Unit Test Generation using Machine Learning

Laurence Saes
l.saes@live.nl

August 23, 2018, 57 pages

Academic supervisor: dr. Ana Oprescu
Host organization: Info Support B.V. www.infosupport.com
Host supervisor: Joop Snijder

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Test suite generators could help software engineers to ensure software quality by detecting software faults. These generators can be applied to software projects that do not have an initial test suite, a test suite can be generated which is maintained and optimized by the developers. Testing helps to check if a program works and, also if it continues to work after changes. This helps to prevent software from failing and aids developers in applying changes and minimizing the possibility to introduce errors in other (critical) parts of the software.

State-of-the-art test generators are still only able to capture a small portion of potential software faults. The Search-Based Software Testing 2017 workshop compared four unit test generation tools. These generators were only capable of achieving an average mutation coverage below 51%, which is lower than the score of the initial unit test suite written by software engineers.

We propose a test suite generator driven by neural networks, which has the potential to detect mutants that could only be detected by manually written unit tests. In this research, multiple networks, trained on open-source projects, are evaluated on their ability to generate test suites. The dataset contains the unit tests and the code it tests. The unit test method names are used to link unit tests to methods under test.

With our linking mechanism, we were able to link 27.41% (36,301 out of 132,449) tests. Our machine learning model could generate parsable code in 86.69% (241/278) of the time. This high number of parsable code indicates that the neural network learned patterns between code and tests, which indicates that neural networks are applicable for test generation.

ACKNOWLEDGMENTS

This thesis is written for my software engineering master project at the University of Amsterdam. The research was conducted at Info Support B.V. in The Netherlands for the Business Unit Finance.

First, I would like to thank my supervisors Ana Oprescu of the University of Amsterdam and Joop Snijder of Info Support. Ana Oprescu was always available to give me feedback, guidance, and helped me a lot with finding optimal solutions to resolve encountered problems. Joop Snijder gave me a lot of advice and background in machine learning. He helped me to understand how machine learning could be applied in the project and what methods have great potential.

I would also like to thank Terry van Walen and Clemens Grelck for their help and support during this project. The brainstorm sessions with Terry were very helpful and gave a lot of new insights in solutions to the encountered problem. I am grateful for Clemens help too in order to prepare me for the conferences. My presentation skills have improved a lot and I really enjoyed the opportunity

Finally, I would like the thank the University of Amsterdam and Info Support B.V. for all their help and funding so that I could present at CompSys 2018 in Leusen, Netherlands, and Sattose 2018 in Athens, Greece. This was a great learning experience, and I am very grateful to have had these opportunities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Test suites are used to ensure software quality when a program's code base evolves. The capability of producing the desired effect (effectiveness) of a test suite is often measured as the ability to uncover faults in a program [ZM15]. Although intensively researched [AHF+17, KC17, CDE+08, FZ12, REP+11], state-of-the-art test suite generators lack test coverage that could be achieved with manual testing. Almasi et al. [AHF+17] explained a category of faults that are not detectable by these test suite generators. These faults are usually surrounded by complex conditions and statements for which complex objects have to be constructed and populated with specific values.

The Search-Based Software Testing (SBST) Workshop of 2017 had a competition of Java unit test generators. In the competition, test suite effectiveness of test suite generators and manually written test suites were evaluated. The effectiveness of the test suites are measured by their ability to find faults and is measured with the mutation score metric. The ability to find faults can be measured with mutation score because mutations are a valid substitute for software faults [JJI+14]. The mutation score of a test suite represents the test suite's ability to detect syntactic variations of the source code (mutants), and is computed using a mutation testing framework. In the workshop, manually written test suites score on average 53.8% mutation coverage, while the highest score obtained by a generated test suite is 50.8% [FRCA17].

However, it is impossible to conclude that all possible mutants are detected even when all generated mutants are covered since the list of possible mutations is infinite. It is infinite because some methods can have an infinite amount of output values, and mutants can be introduced that only change one of these output values.

We need to leverage the ability to automatically test many different execution paths and the capability to learn how to test complex situations of generated and manually written test suites. Therefore, we propose a test suite generator that uses machine learning techniques.

A neural network is a machine learning algorithm, that can learn complex tasks without being programmed with rules. The network learns from examples and captures the logic that is inside. Thus, new rules can be taught to the neural network by just showing examples of how the translation is done.

Our solution uses neural networks and combines manual and automated test suites by learning patterns between tests and code to generate test suites with higher effectiveness.

1.1 Types of testing

There are two software testing methods, i) black box testing [Ost02a] and ii) white box testing [Ost02b]. With black box testing, the project's source code is not used to create tests. Only the specification of the software is used [LKT09]. White box testing is a method that uses the source code to create tests. The source code is evaluated, and its behavior is captured [LKT09]. White box testing focuses more on the inner working of the software, while black box testing focuses more on specifications [ND12]. Black box testing is more efficient with testing of large code blocks as only the specification has to be evaluated, while white box testing is more efficient in testing hidden logic.

Our unit test generator can be categorized as white box testing since we use the source code to generate tests.

1.2 Neural networks

Neural networks are inspired by how our brains work [Maa97]. Our brain uses an interconnected network of neurons to process everything that we observe. A neuron is a cell that receives input from other neurons. An observation is sent to the network as the input signal. Each neuron in the network sends a signal to the next cells in the network based on the signals that it received. With this approach, the input translates to particular values at the output of the network. Humans perform actions based on this output. This mechanism is similar to the concept of how a neural network works.

A neural network can be seen as one large formula. Like the networks in our brain, a neural network also has an input layer, hidden layers, and an output layer. In our solution, an input layer is a group of neural network cells that receive the input for a translation that is going to be made. An encoder performs the mapping of the input over the input cells. After the input layers there are the hidden layers. The first layer receives the values of the input layer and sends a modified version of that value, based on its configuration, to the next layer. The other layers work in the same way as the first layer of the input. The only difference is that they receive the value from the last hidden layer instead of the input layer. Eventually, a particular value arrives at the last layer (output layer) and is decoded as the prediction. A visualization is shown in Figure 1.1.

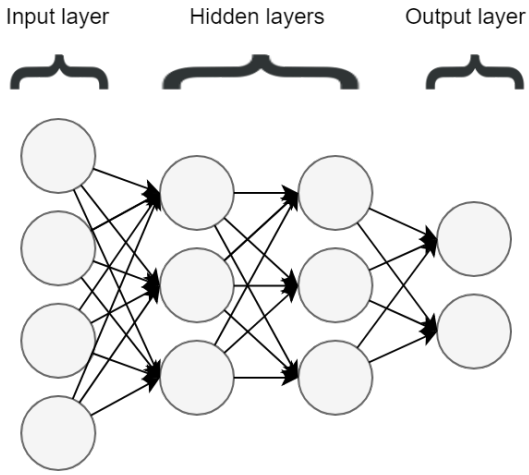


Figure 1.1: Visualization of a neural network

The configuration of the neural network cells is the logic behind the predictions. The configuration has to be taught to the network by giving training examples. For example, we can teach the neural network the concept of a house by giving example pictured of houses and non-houses. The non-houses are needed to teach the difference between a house and something else. The neurons are configured in a way to classify based on the training data. This configuration can later be used to make predictions on unseen data.

For our research, we have to translate a sequence of tokens into another sequence of tokens. A combination of multiple networks with a specific type of cells is required for the translation [CVMG+14]. These cells are designed so that they can learn long-term dependencies. Thus, during the predictions, the context of the input sequence is also considered. Cho et al. [CVMG+14] have designed a network with these specifications. They have used a network with one encoder and one decoder. The encoder network translates the variable length input sequences to a fixed size buffer. The decoder network translates this buffer into a variable size output. We can use this setup for our translations by using the variable sized input and output as our input and output sequences.

1.3 Research questions

Our **research goal** is to study machine learning approaches to generate test suites with high effectiveness: learn how code and tests are linked and apply this logic on the project's code base. Although neural networks are widely used for translation problems [SVL14], training them is often time-consuming. Therefore, we also research heuristics to alleviate this issue. This is translated into the following research questions:

RQ1 What neural network solutions can be applied to generate test suites in order to achieve a higher test suite effectiveness for software projects?

RQ2 What is the impact of input and output sequence compression on the training time and accuracy?

1.4 Contribution

In this work, we contribute an algorithm to link unit tests to the method under test, a training set for translating code to tests with more than 52,000 training examples, software to convert code to different representations and also support the translation back, and a neural network configuration with the ability to learn patterns between code and tests. Finally, we also contribute a pipeline that takes as input GitHub repositories and has as output a machine learning model that can be used to predict tests. As far as we know, we are the first to perform experiments in this area. Therefore, the linking algorithm and the neural network configuration can be used as a baseline for future research. The dataset can also be used on varies other types of machine learning algorithms for the development of a test generator.

1.5 Outline

We address the background of test generation, code analysis and machine learning in Chapter 2. In Chapter 3, we discuss how a test generator could be designed in general that uses machine learning. In Chapter 4, we list projects that can be used for evaluation baseline, and we introduce metrics to measure the progress of developing the test suite generator and how well it performed compared to other generators on a baseline. How we develop our test generator can be found in Chapter 5. Our results are presented in Chapter 6 and discussed in Chapter 7. Related work is listed in Chapter 8. We conclude our work in Chapter 9. Finally, an overview of related work to this thesis can found in Chapter 10.

Chapter 2

Background

Multiple approaches address the challenge of achieving a high test suite effectiveness. Tests could be generated based on the project's source code by analyzing all possible execution paths. An alternative is using test oracles, which can be trained to distinguish between correct and incorrect method output. Additionally, many code analysis techniques can be used to gather training examples and many machine learning algorithms can be used to translate from and/or to code.

2.1 Test generation

Common methods for code-based test generation are random testing [AHF+17], search-based testing [FRCA17, AHF+17], and symbolic testing [CDE+08]. Almasi et al. benchmarked random testing and search-based testing on the closed source project LifeCalc [AHF+17] and found that search-based testing had at most 56.40% effectiveness, while random testing achieved at most 38%. They did not analyze symbolic testing because there was no symbolic testing tool available that supported the analyzed project's language. Cadar et al. [CDE+08] applied symbolic testing on the HiStar kernel achieving 76.4% test suite effectiveness compared to 48.0% with random testing.

2.1.1 Test oracles

A test oracle is a mechanism that can be used to determine whether a method output is correct or incorrect. Testing is performed by executing the method under test with random data and evaluating the output with the test oracle.

Fraser et al. [FZ12] analyzed an oracle generator that generated assertions based upon mutation score. For Joda-time, the oracle generator covered 82.95% of the mutants compared to 74.26% for the manual test suite. For Commons-Math, the oracle generator covered 58.61% of the mutants compared to 41.25% for the manual test suite. Their test oracle generator employs machine learning to create the test oracles. Each test oracle captures method behavior for a single method in the software program by training on the method with random input. Contrary to this approach, our proposed method generates code while this method predicts the output of methods.

2.2 Code analysis

Multiple methods can be used to analyze code. Two possibilities are the analysis of i) bytecode or ii) the program's source code. The biggest difference between bytecode analysis and source code analysis is that bytecode is closer to the instructions that form the real program and has the advantage of more available concrete type information. For this language, it is easier to construct a call graph, which can be used to determine the concrete class of certain method calls.

With the analysis of bytecode, the output of the Java compiler is analyzed and could be performed by using libraries. For instance, the T.J. Watson Libraries for Analysis (WALA) [1]. The library will

[1] http://wala.sourceforge.net

generate the call graph and provides functionality that can be applied on the graph. With the analysis of source code, the source code in the representation of an abstract syntax tree (AST) is analyzed. For AST analysis, JavaParser [2] can be used to construct an AST, and the library provides functionality to perform operations on the tree.

2.3 Machine learning techniques

Multiple neural network solutions could translate sequences (translating an input sequence to a translated sequence). For our research, we expect that sequence-to-sequence (seq2seq) neural networks based on recurrent neural networks (RNNs) or convolutional neural networks (CNNs) are most promising. The version that uses RNNs can be configured to contain long short-term memory (LSTM) nodes [SVL14, SSN12] or gated recurrent unit (GRU) nodes [YKYS17a] and can be configured with an attention mechanism so it can make predictions on long sequences [BCB14]. Bahdanau [BCB14] evaluated the attention mechanism. They tested sequence length until a length of 60 tokens and used bilingual evaluation understudy (BLEU) score as metric. The BLEU score is used to calculate the quality of translations. The higher the score, the better. The quality of the predictions was the same for using both attention or no attention mechanism until a length of 20 tokens. The quality dropped from approximately 27 BLEU to approximately 8 BLEU when using no attention mechanism, and dropped from approximately 27 BLEU to approximately 26 BLEU when using an attention mechanism. Chung et al. [CGCB14] made a comparison between LSTM nodes and GRU nodes to predict the next time step in songs. They found that the quality of prediction of both LSTM and GRU are comparable. GRU outperforms except on one dataset by Ubisoft. However, they stated that the prediction quality of both could not be clearly distinguished. These networks could perform well to translate code to unit tests because they can make predictions on long sequences.

An alternative to RNNs, are CNNs. Recent research shows that CNNs can also be applied to make predictions based on source code [APS16]. In addition, Gehring et al. [GAG+17] were able to train CNN models up to 21.3 times faster compared to RNN models. However, in other research, GRU outperforms CNN with handling long sequences correctly [YKYS17b]. We also look into CNNs because in our case it could make better predictions, especially because they are faster to train what enables us to use larger networks.

There are also techniques in research that could be used to prepare the training data in order to optimize the training process. Hu et al. [HWLJ18] used a structure-based traversal (SBT) in order to capture code structure in a textual representation, Ling et al. [LGH+16] used compression to reduce sequence lengths, and Sennrich et al. used byte pair encoding (BPE) [SHB15] to support better predictions on words that are written differently but have the same meaning.

In conclusion, to answer **RQ**1, we evaluate both CNNs and RNNs, as both tools look promising. We apply SBT, code compression, and BPE to find out if these techniques improve the results when translating methods into unit tests.

[2] https://javaparser.org/

Chapter 3

A Machine Learning-based Test Suite Generator

Our solution focuses on generating test suites for Java projects that have no test suite at all. The solution requires the project's method bodies and the name of the classes to which they belong. The test generator sends the method bodies in a textual representation to the neural network to transform them into test method bodies. The test generator places these generated methods in test classes. The collection of all the new test classes is the new test suite. This test suite can be used to test the project's source code on faults.

