standard tokenization. This is because these allowed more interpretation about the neural learning process for this experiment.

Listing 7.9: Correctly Predicted Example with Standard Tokenization of Diagnostic *VSD0001* with 0 Data Points in Train and 100% Accuracy in Test

```
 }
      }

      public ValueTask DisposeAsync() => StopAsync();

<<<< DIAGNOSTIC: asynchronous method "startcore" does not end
   with 'Async'. >>>>
-         private async Task StartCore()
```

Table 7.7: Percentage Results of Test Dataset

	Standard Tokenization	**Split by Camelcase**
Success Rate		
All Data Points	**9.0% (/1217)**	7.0% (/1240)
Diagnostic in Train	- (/0)	- (/0)
Diagnostic not in Train	9.0% (/1217)	7.0% (/1240)
Normalized Success Rate by		
Diagnostic	11.1% (41 diagnostics)	**11.9% (41 diagnostics)**
Diagnostic in Train	- (0 diagnostics)	- (0 diagnostics)
Diagnostic not in Train	11.1% (41 diagnostics)	11.9% (41 diagnostics)

```
+         private async Task StartCoreAsync()
      {
          _listener.Start();
```

Whilst Listing 7.9 shows a successful fix of applying an identifier change, it speaks against the quality of the dataset. This is because standard tokenization is used and the neural network must have previously learnt to relate `StartCore` to `StartCoreAsync`. With a dataset of 25k fixes, this only appears feasible when a similar diagnostic has been applied to exactly the same file context in the training dataset. Such homogeneity in the dataset can result in the neural network highly over-fitting between identifier tokens, instead of reasoning about what part of the identifier to change.

Listing 7.10: Example of Correct Solution with Standard Tokenization of Diagnostic *RCS1160* with 0 Data Points in Train and 33.3% Accuracy in Test

```
     private readonly Lazy<Task> _clientTask;
          private readonly Stopwatch _stopwatch = new Stopwatch
              ();

<<<< DIAGNOSTIC: abstract type should not have public
    constructors. >>>>
-         public SslClientBase(Configuration config)
+         protected SslClientBase(Configuration config)
```

```
        {
            if (config.MaxConnections < 1) throw new
                ArgumentOutOfRangeException(nameof(
```

Listing 7.11: Incorrect Prediction for Data Point shown in Figure 7.10

```
    private readonly Lazy<Task> _clientTask;
        private readonly Stopwatch _stopwatch = new Stopwatch
            ();
-
<<<< DIAGNOSTIC: abstract type should not have public
   constructors. >>>>
-         public SslClientBase(Configuration config)
+ SslClientBase(Configuration config)
          {
              if (config.MaxConnections < 1) throw new
                  ArgumentOutOfRangeException(nameof(
```

Listing 7.11 shows an interesting prediction, which is difficult to label as entirely incorrect. The diagnostic message allows room for interpretation, whereby removing the statement could be one. It can therefore be argued that the NN has developed a degree of understanding for natural language. It is however surprising that it is not capable of reproducing the same formatting environment, even though this is not entirely relevant to the fix. Nonetheless the latter shortcoming appears to be an outlier in further predictions and is not given much weight in this analysis.

Listing 7.12: Example of Correct Solution with Standard Tokenization of Diagnostic *RECS0092* with 0 Data Points in Train and 0% Accuracy in Test

```
  Collections.Generic;

  namespace Shadowsocks.Protocol.Shadowsocks.Crypto
  {
      static class CryptoProvider
      {
<<<< DIAGNOSTIC: convert field to readonly >>>>
-         static Dictionary<string, CryptoParameter> parameters
    = new Dictionary<string, CryptoParameter>
+         static readonly Dictionary<string, CryptoParameter>
   parameters = new Dictionary<string, CryptoParameter>
```

```
        {
```

Listing 7.13: Incorrect Prediction for Data Point shown in Figure 7.12

```
  Collections.Generic;

  namespace Shadowsocks.Protocol.Shadowsocks.Crypto
  {
      static class CryptoProvider
      {
<<<< DIAGNOSTIC: convert field to readonly >>>>
-         static Dictionary<string, CryptoParameter> parameters
    = new Dictionary<string, CryptoParameter>
+         static Dictionary<String, CryptoParameter> parameters
    = new Dictionary<String, CryptoParameter>
          {
```

In Listing 7.13, the only change the neural network applies is changing `string` to `String`. This implies that the network does not appear to understand the nature of the diagnostic and instead simply auto-encodes a line. This data point could be interpreted as the network developing an own understanding of what is "good"/"bad" code, which is not the goal of this experiment. However, it may also be interpreted as simplifying the representation of the file context, whereby lowercase letters for "String" are not rendered important enough.