In an ideal situation, a model is already trained. When this is not the case, then additional actions are required. Training projects are selected to train the network to generate usable tests. For instance, all training projects should use the same unit test framework. A unit test linking algorithm is used to extract training examples from these projects. The found methods and the unit test method are given as training examples to the neural network. The model can then be created by training a neural network on these training examples. A detailed example of a possible flow can be found in Figure 3.1.

Figure 3.1: Possible development flow for the test suite generator

3.1 Data collection

We set some criteria to ensure that we have useful training examples. In addition, we need a threshold on the number of training examples that should be gathered, because a large amount might not be necessary and is more time-consuming, while too few will affect the accuracy of the model.

3.1.1 Selecting a test framework

The test generator should not generate tests that use different frameworks since each framework works differently. Therefore, it is important to select projects using only one testing framework. In 2018, Oracle reviewed what Java frameworks are the most popular [Poi]. They concluded that the unit test framework Junit is the most used. We selected Junit as test framework based on its popularity. We expect that we require a large amount of test data to train the neural network model.

3.1.2 Testable projects

Unit tests that fail are unusable for our research. Our test generator should generate unit tests that test a piece of code. When a test fails, it fails due to an issue. It is not sure if the issue is a mismatch between the method's behavior and the behavior captured in the test. These tests should

not be included because a mismatch in behavior could teach the neural network patterns that prevent testing correctly. So, the tests have to be analyzed in order to filter out tests that fail. For the filtering, we execute the unit test from the projects, analyze the reports, and extract all the tests that succeed.

3.1.3 Number of training examples

For our experiment, a training set size in the order of thousands should be more likely than a training set size in the order of millions. These numbers are based on findings of comparable research, meaning studies that do not involve translation to or from a natural language. Ling et al. [LGH+16] used 16,000 annotations for training, 1,000 for development, and 1,805 for testing to translate game cards into source code. Beltramelli et al. [Bel17] also used only 1,500 web page screenshots to translate mock-ups to HTML. A larger number of training examples are used in research that translated either to or from a natural language. Zheng et al. [ZZLW17] used 879,994 training examples to translate source code to comments, and Hu et al. [HWLJ18] used 588,108 training examples to summarize code into a natural language. The reason why we need less data could be because a natural language is more complicated compared to a programming language. In a natural language, a word can have different meanings depending on its context, and the meaning of a word also depends on the position within a sentence. For example, a fish is a limbless cold-blooded vertebrate animal. Fish fish is the activity of catching a fish. Fish fish fish means that fish are performing the activity of catching other fish. This example shows that the location and context of a word have a big impact on its meaning. Another example is that a mouse could be a computer device in one context, but it could also be an animal in another context. A neural network that translates either to or from a natural language has to be able to distinguish the semantics of the language. This is not necessary for the analyzed programming language in our research. Here, differences between types of statements, as well as differences between types of expressions, are clear. Ambiguity, like in the example "fish fish fish", does not represent a challenge in our research. Therefore, it does not need to be contained in the training data, and the relationship does not need to be included in the neural network model.

3.2 Linking code to test

To train our machine learning algorithm, we require training examples. The algorithm will learn patterns based on these examples. To construct the training examples, we need a dataset with pairs of source codes of methods and unit tests. However, there is no direct link between a test and the code that it tests. Thus, in order to create the pairs, we need a linking algorithm that can pair methods and unit tests.

3.2.1 Linking algorithm

To our knowledge, an algorithm that can link unit tests to methods does not exist yet. In general, every test class is developed to test a single class. In this work, we propose a linking algorithm that uses the interface of the unit test class to determine what class and methods are under test.

We consider that all classes used during execution of a unit test could be the class under test. For every class, we determine what methods, based on their name, match the best with the interface of the unit test class. The class with the most matches is assumed to be the class under test. The methods are linked with the unit test methods that have the best match. This also means that a unit test method cannot be linked when it does not have a match with the class under test.

However, this algorithm has limitations. For example, in Listing 3.1 `stack` and `messages` are both considered the class under test. It is possible to detect that `stack` is under test when the linking algorithm is only applied to the statements that are required to perform the assertion. Backward slicing could be used to generate this subset of statements, because the algorithm can extract what statements have to be executed in order to perform the targeted statement [BdCHP10]. The subset obtained with backward slicing will only contain calls to `stack` and would therefore find the correct link. However, this algorithm will not work when asserts are also used to check if the test state is

valid to perform the operation that is tested, as now additional statements are included that have nothing to do with the method under test.

```
1 public void push() {
2       ...
3       stack = stack.push(133);
4       messages.push("Asserting");
5       assertEquals(133, stack.top());
6 }
```

Listing 3.1: Unit tests that will be incorrectly linked without statement elimination

3.2.2 Linking methods

The linking algorithm of Section 3.2.1 could be used to construct the links. However, the algorithm requires information about the source code as input. As described in Section 2.2, this could be done by analyzing the AST or bytecode.

Bytecode analysis has the advantage that concrete types can be resolved because it uses a callgraph. This enables the ability to support tests where a base class of the class under test is used. How this is done is illustrated in Listing 3.2 with a pseudo call graph in Listing 3.3. With bytecode analysis, it is possible to determine that the types of AList are only ArrayList and LinkedList, because the initialization of ArrayList and LinkedList are the only initializations that are assigned to AList in the method's call graph.

```
1 public List getList() {
2     ...
3     return (a ? new ArrayList<>() : new LinkedList());
4 }
5
6 public void push() {
7       List AList = getList();
8       assertEquals(AList.empty());
9 }
```

Listing 3.2: Hidden concrete type

```
1     push()
2     getList() [assing to AList]
3       ...
4       new ArrayList<>() [assing to tmp] || new LinkedList() [assing to tmp]
5       return tmp
6
7     AList.empty() [assing to tmp2]
8     assertEquals(tmp2)
```

Listing 3.3: Pseudo call graph for Listing 3.2

Resolving concrete types is impossible with AST analysis. It is possible to list what concrete classes implement the List interface. However, when this is used as the candidate list during the linking process, it could result in false positive matches. The candidate list could contain classes that were not used. This makes it impossible to support interfaces and abstract methods with the AST analysis.

An advantage of the AST analysis is that it does not require the project's bytecode, meaning that the project does not have to be compilable. The code could also be partially processed because only a single class and some class dependencies have to be analyzed. Partial processing reduces the chance of unsupported classes since less has to be analyzed. With bytecode analysis, all dependencies have to be present and every class that is required to build the call graph.

3.3 Machine learning datasets

The datasets for the machine learning can be prepared once enough training examples are gathered. A machine learning algorithm needs these sets in order to train the model. For our neural network, we require a test, training, and validation set. The only difference between these sets is the number of training examples and the purpose of the sets. Any training examples could be included in any of the sets as long as the input sequence is not contained in any another set. We discussed how training examples are selected in Section 3.1.

3.3.1 Training set

The training set is used for training the machine learning model. The machine learning algorithm tries to create a model that can predict the training examples as good as possible.

3.3.2 Validation set

The results achieved with the training set will be better than on unseen data, because the machine learning used this model to learn patterns. The validation set is used to validate at what time more training will negatively impact the results, as a consequence of training too strict on the data, by making the model too specific on the training set. Therefore, continuing the training will reduce the ability to generalize, which has a bad impact for making predictions on unseen data. Usually, during training, multiple models are stored. Learning is interrupted when new models have multiple times a higher loss on the validation set than previous models. Only the model with the smallest loss on the validation set is applied for making predictions on unseen data. It could be the case that during training, multiple times a higher loss is noticed what will decrease again later on. For each dataset, it should be determined at what point training could be stopped without risking that a new lowest point is skipped.

3.3.3 Test set

An additional dataset is required to evaluate how well the model generalizes and if the model is better than other models. The generalization is tested by evaluating how thoroughly the model performs on unseen data. This set, in combination with metrics, can be used to calculate a score. This score can be compared with the scores of other models to determine what model is the best. Using the validation set for the comparison is unsuitable because the model is optimized for this dataset and this does not give information on how well the results are in general. The test set is just another subset of all training examples which is not yet used for a different purpose. The training examples cannot be contained in other sets so that they could not possessively influence the score that is calculated.

Chapter 4

Evaluation Setup

For the evaluation of our approach, we introduce in total three goal metrics that indicate how far we are from generating working code to the ability to find bugs. However, the results of the metrics could be biased because we are using machine learning. The nodes of a neural network are initialized with random values before training starts. From this point, the network is optimized so that it can predict the training set as good as possible. Different initial values will result in different clusters within the neural network, what impacts the prediction capabilities. This means that metric scores could be due to chance. In this chapter, we will address this issue.

We created a baseline with selected test projects to enable comparisons of our results with the generated test suite of alternative test generators and manually written test suites. This baseline can be used when unit tests can be generated. Otherwise, we do not have to use these projects. The evaluation of any method is fine because we do not have to be able to calculate the effectiveness of the tests. In our research, for **RQ**1 we perform multiple experiments with different configurations (different datasets and different network configurations). We have to prove that a change in the configuration will result in a higher metric score. For **RQ**2 we prove that with compression the accuracy will increase, and the required time will decrease.

4.1 Evaluation

The test suite capability should be evaluated if the generator can generate test code. Nevertheless, when the test generator is in a phase where it is unable to produce valid tests, a simple metric should be applied which does not test the testing capability. However, it would qualify how far we are from generating executable code because code that is not executable is unable to test something. This set of metrics enables us to make comparisons over the whole phase of the development of the test generator.

4.1.1 Metrics

The machine learning models can be compared to its ability to generate parsable code (parsable rate), compilable code (compilable rate), and code that can detect faults (mutation score). The parsable rate and compilable rate measure the test generator's ability to write code. The difference between these two is that compilable code is executable, while this is not necessarily true for parsable code. The mutation score measures the test generator's test quality.

These metrics should be used in different phases. The mutation score should be used when the model can generate working unit tests to measure the test suite effectiveness. The compilable code metric should be used when the machine learning model is aware of the language's grammar to measure the ability of writing working code. If the mutation score and compilable code metric cannot be used, the parsable code metric should be applied. This measures how well the model knows the grammar of the programming language.

4.1.2 Measurement

The parsable rate can be measured by calculating the percentage of the code that can be parsed with the grammar of the programming language. The parsable rate can be calculated by dividing the amount of code that parses with the total amount of analyzed code. The same calculation as for parsable rate can be applied for the compilable rate. However, instead of parsing the code with the grammar, the code should be compiled with a compiler.

The mutation score is measured by a fork of the PIT Mutation Testing[1]. This fork is used because it is combining multiple mutation generations, which leads to a more realistic mutation score [PMm17].

4.1.3 Comparing machine learning models

A machine learning model depends on random values. When we calculate metrics for a generated test suite, the results will be different when we use another seed for the random number generator.

When we want to compare results with the metrics from Section 4.1, we have to cancel out the effect of the random numbers. We do this by performing multiple experiments instead of running single experiments. Then, we use statistics to test for significant differences between the two groups of experiments. If there is a significant difference, we use statistics to prove that one group of results is significantly better than the other group of results.

4.2 Baseline

The effectiveness of a project's manually written test suite and of automatically generated test suites are used as the baseline. Only test suit generators that implement search-based testing and random testing are considered, because many open-source tools are available for these methods and they are often used in related work. We use Evosuite[2] for search-based testing and Randoop[3] for random testing, as these are the highest scoring open-source test suite generators in the 2017 SBST Java Unit Testing Tool Competition in their respective categories [PM17].

Once tests can be generated, we will evaluate our approach on six projects. These projects are selected based on a set of criteria. State-of-the-art mutation testing tools (see sec. 4.1) must support these projects, and the projects should have a varying mutation coverage. The varying mutation coverage is needed to evaluate projects with a different test suite effectiveness. This way we could determine how much test suite effectiveness we can contribute to projects with a low, medium, and high test suite effectiveness. We divided projects with a test suite effectiveness around 30% into the low category, projects around 55% into the medium category, and around 80% into the high category. These percentages are based on the mutation coverage of the analyzed projects. The selected projects can be found in Table 4.1.

Table 4.1: Mutation coverage by project

Project	**Java files**	**Mutation coverage**	**Category**
Apache Commons Math 3.6.1	1,617	79%	high
GSON 2.8.1	193	77%	high
La4j 0.6.0	117	68%	medium
Commons-imaging 1.0	448	53%	medium
JFreeChart 1.5.0	990	34%	low
Bcel 6.0	484	29%.	low

[1] https://github.com/pacbeckh/pitest
[2] http://www.evosuite.org/
[3] https://randoop.github.io/randoop/

Chapter 5

Experimental Setup

How a test suite generator can be developed in general was discussed in Chapter 3. The chapter contains details on how training examples can be collected and explains how these examples can be used in machine learning models. How the test generator can be evaluated is discussed in Chapter 4. Several metrics are included, and a baseline is given. This chapter gives insight on how the test suite generator is developed for this research. We included additional criteria to simplify the development of the test generator. For instance, we only extract training examples from projects with a dependency manager to relieve ourselves of manually building projects.

5.1 Data collection

Besides the criteria mentioned in Section 3.1, we added additional requirements to the projects we gathered to make the extraction of training data less time-consuming. We also used a project hosting website to make the process of filtering and obtaining the projects less time-consuming.

5.1.1 Additional project criteria

It is time-consuming to execute the unit test suits for all projects manually. A dependency manager could be used to automate this for these projects. Projects that use a dependency manager use a configuration file that defines the requirements on how to build, and most of the time also on how to execute the project's unit tests. Therefore, we expect that only using projects that have a dependency manager will make this task less time-consuming. We only consider Maven[1] and Gradle[2] because these are the only dependency managers that are officially supported by JUnit [Juna].

5.1.2 Collecting projects

We use the GitHub[3] platform to filter and obtain projects. GitHub has an API that can be used to obtain a list of projects that meet specific requirements. However, the API has some limitations. Multiple requests have to be done to perform complex queries. Each query can show a maximum of 1,000 results which have to be retrieved in batches of maximum 100 results, and the number of queries is limited to 30 per minute [Git]. To cope with the limit of 1,000 results per query, we used the project size criteria to partition the projects into batches with a maximum of 1,000 projects per batch. For our research, we need to make the following requests:

- As mentioned, we have to partition the projects to cope with the limitation of maximum 1,000 results per query. The partitioning is performed by obtaining the Java projects starting from a certain project size and are obtained by performing 10 requests to get the results in batches of 100. This step is repeated with an updated start size until all projects are obtained.

[1] https://maven.apache.org/
[2] https://gradle.org/
[3] https://github.com/

- For each project, a call has to be made to determine if the project uses JUnit in at least one of their unit tests. This can be done by searching for test files that use JUnit. The project is excluded when it does not meet this criterion.
- Additional calls have to be made to determine if the project has a build.gradle (for Gradle projects) with the JUnit4 dependency or a pom.xml (for Maven projects) with the JUnit 4 dependency. An extra call is required for Maven projects to check if it has the JUnit 4 dependency. The dependency name ends either with junit4, or is called JUnit and has version 4.* inside the version tag.

The number of requests needed for each operation could be used to limit the total number of requests required. This can improve the total time required for this process. In our case, we expect that it is best to check first if a project is a Gradle project before checking if it is a Maven project, because more requests are required for Maven projects.

In conclusion, to list the available projects, one request has to be made for every 100 projects. Each project requires additional requests: one request to check if a project has tests, one additional request for projects that use Gradle, and at most two extra requests for projects that use Maven.

So, to analyze **n** projects, at minimum $n * ((1/100) + 1)/30$ and at maximum $n * ((1/100) + 4)/30$ minutes are required.

5.1.3 Training data

With the GitHub API mentioned in Section 5.1.2, we analyzed 1,196,899 open-source projects. From these projects, 3,385 complied with our criteria. We ended up with 1,106 projects after eliminating all projects that could not be built or tested. These projects have in total 560,413 unit tests. These unit tests could be used to create training examples. However, the total amount of training data could be less than the number of unit tests, because the linking algorithm might be unable to link every unit test (as described in Section 3.2.1).

5.2 Extraction training examples

The training example extraction can be performed with the linking algorithm described in Section 3.2.1 by using the analysis techniques described in Section 3.2.2. For the extraction, we use the training projects mentioned in Section 5.1.1.

As mentioned in Section 5.1.3, we have to analyze a large number of training projects. For our research, we lack the infrastructure to process all projects within a reasonable time. To make it possible to process everything in phases, we introduced a queue. This enables us to interrupt processing at any time and continue it later without skipping any project. As mentioned in Section 3.1, we should only include unit tests that succeeded. Thus, we fill the queue based on test reports. When all test reports are contained, we start linking small groups of tests until everything is processed.

5.2.1 Building the queue

The queue is used by both bytecode analysis and AST analysis. All the unit tests of each training project are contained in the queue. We structured the queue in a manner so it contains all the information required for these tools. For instance, we have to store the classpath to perform bytecode analysis, the source location to perform AST analysis, and we have to store the unit test class name and unit test method name for both bytecode and AST analysis.

The source location and classpath can be extracted based upon the name of the test class. For each test class, there exists a ".class" and ".java" file. The ".class" file is inside the classpath, and the ".java" file is inside the source path. The root of these paths can be found based on the namespace of the test class. Often, one level up from the classpath, there is a folder with other classpaths. If this is the case, then usually there is a folder for the generated classes, test-classes, and the regular classes. The classes used in the unit test could be in any of these folders. Therefore, all these paths have to be included to perform a complete analysis.

The test class and test methods can be extracted based on the test report of each project. The test report consists of test classes with the number of unit tests that succeeded and how many failed or did not run for any other reason. From the report we cannot differentiate between test methods that succeeded or failed. So, we only consider test classes for which all test methods succeeded. The test methods from the test class can be extracted by listing all methods with a `@Test` annotation inside the test class.

5.3 Training machine learning models

The last step is to train the machine learning models. In order to train the machine learning model, we need to obtain a training and validation set. To evaluate how well the model performs, we use a test set. We divided the gathered training examples into these three sets. We use 80% of the data for the training, 10% for the validation, and 10% for the test set.

However, the quality of a machine learning model does not only depend on the used training data. It also depends on the data representation. In this section, we introduced four views, namely tokenized view, compression, BPE [SHB15], and AST. From these views, the tokenized view resembles the textual form of the data the most. The other views modify the presented information.

5.3.1 Tokenized view

With the tokenized view, every word and symbol in the input and output sequence is assigned to a number. Predictions are performed based on these numbers instead of on the words and symbols. This method is also used with natural language processing (NLP) [SVL14]. An example of the tokenized view is displayed in Figure 5.1. Code is given as input, parsed, and each symbol is assigned a number.

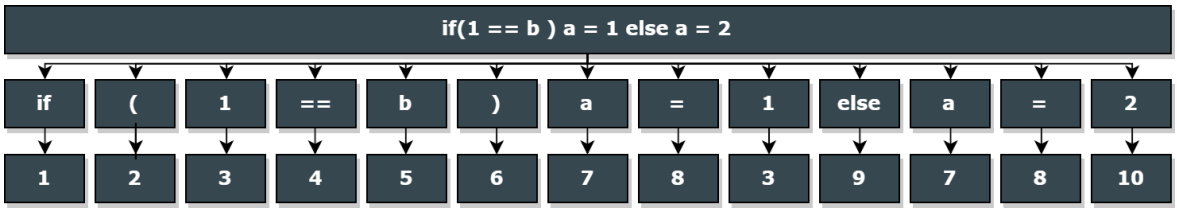


Figure 5.1: Example of tokenized view

5.3.2 Compression

Neural networks perform worse when making predictions over long input [BCS+15]. Compression could improve the results by limiting the sequence length. For this view, we used the algorithm proposed by Ling et al. [LGH+16]. They managed to reduce code size without impacting the quality. On the contrary, it improved their results.

This view is an addition to the tokenized view, described in Section 5.3.1. The view compresses the training data by replacing token combinations with a new token. An example is displayed in Figure 5.2. The input code is converted to the tokenized view, and a new number replaces repeated combination of tokens. Additionally, in an ideal situation, it also could improve the results. For example, when learning a pattern on a combined token is easier than learning them on a group of tokens.

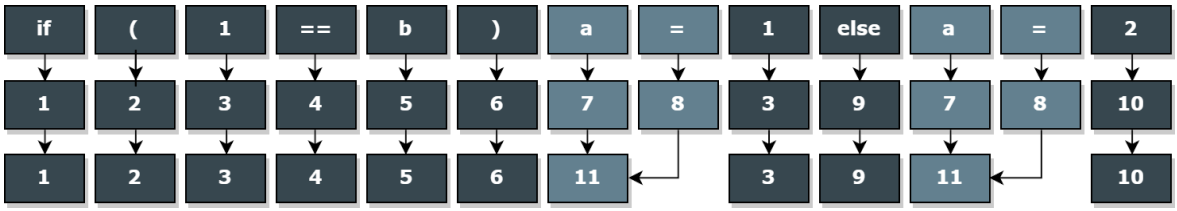


Figure 5.2: Example of compression view

5.3.3 BPE

The tokenization system mentioned in Section 5.3.1 generates tokens based on words and symbols. Nevertheless, the words somehow belong together. This information could be usable during prediction and can be given to the neural network by using BPE. BPE introduces a new token "@@ " to connect subsequences. The network learns patterns based on these subsequences, and they are also applied to words that have similar subsequences. Figure 5.3 shows an example of this technique applied to source code. In this figure, the sequence "int taxTotal = taxRate * total" is converted into "int tax@@ Total = tax@@ Rate * total" so that the first "tax" in connected with Total and the last "tax" is connected with Rate.

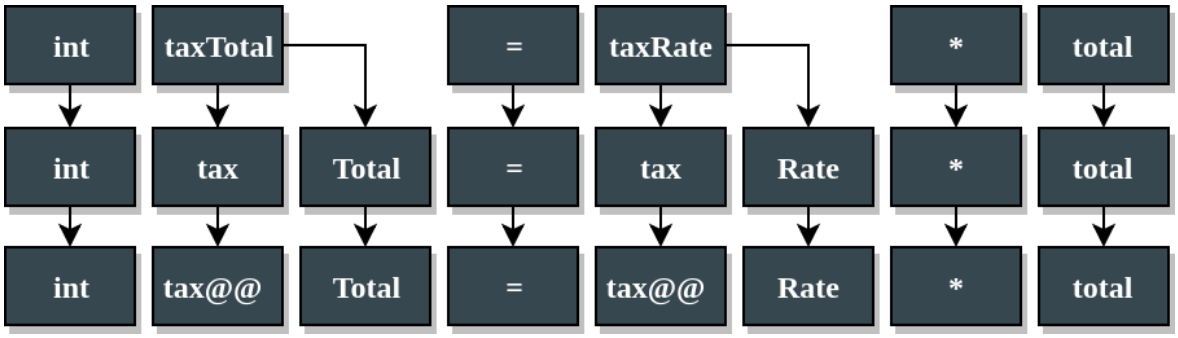


Figure 5.3: Example of BPE view

5.3.4 Abstract syntax tree

When we look at the source code, we can see where the start and stop of an if statement is. So, for programmers, it makes sense. However, this structure is not clear for a neural network. A neural network will perform better when it can see this structure. The grammar of the programing language can be used to add additional information. This information can be added by transforming source code into an AST [DUSK17] and print it with an SBT [HWLJ18] (mentioned in Section 2.3). The SBT will output a textual representation that reflects the structure of the code. An example of an AST representation is shown in Figure 5.4 and the textual representation in Listing 5.4. These examples display how an if statement is converted to a textual representation. The textual representation is created by traversing the AST [HWLJ18]. Each node is converted to text by outputting an opening tag for the node, followed by the textual representation of the child nodes (by traversing the child nodes), and finally by outputting a closing tag.

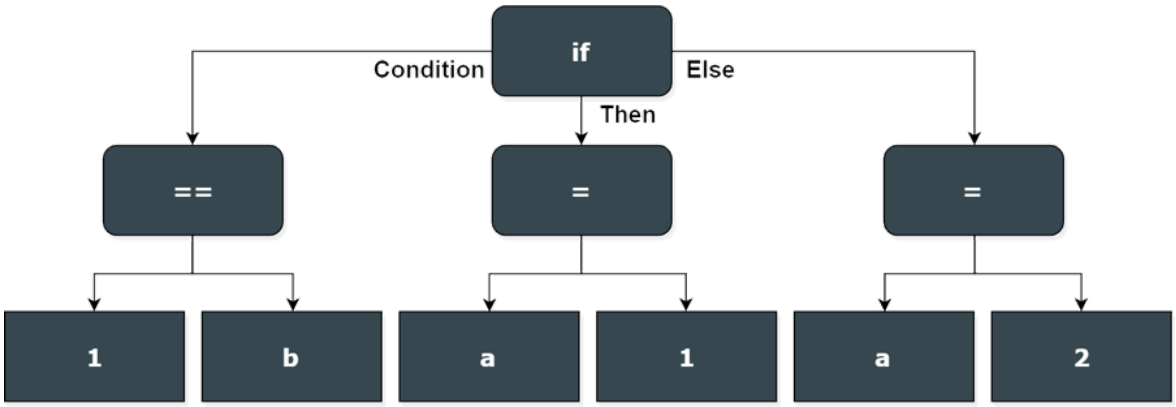


Figure 5.4: Example of AST view

```
(ifStatement
    (assign (variable a)variable (number 1)number)assign
    (equals (number 1)number (variable b)variable)equals
    (assign (variable a)variable (number 2)number)assign
)ifStatement
```

Listing 5.4: Textual representation of the AST from Figure 5.4

5.4 Experiments

In Section 5.3 we explained how the machine learning models can be trained. In this section, we are going to list all the experiments that we are going to perform. First, an ideal set of training examples and basic network configuration are selected where an as high as possible metric score can be achieved on. For **RQ**1, models with different data representations mentioned in Section 5.3 (SBT, BPE, and Compression) are trained. Additional, a model that uses a network configuration of related research is trained and a model with an optimized configuration is also trained. For **RQ**2, a model is trained to measure the time required to train various levels of compression and a model is trained to evaluate the development of accuracy when compression is applied.

To evaluate which experiment has the best results, we have to compare their results. In Section 4.1.3 we stated that we do test this with statistics. In this section, we go into more detail.

5.4.1 The ideal subset of training examples and basic network configuration

We created a basic configuration for our experiments. This configuration is used as the basis for all experiments. It is important that this configuration contains the best performing dataset. Otherwise, it is unclear if bad predictions are due to the dataset or the newly used method.

For our experiments, we use the Google seq2seq project[4] from April 17, 2017. We use attention in our models, as attention enables a neural network to learn from long sentences [VSP+17]. With attention, a soft search on the input sequence is done during predicting in order to add contextual information. This should make it easier to make predictions on large sequence sizes. The network has a single decode and encode layer of 512 LSTM nodes, has an input dropout of 20%, uses the Adam optimizer with a learning rate of 0.0001, and has a sequence cut-off at 100 tokens.

[4] https://github.com/google/seq2seq

5.4.2 SBT data representation

SBT can be used as data representation for the training data. SBT adds additional information to the training data by using the grammar of the programming language. Hu et al. [HWLJ18] achieved with this technique better results when translating source code into a textual representation instead of source code into text. Applying the SBT to our experiment is beneficial because it also could improve our results. We use a seq2seq neural network with LSTM nodes for this experiment because this setup is the most comparable to their setup.

However, when we train our model on how an SBT can be translated into another SBT, it will output an SBT representation as the prediction. Thus, we need to build software that can convert the SBT back into code.

In the research where SBT is proposed, code was converted into text [HWLJ18]. There was no need to convert the SBT back. We had to make a small modification to support back-translation. We extended the algorithm with an escape character. In the AST information, every parenthesis and underscore are prepended with a backslash. This makes it possible to differentiate between the tokens that are used to display the structure of the AST and the content of the AST.

For the translation from the SBT to the AST, we developed an ANTLR[5] grammar that can interpret the SBT and can convert it back to the original AST. For the conversion from the AST to code, we did not have to develop anything. This was built-in our AST library (JavaParser[6]). For the validate of our software, we converted all our code pairs described in Section 6.1.3 back from the SBT to code.