In contrast to the previous experiment, the results to this research question are not as promising. It shows that the neural network is over-fitting to the diagnostic messages that it is trained with, which may result in good results for imitating them, but less so for extrapolating from them. Even though some understanding is developed for the diagnostic messages, it appears that a larger or more diverse dataset is required in training to be able to make more accurate predictions. Nonetheless, one can also argue positively that the network was generally capable of extrapolating from a limited amount of 297 diagnostics / edit representations. As mentioned in the previous experiment, the dataset only encompassed 421 diagnostics of the total of 2092 unique, fixable diagnostics found in RQ1, which potentially leaves significant room for improvement for neural learning.

For neural training, this experiment used the Nvidia GeForce RTX 2080 Ti, which accelerated the training times by more than a factor of two in comparison to the previous experiment.

Once more, using a dataset of around 25k data points with 421 diagnostics, the best normalized results were again achieved by splitting identifiers by camelcase. This resulted in an extrapolation success rate of 11.9% whereby 297 diagnostics had been seen in training. A diverse set of changes could be predicted, even though it was difficult to guarantee that semantically equivalent diagnostics had not been seen in the training set before. To improve these results, a dataset with more diagnostics is likely to help.

7.4 RQ4: Measuring the Influence of Formatting Tokens

Since camelcase tokenization in Imitation performed best, these results are analyzed here. Measuring the impact of percentage of formatting tokens in source/target on the success rate of predictions is depicted in Figure 7.6 and Figure 7.7.

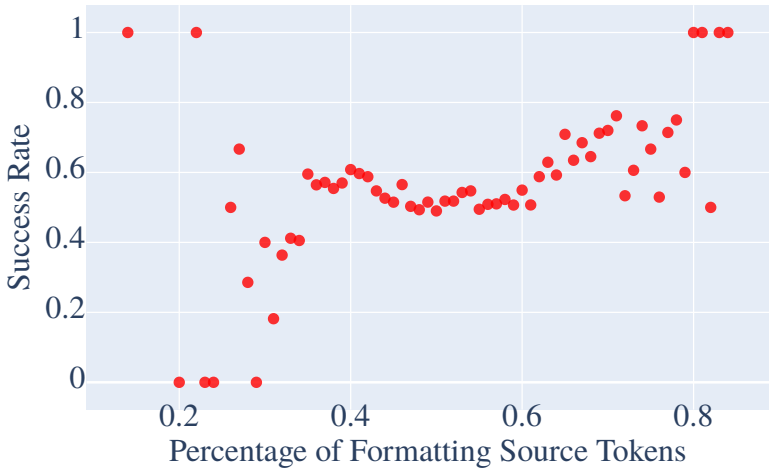

Figure 7.6: Average Success Rates for Data Points with given Percentage of Formatting Tokens in Input

For formatting tokens in source, the result appears ambiguous. Generally, the points on the graph with 0% or 100% success rates can be viewed skeptically, since they can be an indication of containing only one or two data points. The local minimum at around 50% formatting tokens could be explained with simply having most data points in this region and therefore in fact showing the most precise results. Apart from this, the figure suggests that a higher percentage of formatting tokens in input leads to marginally better results.

Regarding the percentage of formatting tokens in output, the result does not produce any pattern. Even output that consists mostly of formatting produces varying results. Therefore, the conclusion is reached, that predicting

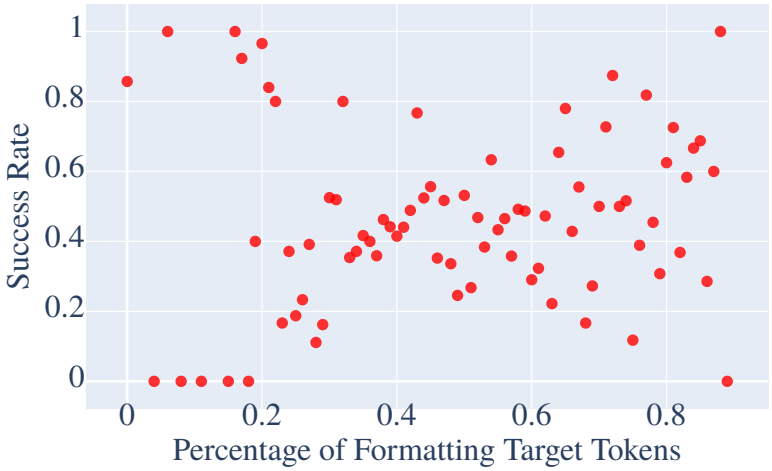

Figure 7.7: Average Success Rates for Data Points with given Percentage of Formatting Tokens in Output

formatting tokens is just as difficult as predicting normal tokens for the applied Transformer model.