5.4.3 BPE data representation

BPE is often used in NLP and is used to achieve a new state-of-the-art BLEU score [SHB15]. They improved up to 1.1 and 1.3 BLEU, respectively [SHB15]. We could use BPE in our research to include links between tokens.

5.4.4 Compression (with various levels) data representation

Ling et al. [LGH+16] applied compression to generate code based on images. They found that every level of compression increased their results. The compression level of 80% showed the best results. In our research, we also translate to code. It is possible that compression also works in our dataset.

5.4.5 Different network configurations

During this experiment, we tried to find the optimal neural network type and neural network configuration. We experiment with different numbers of layers, more or less cells per layer, and with different types of network cells.

In addition, we also evaluated the network settings used by Hu et al. [HWLJ18], as similar as possible. Our network size and layers are comparable to their network, but our dropout and learning rate are not. We did not look at sequence length, because it is clear what length they used. Compared to our configuration, Hu et al. used a dropout of 50% while we used 20%. In addition, they used a learning rate of 0.5 while we used a rate of 0.0001. The values we utilized are the defaults of Google seq2seq project for Neural Machine translation.

5.4.6 Compression timing

To assess the training speed of the neural network, we performed experiments with compression. To make a comparison, we measured the time needed to do 100 epochs between no compression and compression level 1, 2, and 10.

[5] http://www.antlr.org/
[6] https://javaparser.org

5.4.7 Compression accuracy

We evaluated a compression level that is close to the original textual form to test the impact on accuracy when using compression. When compression caused the model to not generate parsable code, we looked at the development of loss over time. The loss represents how far the validation set is from predicting the truth. We can conclude that the model is not learning when the loss increases from the start of the experiment. This would mean that compression does not work on our dataset. We have only used the compression level 1 dataset for this experiment.

5.4.8 Finding differences between experiments

For **RQ**1, we first perform all experiments to create an overview of the results. Then, we select the experiments which we want to test whether there is a significant different result. We only do this for experiments that improved our previous scores. As already mentioned in Section 4.1.3, we perform the same experiments multiple times to enable statistical analysis on those groups of results. For **RQ**2, we evaluate the effect of accuracy and speed during compression. For the evaluation of speed, we measured 30 times the time that was needed to perform 100 epochs when applying no compression and compression level 1, 2, and 10. The evaluation of the accuracy is performed with the metrics discussed in Section 4.1. For this experiment, we ran five tests with a baseline without any compression, and with various levels of compression. However, when compression prevents the model from learning, we analyze the evolution of the loss on the validation set (which is used to determine when training the model further should be stopped) as mentioned in Section 5.4.7. The experiment was repeated five times.

The first step in comparing experiments is to prove that there is a significant difference between the results of the experiments. If there is a difference, we use statistics to evaluate what groups introduced the difference and what relation these differences have.

For evaluating this difference, we use hypothesis testing with analysis of variance (ANOVA), with h0: there is no significant difference between the experiments; h1: there is a significant difference. We use an alpha-value of 0.05 to give a direction to the most promising setup for our research. As there is only one variable in the data, we use the one-way version of ANOVA. For **RQ**1, the variable is the dataset, and for **RQ**2, the variable is either the level of compression when testing speed, the dataset when testing the accuracy, or, when accuracy cannot be measured, the epoch is the variable when evaluating the loss.

The experiment is repeated for five times at least. So, our experiments are groups of results. However, each run in an experiment depends on a variable. This is the epoch number for the speed measurement of **RQ**2. For all other experiments, this is a random value. We need ANOVA for repeated-measures to analyze these independent groups.

Nevertheless, when applying ANOVA for repeated-measures, there is the assumption that the variances between all groups have to be equal (sphericity) [MB89]. We violate this in our experiments. For instance, when we use a different random value, there is a different spread in outcomes because it depends on another variable. When this violation is made, this has to be corrected. We use the Greenhouse-Geisser correction [Abd10] to do this. The correction is done by adjusting the degrees of freedom in the ANOVA test to increase the accuracy of the p-value.

When we apply ANOVA with the correction, we can evaluate if we can reject the h0 hypothesis (no difference between the groups). When we can reject it, then it tells us there is a difference between the groups, without knowing where. To know what caused the difference, we need to do an additional test. This additional test is performed with the Tukey's multiple comparison test. This test is used to compare the differences between the means of each group. For this test, a correction is applied to cancel out the effect of multiple testing. The correction is needed because when more conclusions are made, the more likely an error occurs. For example, when performing five experiments with an error rate of X%, there is a change of 5X% that an error is made within the whole test.

When we know what group is different, we still have to find out which group of results is better. So, we want to prove that the mean of one group is greater than the means of another group. To test this, we perform a one-tail t-test on each individual group. We adjust the alpha according to how many tests we perform on the same dataset to cancel out the effect of repeated measures. When

performing X tests, we divide the alpha by X to keep the total maximum error rate at 5%.

Chapter 6

Results

In Chapter 3 and Chapter 5, it is discussed how our experiments are performed in order to answer **RQ**1 and **RQ**2. In this chapter, we report on the obtained results. In Addition, we also report on techniques used to generate training sets. This does not directly answer a research question. However, the training examples are both used to train models for both **RQ**1 and **RQ**2.

6.1 Linking experiments

In this section, we report on the linking algorithm described in Section 3.2.1. We mentioned in 3.2.2 that both bytecode analysis and AST analysis use different principles, that might have a positive impact on their linking capabilities. We run both bytecode analysis and AST analysis on the queue mentioned in Section 5.2. We assessed how many unit tests are supported by both techniques, how many links both techniques can make on the same dataset, how many links both techniques can make in total, and what contradictory links were made between both techniques.

6.1.1 Removing redundant tests

The projects in our dataset, described in Section 5.1.3, contains 560,413 unit tests in total. However, there are duplicate projects in this dataset. To perform a reliable analysis, we have to remove duplicate unit tests. Otherwise, when a technique supports an additional link, it could be counted as more than one extra link. The duplicates are removed based on their package name, class name, and test name. There remain 175,943 unit tests after removing the duplicates.

Unfortunately, this method will not eliminate duplicates when the name of the unit test, class, or package is changed, but it will remove duplicate tests when they have the same package, class and method name by coincidence. Thus, the algorithm removes methods with the same naming even when the implementation is different.

6.1.2 Unit test support

In Section 3.2.2, we claimed that AST analysis should be able to support more unit tests because it has to analyze less. We assessed this by evaluating how many tests are supported by both techniques. The results can be found in Figure 6.1.

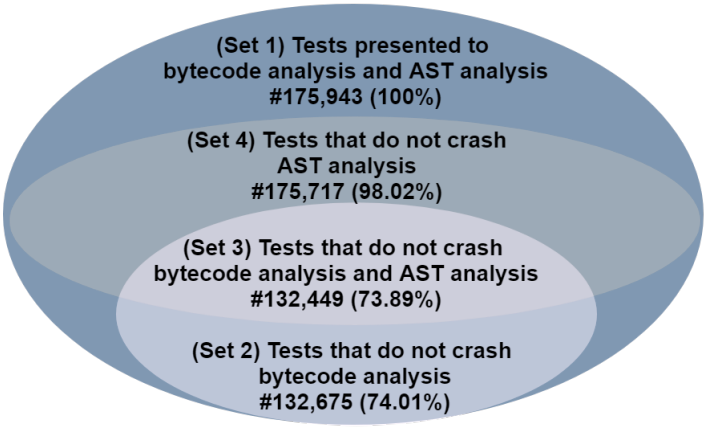


Figure 6.1: Supported unit test by bytecode analysis and AST analysis

6.1.3 Linking capability

In Section 3.2.2, we claimed that bytecode analysis should be able to create more links, since it has better type information than AST analysis. Additionally, AST analysis should be able to create more links because it has to analyze less compared to bytecode analysis.

In Figure 6.2, AST analysis and bytecode analysis are both applied on a set of tests that are supported by both methods. In Figure 6.3a and Figure 6.3b, an overview is given of all links that these methods could make.

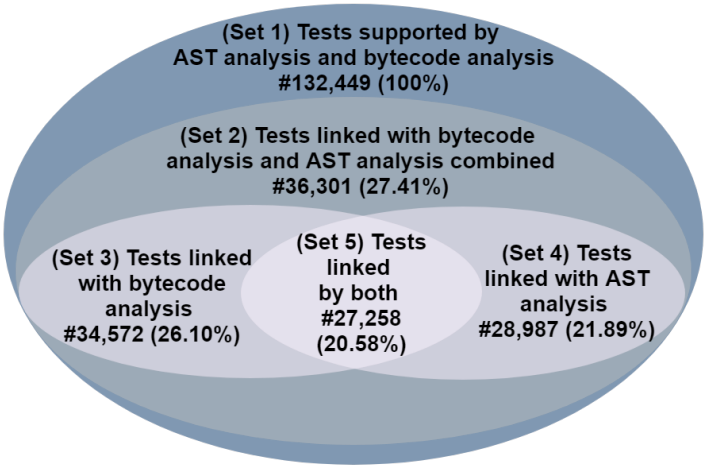
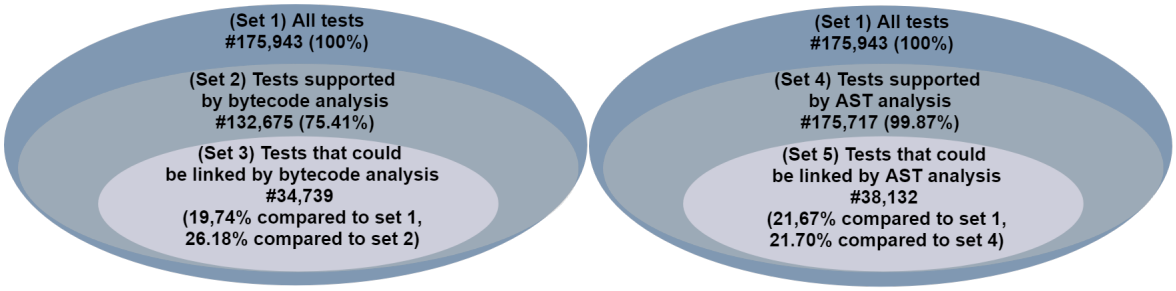


Figure 6.2: Unit test links made on tests supported by bytecode analysis and AST analysis



(a) Total link with bytecode analysis

(b) Total link with AST analysis

Figure 6.3: Total link with AST analysis and bytecode analysis

In Figure 6.2 it is shown that bytecode analysis can create more links compare to AST analysis. In Figure 6.3a and Figure 6.3b, it is shown that AST analysis could create more links as it had support for more tests.

6.1.4 Total links

In section 6.1.1, we mentioned that we eliminated duplicates in our dataset to perform a valid analysis. We also mentioned that the algorithm to remove duplicates properly will remove too much. A perfect but time-consuming way to remove duplicates would be to remove duplicate based on the unit test code and method code. When we apply this method, we could link 52,703 tests with a combination of bytecode analysis and AST analysis. With only bytecode analysis, we were able to link 38,382 tests, and with AST analysis we managed to link 44,412 tests.

6.1.5 Linking difference

From the 36,301 links that were made via bytecode analysis and AST analysis (Figure 6.2), 234 unit tests were linked to other methods. In this section, we report on these differences.

Concrete classes

In 83 of the cases, a concrete class was tested. Bytecode analysis, unlike AST analysis, could link these because it knows that they were used. AST analysis tries to match the interfaces without having knowledge of the real class under test and incorrectly links it to another class that has some matching names by coincidence.

Additionally, in 37 other cases, a class was tested that overrides methods from another class. AST analysis lacks the information about what class is used when a base class is used in the type definition. So again, AST analysis fails to create a correct link, due to its lack of awareness of the real class under test.

However, in 25 cases there were multiple concrete classes tested within one unit test class. These were included in the test class, since they all have the same base type. Bytecode analysis will treat every concrete class as a different class and will divide all the matches among all classes. With bytecode analysis, an incorrect link was made to another class that had a similar interface. AST analysis was not aware of the concrete classes and linked the tests to the base class. Additionally, in 6 other cases, bytecode analysis failed to link to the correct concrete class that was tested. AST analysis linked these tests to the base class.

Subclasses

For 24 unit tests, the method that was tested was within a subclass or was a lambda function. Bytecode analysis could detect these, because these calls are included in the call graph. We did not support this in the AST analysis.

Unfortunately, this also has disadvantages. In 14 other cases mocking was used. Bytecode analysis knew that a mock object was used and linked some tests to the mock object. However, the mock object is not tested. The unit tests validated, for example, that a specific method was called during the unit test. Bytecode analysis incorrectly linked the test with the method that should be called and not the method that made the call. AST analysis did not recognize the mock objects, and therefore it could link the test to the method that was under test.

Unclear naming

The naming of a unit test does not always match the intent of the unit test. In 13 cases, multiple methods were tested in the unit test. We only allowed our analysis tools to link to one method. Both AST analysis and bytecode analysis were correct, but they selected a different method. In 19 cases for AST analysis and 3 cases for bytecode analysis, an incorrect method was linked because of unclear naming. In 8 cases, it was not clear what method performed the operation that was tested. Multiple methods could do the operation. In 2 cases, it was not clear what was tested.

6.2 Experiments for RQ1

All executed experiments are combined into the roadmap shown in Figure 6.4. Experiments are built on-top of the configuration of a previous category when it improves the latest results. We start with a training a model on all training examples, followed by the experiments discussed in Section 5.4.8. These experiments can be categorized into simplifications, combination of simplifications, different data representations, and different network configurations.

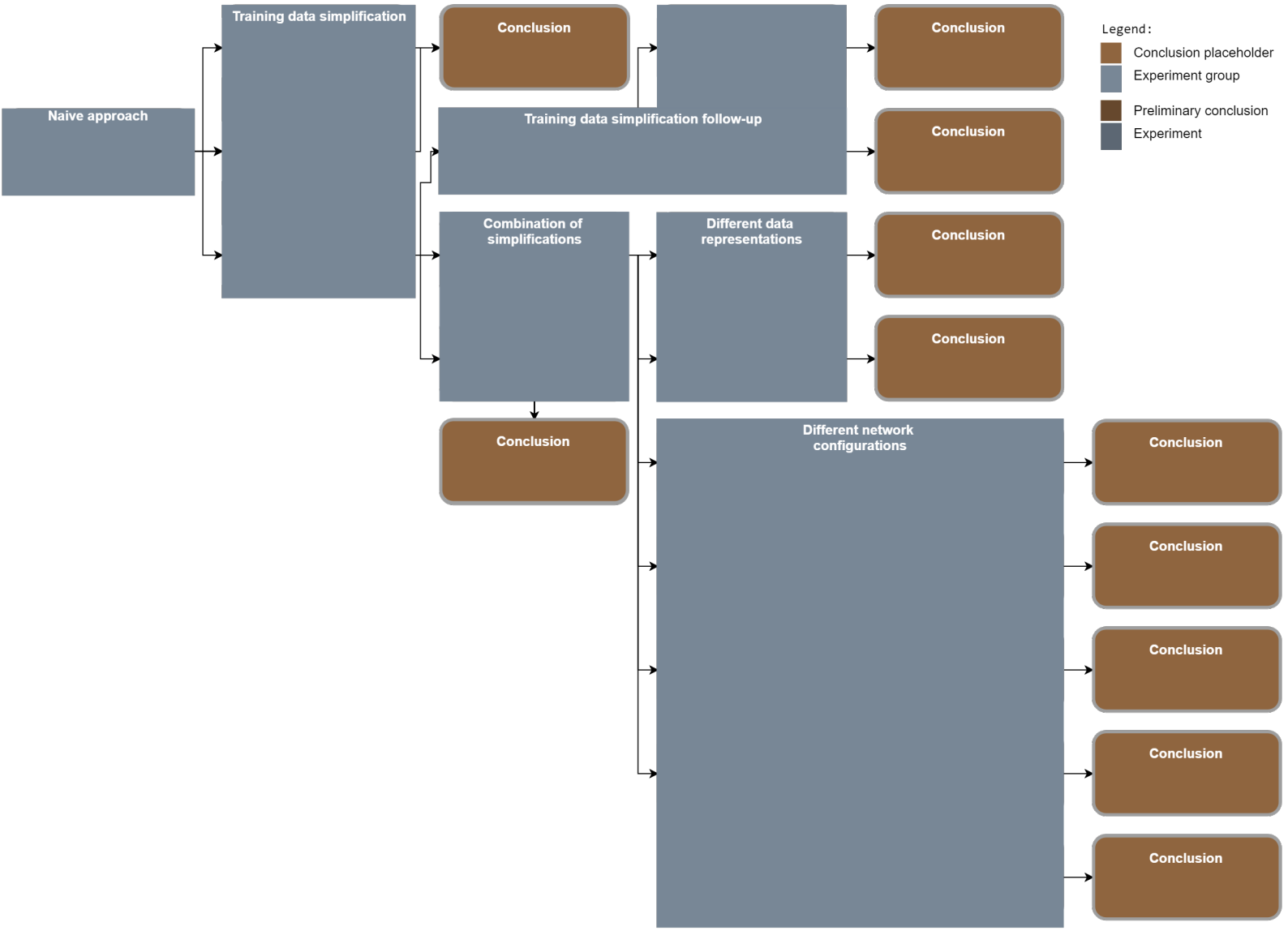


Figure 6.4: Roadmap of all experiments

6.2.1 Naive approach

First, we trained a neural network on all our training examples. The generated model could predict a large number of valid testing code for the experiment's test set. Unfortunately, the generation of valid testing code was not due to the model knowing how to generate tests. It was because of incorrectly dividing the training examples over the test, training, and validation set. In our dataset, methods were associated to multiple tests. Each test from these groups was divided over the training, validation, and test set. So, during training, the neural network already saw how some methods should be translated to a unit test that was asked during evaluation. During evaluation, the neural network just copied the unit test from the training phase, what explains why it could generate valid tests.

Division of training examples

Our dataset consists of methods that are linked with multiple unit tests. Therefore, we have **n** methods, and each of these **n** methods can be linked to **m** different unit tests. This means that

multiple training examples share the same method code, but have different test code. Pairs with the same method code were divided over the training, test, and validation set. Because of this division, the neural network already knew how some of the methods, those which were used during evaluation, could be converted to test code. This made it very easy for the network to generate tests, as it was already taught how to generate answers in some cases. This led to a false evaluation score.

To resolve this issue, examples that have the same method body were grouped and assigned to a single dataset. This prevented that code with the same implementation is assigned across the test, validation, and training set. However, because groups are assigned to sets instead of single tests, it could happen that the sets will not have the intended size. Nevertheless, a couple more or less training examples in a set will not make a difference. Therefore, we performed our experiments with the sets even if they were of different sizes.

Naive approach with correct divided training examples

After correctly dividing the training examples, the new model was trained. With this model, we only managed to generate parsable code. We were unable to generate any code that is compilable or tests something. As shown in Table 6.1, we were able to generate 15.45% parsable code.

Table 6.1: Details on naive approach experiment

Result	**Result value**
Experiment short name	S1
Training examples	52,588
Examples in training set	42,070
Examples in validation set	5,259
Examples in test set	5,259
Unique methods in test set	1,346
Parsable predictions on test methods	208
Parsable score	15.45% (208/1,346)

Experiment results

We expect the cause of the low parsable code rate is due to the high complexity of the training data. During manual analysis, we found the following complexities:

- Training examples were cut-off to a maximum of 100 tokens, because of our configured sequence length
- There are training examples that use more complex language features such as inheritance and default methods
- The training set contains token sequences that are used only once

An overview of the current results are shown in Figure 6.5.

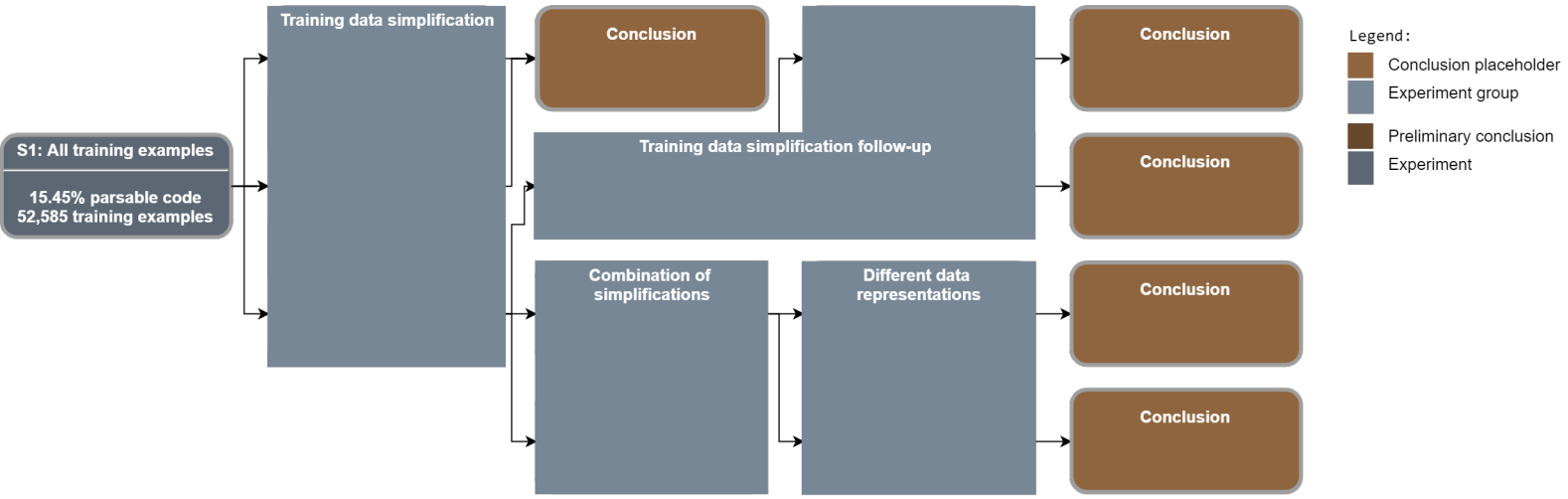


Figure 6.5: Experiments within the native approach group

6.2.2 Training data simplification

In these experiments, we are trying to reduce the complexity that was encountered with the naive approach. A list of complexities can be found in Section 6.2.1. We tried to train a model on less complex training examples, training examples that have common subsequences, and training examples with a limited size.

Less complex training data

It can be possible that the code is too complex to generate code. We filtered the training examples based on code complexity. We excluded all training examples that test classes which use inheritance or default methods. This is done by selecting only training examples that were linked by the AST analysis. AST analysis does not support these features. In addition, we only selected examples that were also linked by bytecode analysis. This will also remove all the incorrect links that were discussed in section 6.1.5. With these criteria, we achieved 14.46% parsable code. More details of the experiment can be found in Table 6.2.

Table 6.2: Details on experiment with less complex training data

Result	**Result value**
Experiment short name	S2
Training examples	30,485
Examples in training set	24,378
Examples in validation set	3,049
Examples in test set	3,058
Unique methods in test set	802
Parsable predictions on test methods	116
Parsable score	14.46% (116/802)
Last best model	*S1*
Last best model parsable code	*15.45%*

Training examples with overlapping subsequences

Many training examples use unique token combinations. It can be the case that the network is unable to make predictions on sequences that do not occur in other examples. To overcome this problem, we used the compression algorithm from Ling et al. [LGH+16]. The training examples used, are

those which are reduced by more than 80% when they are compressed ten times. This will reduce the number of training examples that do not share token sequences with other training examples. We achieved 12.41% parsable code with the training examples that have common subsequences. More details on the experiment can be found in Table 6.3.

Table 6.3: Details on experiment with common subsequences

Result	**Result value**
Experiment short name	S3
Training examples	47,178
Examples in training set	37,740
Examples in validation set	4,720
Examples in test set	4,718
Unique methods in test set	975
Parsable predictions on test methods	121
Parsable score	12.41% (121/975)
Last best model	S1
Last best model parsable code	15.45%

Limited size

In our default network configuration, we cut off sequences at a sequence length of 100 tokens. To prevent the training examples being cut off, we filtered the examples on a maximum size of 100 tokens. With this criterion, we achieved 69.79% parsable code. More details on the experiment can be found in Table 6.4.

Table 6.4: Details on experiment with maximum sequence size 100

Result	**Result value**
Experiment short name	S4
Training examples	11,699
Examples in training set	9,358
Examples in validation set	1,170
Examples in test set	1,171
Unique methods in test set	374
Parsable predictions on test methods	261
Parsable score	69.79% (261/374)
Last best model	S1
Last best model parsable code	15.45%

Experiment results

As shown in Figure 6.6, only the experiment where we limited the sequence size results in a higher percentage of parsable code.

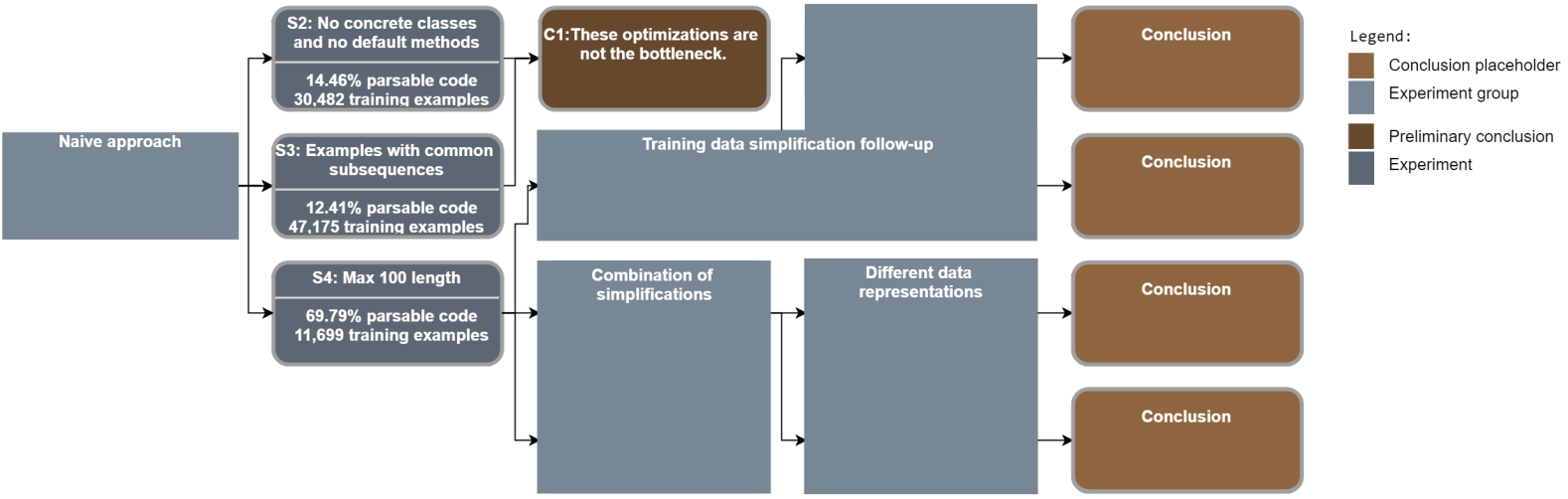


Figure 6.6: Experiments within the training data simplification group

6.2.3 Training data simplification follow-up

We noticed an improvement from 15.45% to 69.79% parsable code when we decreased the sequence length of the training examples. However, the improvement could be due to the smaller number of training examples or because the network trains better on small sequence size. We trained additional models to evaluate what caused the different results.

Increase sequence length to 200 with limited training examples

We increased the sequence length to analyze whether the decreased sequence length resulted in more parsable code. In this experiment, we increase the maximum allowed sequence length to 200 and accepted examples with a maximum sequence length of 200 tokens. We kept the same number of training examples, so we can monitor what happens when we allow longer sequences. As shown in Table 6.5, we achieved 31.78% parsable code with this configuration.

Table 6.5: Details on experiment with maximum sequence size 200 and limited number of training examples

Result	**Result value**
Experiment short name	S5
Training examples	11,699
Examples in training set	9,359
Examples in validation set	1,170
Examples in test set	1,170
Unique methods in test set	428
Parsable predictions on test methods	136
Parsable score	31.78% (136/428)
Last best model	S4
Last best model parsable code	69.79%

Increase sequence length to 200 and use more training example

The decreased parsable rate when using a sequence length of 200 and the same number of training examples as the experiment with sequence length 100, could be caused by too few training examples to make a prediction over a larger sequence. To evaluate this, we increased the number of training

examples. With these criteria, we achieved a 32.18% parsable code. More details on this experiment can be found in Table 6.6.