For both input and output, the percentage of formatting tokens does not have a clear impact on success rates. However, whilst it seems that for input, more formatting can increase success rates minimally, the relation for target tokens appears highly random.

Chapter 8

Discussion

8.1 Threats to Validity

RQ1 Often the assumption was made that the analyzer and fixer for a diagnostic was contained in the same NuGet. Validating this would have required reading through diagnostic messages and analyzing repair behaviour for single diagnostics. Furthermore, analyzing assemblies with reflection did occasionally cause errors. There is a possibility that valid diagnostics were excluded in the quantification.

RQ2 With less than 25k data points, the total dataset is small. Furthermore, when gathering the dataset, the assumption was made that a fix would be applied in the immediate proximity of the diagnostic. It may have been the case that this caused the neural network to learn unrelated diagnostic messages and fixing patterns, thereby generally skewing the results.

RQ3 In addition to RQ2, public benchmarks may have been necessary to reflect the true potential of the model, instead of using data from the same distribution. Furthermore, the results may have changed significantly if the diagnostics had been shuffled differently.

RQ4 The calculations do not take into account the amount of data points with a high and small amount of percentual formatting. Thereby, these areas of the depicted figures are more easily affectable by single data points.

8.2 Limitations

A major limitation for RQ2 and RQ3 is that only one-batch diffs were gathered for the dataset. This was done because it took a fraction of the time to fix all occurrences of a diagnostic in a C# solution using the FixAllProvider than to fix one occurrence after another. Had the latter been applied, one diagnostic location could have been mapped to changes on non-consecutive lines, providing a more accurate dataset.

Furthermore, had time not been a constraint, all 54 C# repositories proposed by Yin et al. would have been mined for fixes instead of only 28 [38]. Even though judging by Figure 7.3, it is not clear whether more data points for single diagnostics would have been useful, a larger diversity of diagnostics is likely to have made results more accurate.

In terms of neural network architecture, a tool that may have improved results significantly for RQ2 and RQ3 is the copy mechanism, which is also applied by Chen et al. [11]. This allows the neural network to avoid outputting identifiers when it is more straightforward to simply copy them from the input.

All research questions only focus on static analysis in C#. Even though this is a large limitation, it also helped in focusing efforts. Also, no experiments were made to see how these experiments performed with real humans. It was measured to what extent the neural network could learn from the dataset, which may in itself also had fixes that were unacceptable to humans.

8.3 Ethics and Sustainability

Finding code bugs and fixing them can be a time-consuming line of work. Given the modern tools of artificial intelligence, is is questionable whether these tasks should still be entirely assigned to humans. This thesis aims to enable developers to focus more on the creation of usable software, rather than cleaning up errors. This essentially saves humanity time, effort and money.

Chapter 9

Conclusions and Future work

9.1 Future Work

9.1.1 Generating More and Better Data

Due to time constraints and difficulties porting various C# codebases between Unix and Windows, the generation of the dataset can be improved to a great extent. Specifically, instead of launching Roslynator from Powershell, and loading every C# solution each time an analysis or fix is run for a single diagnostic, it may be beneficial to edit the source-code of Roslynator and port the logic described in Section **??** to C# as well. Furthermore, this may enable better error-handling, so that both Roslynator runtime errors can be debugged more efficiently and time is spared in generating the dataset. As mentioned previously, only 421 diagnostic of the 2540 fixable diagnostics found in Section 7.1, and 28 of 54 C# repositories were used in the dataset.