Table 6.6: Details on experiment with maximum sequence size 200 and more training examples

Result	**Result value**
Experiment short name	S6
Training examples	28,182
Examples in training set	22,546
Examples in validation set	2,818
Examples in test set	2,818
Unique methods in test set	808
Parsable predictions on test methods	260
Parsable score	32.18% (260/808)
Last best model	S4
Last best model parsable code	69.79%

Increase sequence length to 300 and limited training data

Previously, we observed that increasing the training examples reduced the parsable rate. Additionally, we also experimented with a maximum sequence length of 300 and a limited set of training examples to validate if the same effect occurs. As shown in Table 6.7, we only achieved 16.52% parsable code with the increased sequence length.

Table 6.7: Details on experiment with maximum sequence size 300 and limited number of training examples

Result	**Result value**
Experiment short name	S7
Training examples	11,699
Examples in training set	9,349
Examples in validation set	1,177
Examples in test set	1,173
Unique methods in test set	345
Parsable predictions on test methods	57
Parsable score	16.52% (57/345)
Last best model	S4
Last best model parsable code	69.79%

Experiment results

As shown in 6.7, a sequence length of 100 led to the highest parsable rate. Reducing the size further could improve the results. However, with these criteria, we ended up with only 11,699 training examples out of 52,585 training examples. This is a reduction of 77.75%. We did not perform more experiments with limiting the sequence length further, because this also limits the number of training examples that we can use. Experimenting further with a small set and filtering can lead to unexpected results. In conclusion, the experiments in this section did not contribute to a higher parsable score. An overview of the experiments is shown in Figure 6.7.

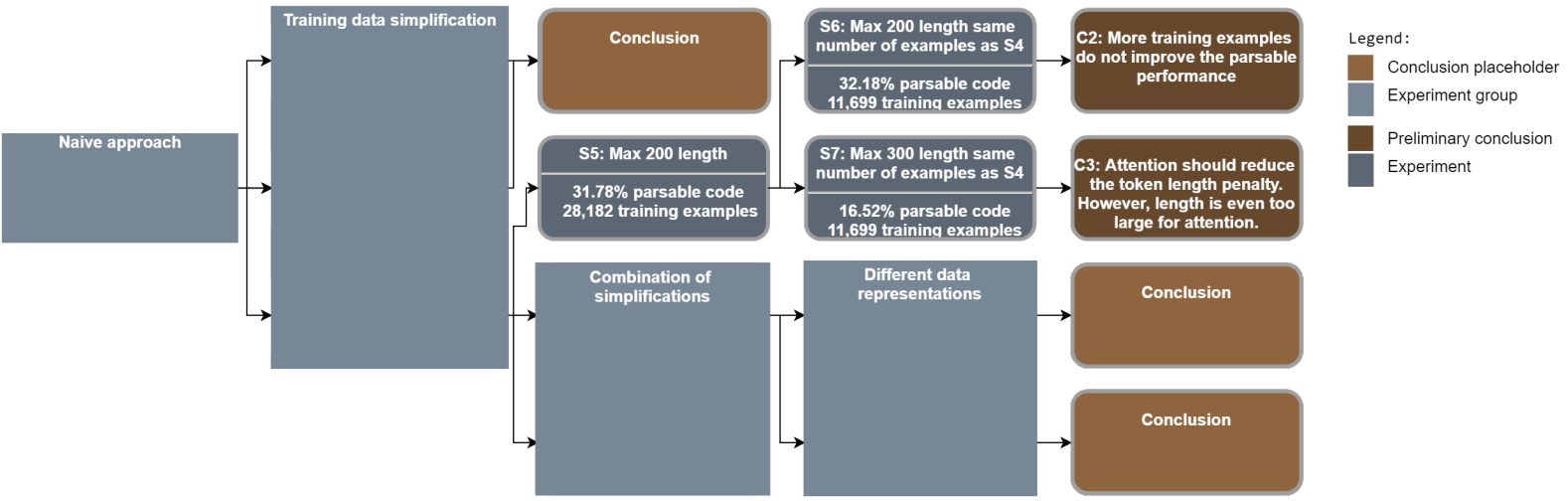


Figure 6.7: Experiments in optimizing sequence length

6.2.4 Combination of simplifications

As described in Section 6.2.3, the training set of our best model is limited by a sequence length of 100 tokens. In this section, we experimented with combinations of filters described in section 6.2.2.

Training examples with common subsequences and limited sequence length

When we combined the experiments with limiting the training examples (Section 6.2.2) and only using training examples with common subsequences (Section 6.2.2), we achieved 70.50% parsable code. More details on this experiment can be found in Table 6.8. Compared to the last best model score, this combination increased the percentage of parsable code by 0.71 percentage points.

Table 6.8: Details on experiment with maximum sequence size 100 and common subsequences

Result	**Result value**
Experiment short name	S8
Training examples	9,998
Examples in training set	7,989
Examples in validation set	1,005
Examples in test set	1,004
Unique methods in test set	278
Parsable predictions on test methods	196
Parsable score	70.50% (196/278)
Last best model	S4
Last best model parsable code	69.79%

No concrete classes and default methods and limited sequence length

When we combined the experiments with limiting the training examples (Section 6.2.2) and removed more advanced language features (Section 6.2.2), we achieved 65.07% parsable code. More details on this experiment can be found in Table 6.9. Compared to the last best model score, this combination did not improve the percentage of parsable code.

Table 6.9: Details on experiment with maximum sequence size 100 and no concrete classes and no default methods

Result	Result value
Experiment short name	S9
Training examples	6,749
Examples in training set	5,396
Examples in validation set	677
Examples in test set	676
Unique methods in test set	229
Parsable predictions on test methods	149
Parsable score	65.07% (149/229)
Last best model	S8
Last best model parsable code	70.50%

Experiment results

As shown in Figure 6.8, we were able to improve the percentage of parsable code from 69.79% to 70.50% by only using training examples with common subsequences. This is a small improvement and could be caused randomly. We address this issue in Section 6.2.8.

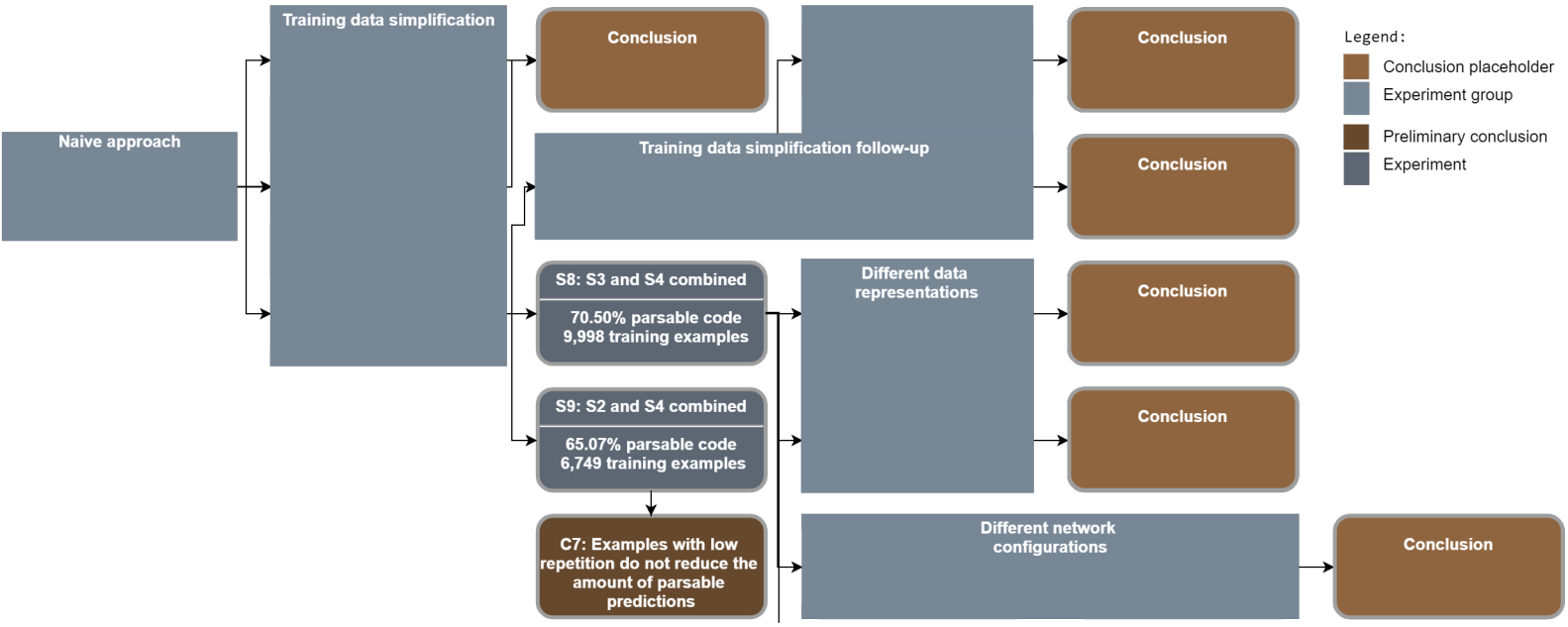


Figure 6.8: Experiments in combination of optimizations

6.2.5 Different data representations

Representing data in a different format could make it easier for the network to find patterns. In this section, we investigated compression and BPE.

Compression

With compression, tokens that belong together could be merged. Therefore, the neural network does not need to learn that a combination of multiple tokens has a specific effect. This effect only has to be linked to one token. We applied compression on the last best-performing dataset. We trained models with compression levels 1 to 10 (S10). The change in the maximum size of the training examples are

shown in Figure 6.9. When we analyze the models after training, we can observe that the model's loss increases immediately. This means that the model is unable to find patterns in the dataset and that compression is not suitable for our dataset. This is further investigated in Section 6.3.2. More details on this experiment can be found in Table 6.10.

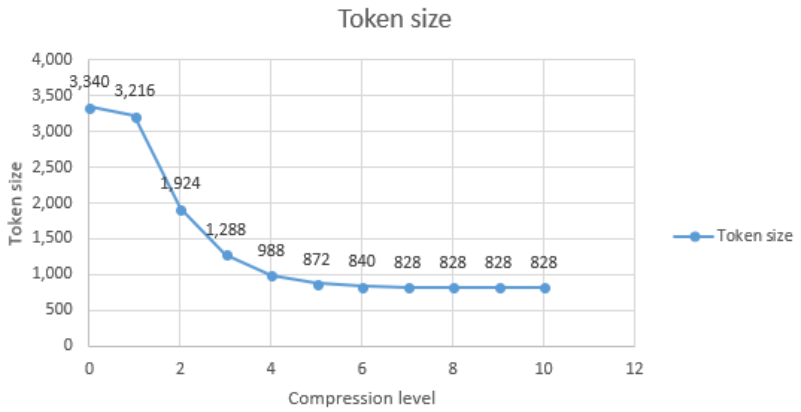


Figure 6.9: Maximum sequence length differences with various levels of compression

Table 6.10: Details on experiment with maximum sequence size 100, common subsequences, and compression

Result	**Result value**
Experiment short name	S10
Training examples	9,998
Examples in training set	7,989
Examples in validation set	1,005
Examples in test set	1,004
Unique methods in test set	278
Parsable predictions on test methods	0
Parsable score	0% (0/278)
Last best model	S8
Last best model parsable code	70.50%

BPE

With BPE, we tell the neural network that two tokens belong together. We applied BPE to the last best performing dataset. With these criteria, we achieved 57.55% parsable code. More details on this experiment can be found in Table 6.11.

Table 6.11: Details on experiment with maximum sequence size 100, common subsequences, and BPE

Result	Result value
Experiment short name	S11
Training examples	9,998
Examples in training set	7,989
Examples in validation set	1,005
Examples in test set	1,004
Unique methods in test set	278
Parsable predictions on test methods	160
Parsable score	57.55% (160/278)
Last best model	S8
Last best model parsable code	70.50%

Experiment results

Both compression and BPE did not improve our results. An overview of the experiments can be found in Figure 6.10.

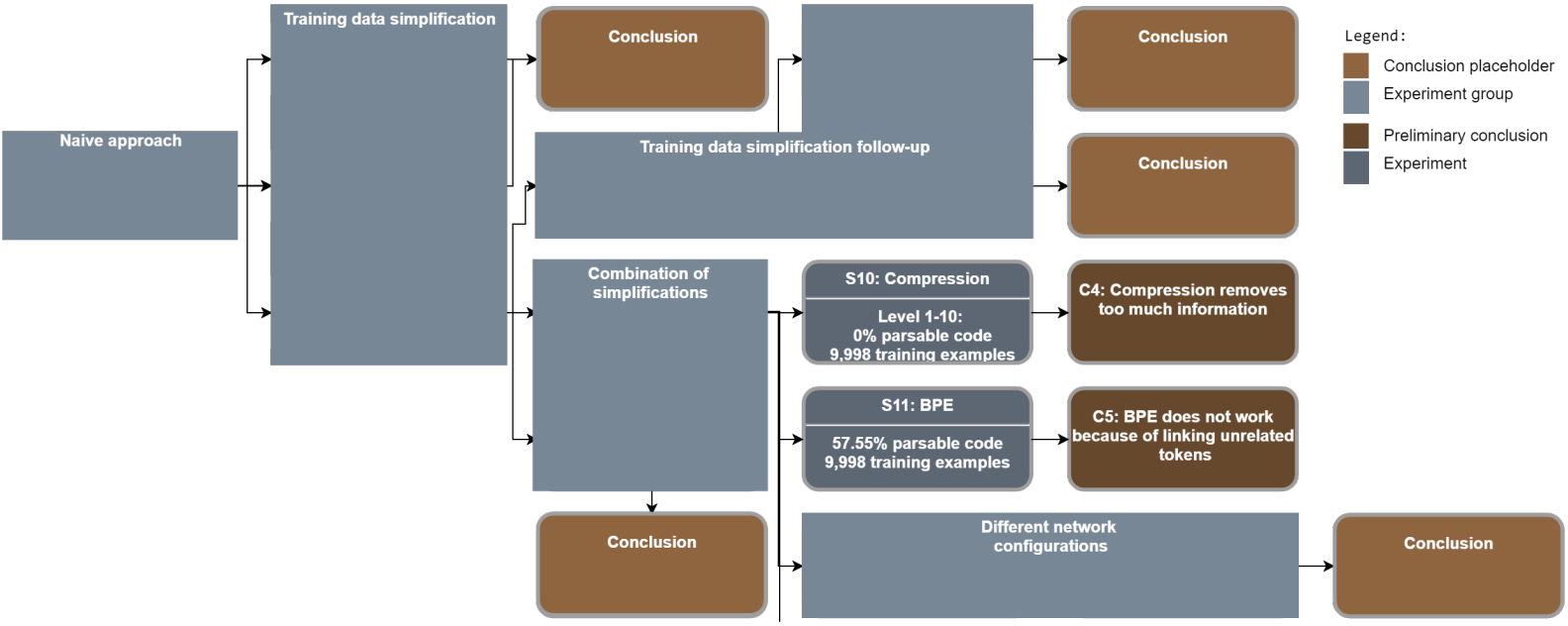


Figure 6.10: Experiments with different data representations

6.2.6 Different network configurations

A different configuration of the neural network can result in different outcomes. We are applying a neural network similar to one used in another research, and fine-tune the model ourselves. For example, we tune the number of layers inside the network. The neural network should be able to make more complex conclusions when more layers are used. The first layer could learn how simple patterns can be applied, and the second layer could learn how a combination of simple patterns can form a more complex pattern. Additionally, when the size of each network layer is increased, more patterns could be captured by each layer. GRU cells can be used instead of LSTM cells. The dropout can be increased, which will remove the connection from some cells to other cells. This makes the cells more independent of each other. An overview of experiments on the network configuration can be found in Table 6.12. More details about the training set and the last best performing model can

be found in Table 6.13.