9.1.2 PCA Analysis

A task that once more combines RQ1 with RQ2 and RQ3, is doing a Principal Component Analysis (PCA) analysis on the trained neural network, similarly to the work of Yin et al. [38]. The diagnostic message without the file content could be inputted into the neural network and the resulting encodings plotted in such a way that semantic clusters can be made visible. This could perhaps add further insight to how many truly independent diagnostic messages there exist, which might add further value in identifying duplicates or avoiding the creation of analyzers and fixers that already exist. For this task, it may also be more useful to either use a LSTM or a multi-encoder NN, which is

able to separate the encoded diagnostic message from the code snippet in the training process. Both tools are provided by the OpenNMT implementation in Tensorflow.

9.1.3 Code Reviews

Another task highly related to neural repair of static analysis warnings is using the trained model from RQ2 or RQ3 to evaluate on code reviews for pull requests. This is because a code review can essentially be viewed as a diagnostic to a code snippet. On the other hand, the same neural network could also be trained with code review data and then used in static analysis tools instead. Using real code reviews would generally also introduce the problem statement of false positives in static analysis, which are not addressed in this thesis. The neural network thereby has an additional option for output, which is ignoring a message. For this task, the work of Gupta et al. can be examined, which train a neural network to identify whether a code review is actionable or not [25]. An example of a non-actionable review would be "What's the point of this change?". According to Gupta, code reviews and static analysis share the similarity of being targeted towards "obvious bugs or best practices".

9.1.4 Changes in Handling Identifiers and Natural Language

For RQ2 and RQ3, there are a few more variables that can be tested to improve the results of the neural network. Additionally to the camelcase tokenization, a multi-embedding framework such as proposed by Siow et al. can be used to combine both character-level with word-level tokenization [53]. Once again, the idea would be to capture more semantics and enable the neural network to repair more types of fixes. However, as already mentioned in Section 8.2 for limitations, using a copy mechanism could also already provide a large benefit in combination with camelcase tokenization. On the other hand, Siow et al. provide further inspiration in terms of applying word lemmatization by NLTK, which may be considered for diagnostic messages in future work [26]. This could reduce the variability in input for tokens that represent a similar meaning.

9.1.5 Other Neural Models

In terms of alternative neural network models, specifically reinforcement learning could provide an interesting new approach to the problems presented in RQ2 and RQ3. For this, a static analysis tool such as Roslynator could

be provided as an environment, which gives the neural-based actor a reward, based on whether it has removed or created new static analysis warnings. This would be similar to the work of Gupta et al., which have used a compiler as an oracle [20]. However, this approach would assume that all static analyzers are "correct", which is not done in this thesis.

9.1.6 Recommendation for EA

In case there exists an interest in building a code generator to supplement code reviews with transfer-knowledge from static analysis, one could supplement the built dataset with accepted code changes from any version control tool such as GitHub.

On the other hand, one may also assume that the next generation of static analysis is not based on open-source template-based static analysis tools, but on neural networks that have learnt from live fixes. Given the definition of static analysis of providing diagnostics and fixing them without executing the code, this mechanism would still be included. There may be a risk that attempting to learn from static analysis tools may be the wrong path when building tools to fix unanticipated bugs, as in RQ3. Synthetically manipulated code or code fixes mined from GitHub could be more effective to learn from as for example done in SequenceR by Chen et al. [11].

9.2 Conclusion

Given the contribution in this thesis of presenting the first meta analysis of the state-of-the art of open-source static analysis in C#, it would be interesting to see whether this has any impact on future frameworks in this field. Furthermore, it cannot be ruled out that this sparks future research within this field, related to neural repair or not. This can be compared to the initial situation for this thesis, where it was unclear whether the time-intensive task of exploring nuget.org for code analysis would provide any useful knowledge. Therefore this research does perhaps provide a lower entry boundary in the area.

For developers and researchers wanting to create new repair packages with the given C# static analysis tools, this research may also yield another entrypoint in understanding the underlying mechanisms. Furthermore, time may also be saved in either more effective research on whether a certain fixing method has already been written or how to create the fixer through the proposed method of using neural networks.

In terms of personal goals, it has been highly interesting to observe how quickly and effectively neural networks can learn to imitate complex features in a limited dataset in training times of a few hours. In this sense, the outcome has been rewarding. However, to reduce the dependency of generating a dataset, future personal work would focus more on unsupervised learning or reinforcement learning.