Table 6.12: Overview of all network experiments

Name	Description	Parsable rate
S12	1,024 LSTM cells	67.27% (187/278)
S13	2 Output layers	78.42% (218/278)
S14	GRU cells	71.22% (198/278)
S15	2,048 LSTM cells	51.44% (143/278)
S16	4 Output layers	64.39% (179/278)
S17	2 input and 4 output layers	62.59% (174/278)
S18	Network of related research [HWLJ18]	0 (0/278)
S19	Dropout of 50%	72.30% (201/278)
S20	Dropout of 50% and 2 output layers	86.69% (241/278)

Table 6.13: Details on experiment with maximum sequence size 100, common subsequences, and different network configurations

Result	Result value
Training examples	9,998
Examples in training set	7,989
Examples in validation set	1,005
Examples in test set	1,004
Unique methods in test set	278
Last best model	S8
Last best model parsable code	70.50%

Conventional neural network

Additional experiments with a conventional neural network were performed. However, we were unable to make predictions on a sequence length of more than 50 tokens. This could be caused by hardware or software limitations. We did not explore the problem in detail, because, in related research, CNNs only improve the BLEU score of their prediction by 4.26% (61.19/58.69) [ZZLW17]. We expect that an improvement like this will not have a big impact on generating test code.

SBT

We also performed tests with SBT. We were unable to generate parsable code with this technique. More information about the results can be found in Section 6.4.

Experiment results

An overview of all the experiments can be found in Figure 6.11. It can be concluded that S20 performs the best. This model uses two output layers and a dropout of 50%.

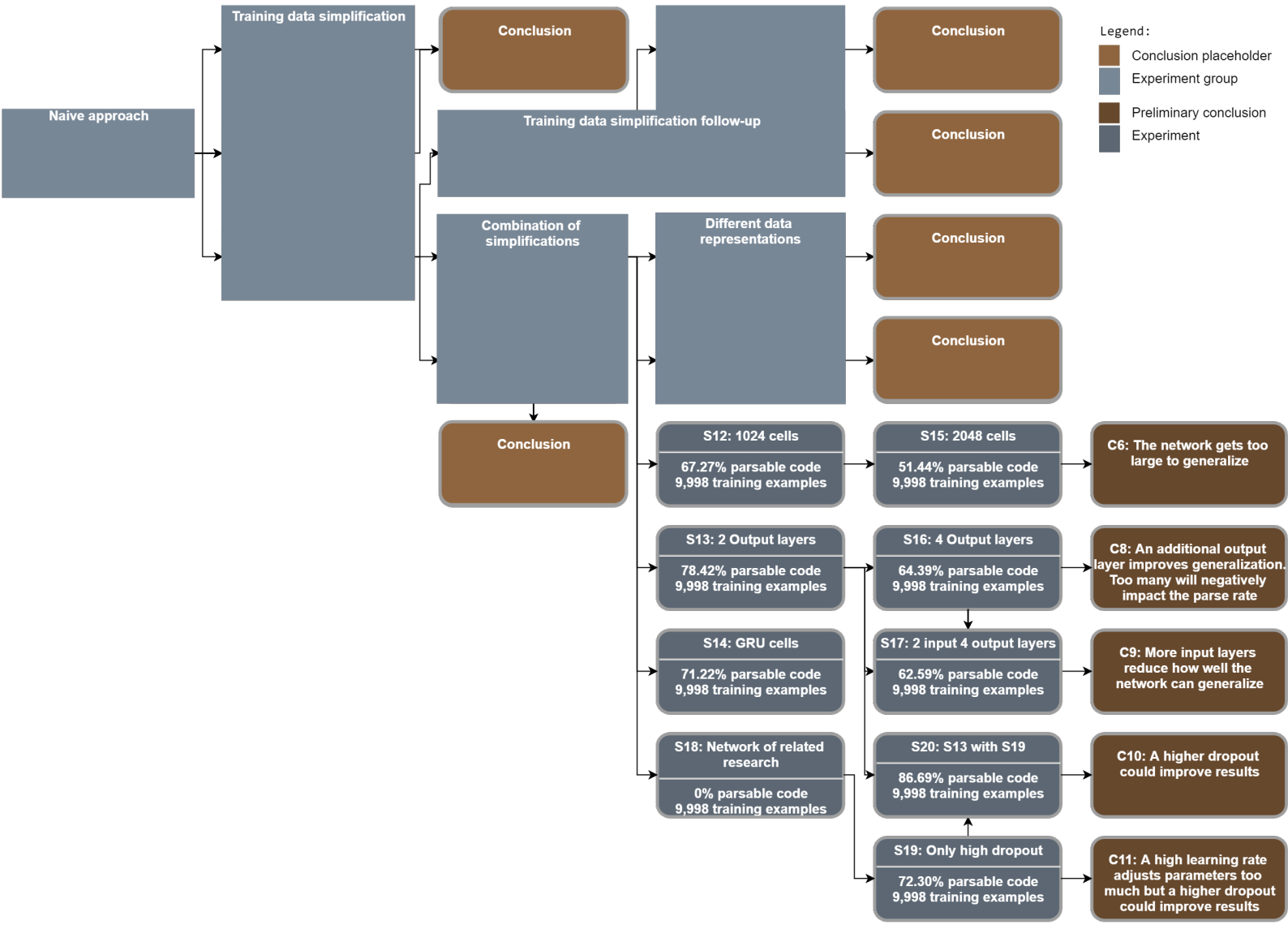


Figure 6.11: Experiments in different network configurations

6.2.7 Generated predictions

We manually analyzed the results of the trained models. The generated test code is often not related to the method that was used as input. For example, when we want to predict a test for the method updateVersion, shown in Listing 6.5, then the model will generate the test shown in Listing 6.6. This test is unrelated to the code. The expected output can be found in Code 6.7.

```
1 public void updateVersion() {
2   ++version;
3 }
```

Listing 6.5: Unit test prediction input

```
@Test public void testGetSequence() {
  assertEquals(1, m_SIZE.getBytes());
}
```

Listing 6.6: The generated unit test

```
@Test public void testUpdateVersion() {
  target.updateVersion();
  String result = target.getVersion();
  assertThat(result, is("1"));
}
```

Listing 6.7: Expected unit test

6.2.8 Experiment analysis

An overview of all experiments is shown in Figure 6.12.

Figure 6.12: All experiment results

The percentages in the figure are not statistically checked. As mentioned in Section 5.4.8, we only analyzed the experiments that improved our results. These are experiments S4, S8, S13, and S20. In total, we trained five models for each experiment. The parsable scores are shown in Table: 6.14. The columns represent the experiments, and the rows are the different runs.

Table 6.14: Parsable code score for the most important experiments with different seeds

ID	Random number	S4	S8	S13	S20
1	5,274	56.42% (211/374)	55.04% (153/278)	67.99% (189/278)	73.38% (204/278)
2	8,526	59.36% (222/374)	52.52% (146/278)	68.71% (191/278)	75.54% (210/278)
3	1,873	67.11% (251/374)	50.72% (141/278)	60.43% (168/278)	73.74% (205/278)
4	1,944	67.91% (254/374)	59.35% (165/278)	64.03% (178/278)	70.14% (195/278)
5	1,234	69.79% (261/374)	70.50% (196/278)	78.42% (218/278)	86.69% (241/278)

For our statistical check, we came up with two hypotheses:

- H0: There is no significant difference between the samples

- H1: There is a significant difference between the samples

We apply the algorithms discussed in Section 5.4.8 to evaluate if H0 can be rejected. The results of the experiments can be found in Table: 6.15. We have a p-value lower than 0.05, so we can reject H0. This means that there is a difference between the groups.

Table 6.15: ANOVA experiment results

Source of variation	**Sum of squares**	**Degrees of freedom**	**Mean square**	**F-value**	**p-value**
Between groups	873.6	3	291.2	15.9	0.0030
Within groups	509.8	4	127.4	7.0	0.0039
Residual	219.7	12	18.3		
Total	1.6	19			

We apply the algorithms discussed in Section 5.4.8 to evaluate what the differences in results are. The results of the experiments can be found in Table: 6.16. We can reject H0 for the experiments that have a p-value lower than 0.05. That means that there is a statistically significant difference between S8 and S13, S8 and S20, and S13 and S20.

In Section 6.2.4, we discussed that common subsequences with a maximum sequence length of 100 (S8) improved the parsable score of a maximum sequence length of 100 (S4) by 0.71 percentage points. When we calculate the averages of the follow-up experiments in Table 6.14, then S4 has an average of 57.63%, while S8 has an average of 64.12%. We did not find a significant difference between these experiments. Therefore, we cannot conclude what filter on the dataset is better.

Table 6.16: Difference between experiments

Experiment	**Mean Difference**	**95% confidence interval of difference**	**Adjusted p-value**
S4 vs. S8	6.492	-5.739 to 18.72	0.2757
S4 vs. S13	−3.796	-19.12 to 11.53	0.7544
S4 vs. S20	−11.78	-24.34 to 0.774	0.0612
S8 vs. S13	−10.29	-18.40 to -2.180	0.0225
S8 vs. S20	−18.27	-27.62 to -8.927	0.0047
S13 vs. S20	−7.986	-13.74 to -2.232	0.0164

As shown in Table 6.16, S20 should have better results than S8 and S13, and S13 should have better results than S8. We came up with the following hypothesis for the directional t-test:

- H0: There is no significant difference between the results of Sx and Sy
- H1: The results of Sx are significantly higher than the results of Sy

We apply the directional t-test discussed in Section 5.4.8 to evaluate the hypothesis. The results are displayed in Table 6.17. For these experiments, we corrected the alpha to 0.05/3 to keep the same chance on an error. In these tables, it is shown that the p-value is lower than 0.02 (0.05/3). This means that H0 can be rejected in all cases.

Table 6.17: Directional t-test of significantly different experiments

	S8 vs. S13	S8 vs. S20	S13 vs. S20
Number of pairs	5	5	5
One- or two-tailed P value?	One-tailed	One-tailed	One-tailed
T-value	5.165	7.959	5.65
Degrees of Freedom	4	4	4
Averages	57.626 - 67.916	57.626 - 75.898	67.916 - 75.898
P-Value	0.0033	0.0007	0.0024

6.3 Experiments for RQ2

In this section, we address **RQ2**. We compared the time to compress across various levels of compression and what compression does with the accuracy of the model. In Section 6.2.5, we already observed that any level of compression prevents the model from predicting parsable code. In Section 5.4.7, we discussed that we should analyze the loss instead of accuracy if the model is unable to find patterns.

6.3.1 Compression timing

We trained a neural network with uncompressed training examples, and other neural networks with training data that have a compression level of 1, 2, or 10. The results are visualized in Figure 6.13

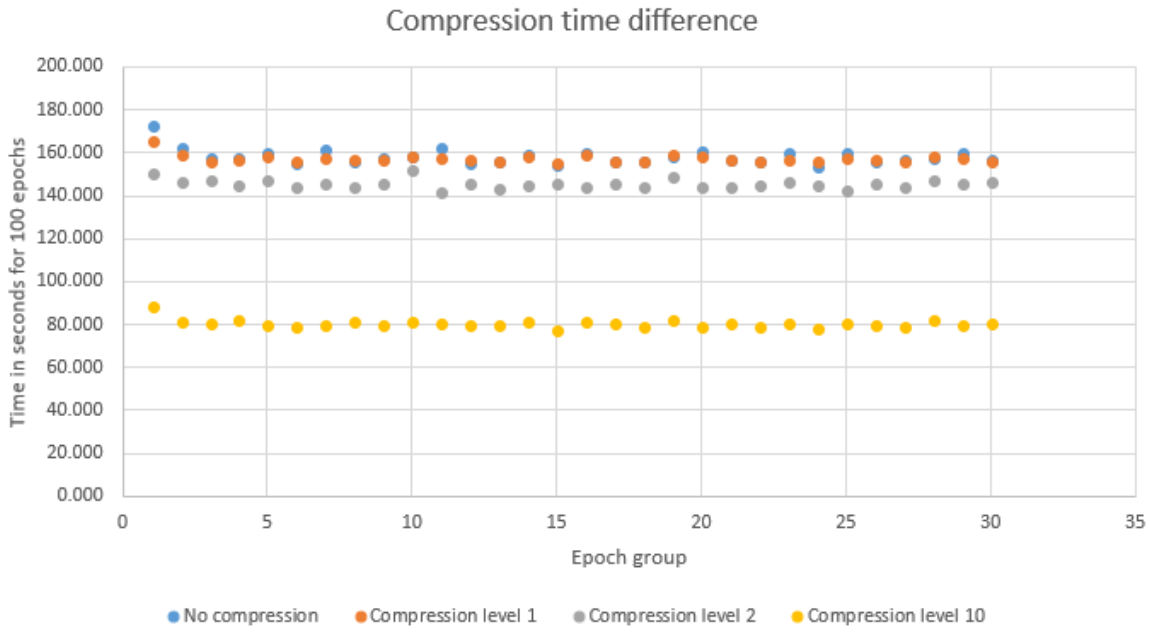


Figure 6.13: Compression timing scatter plot

We hypothesized that compression will impact the time needed to make predictions. This leads to the following hypotheses:

- H0: There is no significant difference between the samples
- H1: There must be an alternative hypothesis

We applied the algorithms discussed in Section 5.4.8 to evaluate if H0 can be rejected. The results of the experiments can be found in Table: 6.18. We have a p-value lower than 0.05. Thus, H0 can be rejected. This means that there is a significant difference between the groups.