References

[1] B. Wagner and Y. Victor, "Tutorial: Write your first analyzer and code fix," Mar. 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix

[2] M. Monperrus, "Automatic software repair: a bibliography," *CoRR*, vol. abs/1807.00515, 2018. [Online]. Available: http://arxiv.org/abs/1807.00515

[3] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1st ed. USA: CRC Press, Inc., 1999. ISBN 0849371813

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735

[5] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[6] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *CoRR*, vol. abs/1409.1259, 2014. [Online]. Available: http://arxiv.org/abs/1409.1259

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[8] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions*

on Neural Networks, vol. 20, no. 1, pp. 61–80, 2009. doi: 10.1109/TNN.2008.2005605

[9] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2017.

[10] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[11] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *CoRR*, vol. abs/1901.01808, 2019. [Online]. Available: http://arxiv.org/abs/1901.01808

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014. doi: 10.1145/2610384.2628055. ISBN 9781450326452 p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[13] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," 2017. [Online]. Available: https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603/13921

[14] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," *CoRR*, vol. abs/2005.10636, 2020. [Online]. Available: https://arxiv.org/abs/2005.10636

[15] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[16] V. Okun, A. Delaitre, and P. Black, "Report on the static analysis tool exposition (sate) iv," 2013-02-04 2013.

[17] J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and S. P. Chin, "Learning to repair software vulnerabilities with generative adversarial networks," *CoRR*, vol. abs/1805.07475, 2018. [Online]. Available: http://arxiv.org/abs/1805.07475

[18] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016. doi: 10.1038/nature16961

[19] Z. Li, X. Jiang, L. Shang, and H. Li, "Paraphrase generation with deep reinforcement learning," *CoRR*, vol. abs/1711.00279, 2017. [Online]. Available: http://arxiv.org/abs/1711.00279

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[21] R. Gupta, A. Kanade, and S. K. Shevade, "Deep reinforcement learning for programming language correction," *CoRR*, vol. abs/1801.10467, 2018. [Online]. Available: http://arxiv.org/abs/1801.10467

[22] R. Bellman, *Dynamic Programming*. Dover Publications, 1957. ISBN 9780486428093

[23] A. Barriga, A. Rutle, and R. Heldal, "Automatic model repair using reinforcement learning," in *MODELS Workshops*, 2018.

[24] Y. Keneshloo, T. Shi, N. Ramakrishnan, and C. K. Reddy, "Deep reinforcement learning for sequence to sequence models," *CoRR*, vol. abs/1805.09461, 2018. [Online]. Available: http://arxiv.org/abs/1805.09461

[25] A. Gupta, "Intelligent code reviews using deep learning," 2018.

[26] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[27] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[28] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018. [Online]. Available: http://arxiv.org/abs/1808.01400

[29] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," *CoRR*, vol. abs/1911.01205, 2019. [Online]. Available: http://arxiv.org/abs/1911.01205

[30] "Common weakness enumeration," Jul. 2021. [Online]. Available: https://cwe.mitre.org/index.html

[31] A. Tanwar, K. Sundaresan, P. Ashwath, P. Ganesan, S. K. Chandrasekaran, and S. Ravi, "Predicting vulnerability in large codebases with deep code representation," *CoRR*, vol. abs/2004.12783, 2020. [Online]. Available: https://arxiv.org/abs/2004.12783

[32] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," *CoRR*, vol. abs/1807.04320, 2018. [Online]. Available: http://arxiv.org/abs/1807.04320

[33] Y. Zheng, S. Pujar, B. L. Lewis, L. Buratti, E. A. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2A: A dataset built for ai-based vulnerability detection methods using differential analysis," *CoRR*, vol. abs/2102.07995, 2021. [Online]. Available: https://arxiv.org/abs/2102.07995

[34] "Clang static analyzer," Jul. 2021. [Online]. Available: https://clang-analyzer.llvm.org

[35] "Cppcheck," Jul. 2021. [Online]. Available: http://cppcheck.sourceforge.net

[36] D. A. Wheeler, "Flawfinder," Jul. 2021. [Online]. Available: https://dwheeler.com/flawfinder/

[37] Facebook, "Infer - static analyzer." [Online]. Available: https://fbinfer.com

[38] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *CoRR*, vol. abs/1810.13337, 2018. [Online]. Available: http://arxiv.org/abs/1810.13337

[39] S. A. Qureshi, S. Mehta, R. Bhagwan, and R. Kumar, "Assessing the effectiveness of syntactic structure to learn code edit representations,"

CoRR, vol. abs/2106.06110, 2021. [Online]. Available: https://arxiv.org/abs/2106.06110

[40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. doi: 10.1145/3290353. [Online]. Available: https://doi.org/10.1145/3290353

[41] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014. doi: 10.1145/2635868.2635883. ISBN 9781450330565 p. 281–293. [Online]. Available: https://doi.org/10.1145/2635868.2635883

[42] "Checkstyle," Jul. 2021. [Online]. Available: https://checkstyle.sourceforge.io

[43] B. Loriot, F. Madeiral, and M. Monperrus, "Styler: Learning formatting conventions to repair checkstyle errors," *CoRR*, vol. abs/1904.01754, 2019. [Online]. Available: http://arxiv.org/abs/1904.01754

[44] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 780–791. [Online]. Available: https://proceedings.mlr.press/v139/berabi21a.html

[45] "eslint- npm," 2021. [Online]. Available: https://www.npmjs.com/package/eslint

[46] B. C. Stivalet, "C# vulnerability test suite," Sep. 2016. [Online]. Available: https://samate.nist.gov/SRD/view.php?tsID=105

[47] A. Delaitre, B. Stivalet, P. Black, V. Okun, T. Cohen, and A. Ribeiro, "Sate v report: Ten years of static analysis tool expositions," 2018-10-23 2018.

[48] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der

Walt and Jarrod Millman, Eds., 2010. doi: 10.25080/Majora-92bf1922-00a pp. 56 – 61.

[49] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://www.aclweb.org/anthology/P17-4012

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[51] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[52] "Pygments," Aug. 2021. [Online]. Available: https://pygments.org/faq/

[53] J. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 284–295, 2020.

Appendix A

Additional Tables

Table A.1: Example Host Packages

NuGet Package	Version
AIT.CodeQualityProfile	0.2.0
Strato.Analyzers	1.0.18
qhy.codeanalyzers	1.1.18
AnalyzerDotNet	1.0.3
Akka.Analyzer	0.1.0
Bet.CodeAnalyzers	1.0.10
OBeautifulCode.Build.Analyzers	1.1.136
Appccelerate.Development.Analyzers	2.2.0

Table A.2: Examples of Analyzers Missing

Diagnostic ID	NuGet Package	Analyzer/Fixer
CS1570	Microsoft.CodeAnalysis.CSharp v3.9.0	Analyzer
CS1570	DotNetAnalyzers. DocumentationAnalyzers.Unstable v1.0.0.59	Fixer
CS1105	Microsoft.CodeAnalysis.CSharp v3.9.0	Analyzer
CS1105	NR6Pack v0.10.0	Fixer

Table A.3: Formatting Changes: Diagnostic Messages of Diagnostics with 100% Success Rate in Imitation for Camelcase Tokenization

NuGet	**Diagnostic ID**	**Diagnostic Title**
MR.Analyzers.Whitespace.1.0.1	WS1001	Missing final newline.
Roslynator.Analyzers.3.1.0	RCS1036	Remove redundant empty line.
StyleCop.Analyzers.1.1.118	SA1000	Keywords should be spaced correctly
StyleCop.Analyzers.1.1.118	SA1012	Opening braces should be spaced correctly
StyleCop.Analyzers.1.1.118	SA1513	Closing brace should be followed by blank line
StyleCop.Analyzers.1.1.118	SA1107	Code should not contain multiple statements on one line
StyleCop.Analyzers.1.1.118	SA1514	Element documentation header should be preceded by blank line
StyleCop.Analyzers.1.1.118	SA1515	Single-line comment should be preceded by blank line
StyleCop.Analyzers.1.1.118	SA1516	Elements should be separated by blank line
ZWA.Infrastructure.CodeAnalysis.1.0.0	FS0065	Remove trailing whitespace

Table A.4: Documentation Changes: Diagnostic Messages of Diagnostics with 100% Success Rate in Imitation for Camelcase Tokenization

NuGet	**Diagnostic ID**	**Diagnostic Title**
APIDocCodeAnalyzer.1.0.6253.30470	DA4004	Type doesn't have example in the comment
APIDocCodeAnalyzer.1.0.6253.30470	DA3006	Method doesn't have /// returns in comment

Table A.5: Coding Conventions Changes: Diagnostic Messages of Diagnostics with 100% Success Rate in Imitation for Camelcase Tokenization

NuGet	**Diagnostic ID**	**Diagnostic Title**
Agoda.Analyzers.1.0.517	SA1107	Code must not contain multiple statements on one line
AsyncAnalyzers.1.1.7.0	_MissingAsync	TAP methods must end with Async.
WTG.Analyzers.2.10.4	WTG1009	Using directives must be ordered by kind.
Roslynator.Analyzers.3.1.0	RCS1001	Add braces (when expression spans over multiple lines).
ZWA.Infrastructure.CodeAnalysis.1.0.0	FS0037	Remove commented code.
codecracker.CSharp.1.1.0	CC0049	Simplify expression

Table A.6: Behavioural Changes Part 1: Diagnostic Messages of Diagnostics with 100% Success Rate in Imitation for Camelcase Tokenization

NuGet	**Diagnostic ID**	**Diagnostic Title**
BND.Convention.Analyzer.1.0.0	ConstantAnalyzer	Variable can be made constant,
CodeFix.1.2.1.0	UniversalTimeStamp	Use DateTime.Now.ToUniversalTime() insted of DateTime.Now
DebuggerStepThroughRemover.1.0.5823.28645	DebuggerStepThroughRemover	Type is decorated with DebuggerStepThrough attribute
Menees.Analyzers.2017.2.0.3	MEN009	Use the preferred exception type

Table A.7: Behavioural Changes Part 2: Diagnostic Messages of Diagnostics with 100% Success Rate in Imitation for Camelcase Tokenization

NuGet	**Diagnostic ID**	**Diagnostic Title**
Microsoft.AnalyzerPowerPack.1.1.0	CA1052	Static holder types should be Static or NotInheritable
Microsoft.CodeAnalysis.NetAnalyzers.5.0.3	CA1836	Prefer IsEmpty over Count
Microsoft.CodeAnalysis.NetAnalyzers.5.0.3	CA1805	Do not initialize unnecessarily
RefactoringEssentials.5.6.0	RECS0014	If all fields, properties and methods members are static, the class can be made static.
RefactoringEssentials.5.6.0	RECS0122	Initializing field with default value is redundant
Roslynator.Analyzers.3.1.0	RCS1090	Add call to 'ConfigureAwait' (or vice versa).
Roslynator.Analyzers.3.1.0	RCS1015	Use nameof operator.
Roslynator.Analyzers.3.1.0	RCS1156	Use string.Length instead of comparison with empty string.
SmartAnalyzers.MultithreadingAnalyzer.1.1.31	MT1001	Lock on this reference
StyleCop.Analyzers.1.1.118	SA1205	Partial elements should declare access
StyleCop.Analyzers.1.1.118	SA1209	Using alias directives should be placed after other using directives
VSDiagnostics.1.10.0	VSD0007	An ArgumentException should use nameof() to refer to a variable.
WTG.Analyzers.2.10.4	WTG1007	Do not compare bool to a constant value.
ZWA.Infrastructure.CodeAnalysis.1.0.0	FS0052	Make field readonly
ZWA.Infrastructure.CodeAnalysis.1.0.0	FS0034	Redundant field assignment
codecracker.CSharp.1.1.0	CC0060	Abstract class should not have public constructors.
mstest-analyzer.0.6.1	MT1001	Public class should have a test attribute

For DIVA

{
"Author1": {
"Last name": "Lohse",
"First name": "Vincent Paul",
"Local User Id": "vplohse",
"E-mail": "vplohse@kth.se",
"ORCiD": "0000-0002-3539-675X",
"organisation": {"L1": "School of Electrical Engineering and Computer Science ",
}
},
"Degree": {"Educational program": "Master's Programme, Software Engineering of Distributed Systems, 120 credits"},
"Title": {
"Main title": "Exploring the Usage of Neural Networks for Repairing Static Analysis Warnings",
"Language": "eng" },
"Alternative title": {
"Main title": "Utforsking av användningen av neurala nätverk för att reparera varningar för statisk analys",
"Language": "swe"
},
"Supervisor1": {
"Last name": "Chen",
"First name": "Zimin",
"Local User Id": "zimin",
"E-mail": "zimin@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science ",
"L2": "Computer Science" }
},
"Examiner1": {
"Last name": "Monperrus",
"First name": "Martin",
"Local User Id": "monperrus",
"E-mail": "martin.monperrus@csc.kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science ",
"L2": "Computer Science" }
},
"Cooperation": { "Partner_name": "DICE (EA Digital Illusions CE AB)"},
"Other information": {
"Year": "2021", "Number of pages": "xvii,75"}
}

TRITA -EECS-EX-2021:830