Table 6.18: ANOVA on compression timing

Source of variation	Sum of squares	Degrees of freedom	Mean square	F-value	p-value
Between groups	128.350	3	42.783	16.930	< 0.0001
Within groups	481.4	30	16.05	6.35	< 0.0001
Residual	227.4	90	2.527		
Total	129.059	123			

We applied the algorithms discussed in Section 5.4.8 to evaluate what the differences are in duration. The results of the experiments can be found in Table: 6.19. We found a p-value lower than 0.05 for all groups, except for no compression paired with compression level 1. This means that there is a significant difference between all groups, except for the group with no compression paired with compression level 1.

Table 6.19: Difference in compression timing

Experiment	Mean Difference	95.00% confidence interval of difference	Adjusted p-value
Compression level 1 vs. No compression	−0.8976	-1.85 to 0.055	0.0704
Compression level 2 vs. No compression	−12.75	-14.42 to -11.08	< 0.0001
Compression level 10 vs. No compression	−77.93	-79.12 to -76.74	< 0.0001
Compression level 2 vs. Compression level 1	−11.85	-12.8 to -10.9	< 0.0001
Compression level 10 vs. Compression level 1	−77.04	-77.54 to -76.53	< 0.0001
Compression level 10 vs. Compression level 2	−65.19	-66.17 to -64.2	< 0.0001

To test if the time needed for compression reduces when the compression level increases, we formulated the following hypothesis:

- H0: There is no difference in timing of Cx and Cx+1
- H1: The timing of Cx+1 is significantly lower than the timing of Cx

We applied the directional t-test discussed in Section 5.4.8 to evaluate the hypothesis. An overview of the p-values can be found in Table 6.20. H0 can be rejected for all tested compression levels.

Table 6.20: Directional t-test on compression timing

Experiment	p-value
No compression vs. Compression level 2	<0.0001
No compression vs. Compression level 10	<0.0001
Compression level 1 vs. Compression level 2	<0.0001
Compression level 1 vs. Compression level 10	<0.0001
Compression level 2 vs. Compression level 10	<0.0001

6.3.2 Compression accuracy

We hypothesized that compression will prevent the model from training. In this section, we want to prove that the loss in an epoch is not due to chance. When the loss difference between multiple models is not by chance, and the loss increases from the first epoch when the number of epochs is higher, then we have proven that compression does not work on our dataset.

For our next experiments, we trained five new neural network models with a compression level of one. We used a compression level of one as this is closest to a training set for which we can generate

parsable code (based on S8 in Figure 6.8). We visualized the loss on the validation set in Figure 6.14. The higher the loss, the worse the model performs.

Figure 6.14: Loss for every 100th epoch

The first step in this experiment is to show that our results are not because of chance. Therefore, we want to determine if the loss between every epoch is different. We formulated the following hypotheses:

- H0: There is no significant difference between the samples
- H1: There is a significant difference between the samples

We applied the algorithms discussed in Section 5.4.8 to evaluate whether H0 can be rejected. The results of the experiments can be found in Table: 6.21. We have a p-value lower than 0.05, so we can reject H0. This means that there is a significant difference between the groups.

Table 6.21: ANOVA experiment for compression loss

Source of variation	**Sum of squares**	**Degrees of freedom**	**Mean square**	**F-value**	**p-value**
Between groups	52.96	9	5.885	40.246	< 0.0001
Within groups	0.002 947	4	0.000 736 8	5.039	0.0025
Residual	0.005 264	36	0.000 146 2		
Total	52.97	49			

We applied the algorithms discussed in Section 5.4.8 to evaluate what the differences in the loss are. The results of the experiments can be found in Table: 6.22. We found a p-value lower than 0.05 for all groups. This means that there is a significant difference between all the groups.

Table 6.22: Difference in loss groups

Experiment	Mean Difference	95.00% confidence interval of difference	Adjusted p-value
Epoch 200 vs. Epoch 100	1.166	1,132 to 1,201	< 0.0001
Epoch 300 vs. Epoch 200	0.5446	0,5224 to 0,5668	< 0.0001
Epoch 400 vs. Epoch 300	0.396	0,3729 to 0,4192	< 0.0001
Epoch 500 vs. Epoch 400	0.3231	0,3063 to 0,3398	< 0.0001
Epoch 600 vs. Epoch 500	0.2656	0,2211 to 0,3101	< 0.0001
Epoch 700 vs. Epoch 600	0.2346	0,1818 to 0,2873	0.0002
Epoch 800 vs. Epoch 700	0.1964	0,1602 to 0,2325	0.0001
Epoch 900 vs. Epoch 800	0.1888	0,1589 to 0,2188	< 0.0001
Epoch 1000 vs. Epoch 900	0.1464	0,1017 to 0,191	0.0005

To test if the loss increases, we formulated the following hypothesis:

- H0: There is no significant difference between the loss of Ex and Ex+1
- H1: The loss of Ex+1 is significantly higher than the loss of Ex

We applied the directional t-test discussed in Section 5.4.8 to evaluate the hypothesis. An overview of the p-values can be found in Table 6.23. We can reject H0 for all compression levels, meaning that the loss increases after every epoch.

Table 6.23: Directional t-test on compression loss

Experiment	p-value
Epoch 200 vs. Epoch 100	<0.0001
Epoch 300 vs. Epoch 200	<0.0001
Epoch 400 vs. Epoch 300	<0.0001
Epoch 500 vs. Epoch 400	<0.0001
Epoch 600 vs. Epoch 500	<0.0001
Epoch 700 vs. Epoch 600	<0.0001
Epoch 800 vs. Epoch 700	<0.0001
Epoch 900 vs. Epoch 800	<0.0001
Epoch 1,000 vs. Epoch 900	<0.0001

6.4 Applying SBT in the experiments

In Section 6.2 and 6.3, we trained our models based on a textual representation. However, in this form, the structure of the grammar is invisible. With SBT, this information can be captured in a textual form so that the neural network can learn the code structure. This could make it easier for the machine learning algorithm to capture the structure of the code and improve its prediction capabilities [HWLJ18]. In Section 6.2.6 we mentioned that we applied SBT to our experiments, but that we could not get any parsable code. This chapter addresses the problems that we encountered.

6.4.1 Output length

Our goal is to translate source code into tests. However, the SBT representations are very long. When we select the 10,000 smallest training examples, we have a maximum token length of 714 compared to 91 tokens maximum for method code. The problem of a larger sequences is that it increases the memory usage on the GPU. In this case, SBT has 7.85 (714/91) times the size of normal code. This can cause issues during training.

Limiting the training examples is not an option. The number of training examples is already low when we compared it to the number of training used in related research mentioned in Section 3.1.3.

6.4.2 Training

To support the sequence length of 714, we could only use a neural network with LSTM nodes of size 64 instead of 512. Using LSTM nodes of size 512 was impossible because of memory limitations. We selected the smallest 9,998 training examples for method code and 9,995 training examples for SBT. We trained a model for both sets with this network setup. We could not generate valid SBT trees. We were able to generate parsable method code. However, the generated code did not make any sense. It consists of method bodies that asserted two empty string. The results are inconclusive, because it was impossible to use a reasonable neural network in combination with SBT.

6.4.3 First steps to a solution

We tried to reduce the AST structure as much as possible by removing redundant layers. For example, in the AST, a NameExpr only has a SimpleName as a child which only has a string as a child. This makes the SimpleName in between redundant, so we removed it. With this type of compression, we were able to reduce the maximum token size of the first 10,000 training example by 20.17% (714 to 570). However, this reduction is too small to enable the use of a reasonable size neural network. It should at least be possible to use a network of 512 LSTM nodes in the encoder and decoder.

We also tried to use the compression algorithm of Ling et al. [LGH+16]. However, the algorithm will increase the vocabulary size because there are more repeated token combinations. An overview of the compression levels can be found in Table 6.24. This greater vocabulary size increased the memory needed to train the model, which made training the model impossible.

Table 6.24: SBT compression effect

Level of compression	**Vocabulary size**	**Max sequence size**
None	9,123	570
Level 2	176,546	220
Level 3	227,575	168
Level 4	237,795	144
Level 5	222,916	136

Chapter 7

Discussion

In this chapter, we discuss our experiment results on i) what neural network solution performs best to generate unit tests, and ii) how compression affects accuracy and performance. First, we start with a summary of our most important results.

7.1 Summary of the results

We noticed that we could extract more training examples by using an AST analysis instead of a bytecode analysis. However, bytecode analysis has still advantages over AST analysis. Only bytecode analysis can extract training examples when in the example the base type of the class under test is used. This means even if AST analysis can extract more training examples, bytecode analysis is still needed in order to get a dataset that includes examples that use basic object-oriented code.

When we train a neural network, then we see an improvement in the amount of parsable code generated when we only including examples with a sequence length of maximal 100 tokens. Increasing the sequence length more reduces this. There is no improvement noticed when different or a combination of other optimizations are evaluated. Besides, there was also no improvement when different data views are applied. One of the data views was the compressed view. For this view, we saw a reduction of time needed to train models. However, with this view, we were unable to produce parsable code. When we applied SBT, we also could not generate working models. This was since SBT had a large negative impact on sequence length. This made it impossible for our neural network to train on this data. Tuning the neural network configuration improved how much parsable code could be generated. There was an improvement when the number of output layers is set to two instead of one, and when a 50% dropout is configured. We are unable to generate test code, but the increase in the amount of parsable code could indicate that the neural network is starting to learn how code is written. Writing code itself is complicated. When we can improve our test generator further, then we could start to see patterns in the predictions on how code can be tested.

7.2 RQ1: What neural network solutions can be applied to generate test suites in order to achieve a higher test suite effectiveness for software projects?

For this research question, we first addressed the parsable code metrics that we used to measure performance. This metric is critical in this research, and is used to guide the research towards more promising solutions. With this metric, we analyzed the results of various models.

7.2.1 The parsable code metric

The application of machine learning to test suite generators is very new. For this research, only the generation of parsable code was achieved. Unfortunately, this does not give any insights into

how well the code tests. It will give an idea if the machine learning model starts to understand the programming language. This is a first step towards generating test code. However, more research on other metrics could give a better insight into the progress towards a unit test generator using machine learning.

7.2.2 Training on a limited sequence size

A big part of generating more parsable code was filtering training examples. A great improvement was made when we limited training examples to a sequence length of 100 tokens. The effect of better predictions with smaller sequences was also noticed in other research by Bahdanau et al. [BCS+15]. We confirm this finding.

7.2.3 Using training examples with common subsequences

During our experiments, we got better results when we only used training examples with common subsequences and a sequence length of 100 tokens. Later, it turned out that it did not improve our results. The reason why this should have worked is that the neural network can learn context. This is related to research into BPE. With BPE, translation scores were improved by linking words that are written differently (loving - love) but have (almost) the same meaning [SHB15]. So, including training examples that are related to other training examples should give the neural network the ability to learn more complex transformations. However, when we only include training examples with a lot of context, we make it harder for the model to make predictions. In addition, the test set also consists of training examples with a lot of context. Therefore, a lower score in parsable rate does not necessarily mean that it is worse in predicting test code. Future research could give more insight when a metric can be applied that is more accurate in the development phase of the test generator.

We performed many experiments on top of this. When this set indeed reduces the ability to write test code, then this only means that we could have achieved a higher parsable rate. This will not invalidate any results.

7.2.4 BPE

Applying BPE on our datasets reduced the amount of parsable code that could be generated. We used BPE to include relations between words and symbols instead of word part. Other research found that in NLP the addition of the relation between word part improves results [SHB15]. A reason why this does not work on words in code could be because there are too many relations that make no sense. For example, because of BPE, the word stop in both stopSign and stopWalking share some level of meaning. These two identifiers do not have much in common when translating to code. This could be the reason why we observed a drop in the percentage of generated parsable code. However, we did not perform a statistical check to validate these results.

7.2.5 Network configuration of related research

During the experiments, we have tried various network configurations. One configuration we tried was similar to the network used by Hu et al. [HWLJ18], which they used to summarize code. This configuration did not work with our experiments, because they used a too high learning rate for our data. With a higher learning rate, less time is needed to create the model, since great steps are taken during the learning process. A high learning rate could mean that it never reaches the ideal model. However, our model did not learn anything. We expect that our data is harder to generalize, so adjusting each training step too much will cause the modifications to be too specific.

7.2.6 SBT

We were unable to use SBT to generate code, since it outputs sequences which are too long. We were unable to reduce the sequences to a level where we can train a sufficient neural network. Also, we could not compress the sequences too much, because the whole point is to add additional information.

When we compress it too much, we lose this information. In future research, other machine learning solutions could be applied which are capable of handling this kind of data. For instance, instead of translating based on words, translations could be done based on sections [HWZD17].

7.2.7 Comparing our models

In Section 6.2, we observed big differences between experiments. In Section 6.2.8, we did not find a statistical correlation between all these experiments. There still could be a correlation, but it was not noticeable. This could be caused by a too small sample size. An overview of all experiments can be found in Figure 6.12. Table 6.17 shows that the result of S20 is better than S8 and S13, and the result of S13 is better than S8. From these relations, we can conclude that on our dataset S8 we have better results when we use one input layer and two output layers instead of one input layer and one output layer. In addition, by adding a dropout of 50% to this configuration will improve the results further. The reason for the improved results is that, when adding another output layer and 50% dropout, the network now can make more advanced conclusions and the network can make more independent decisions. The first layer can capture the general conclusion and the second layer can make more advance conclusions based on the previous conclusions.

7.3 RQ2: What is the impact of input and output sequence compression on the training time and accuracy?

Compression can make the application of the used machine learning technique easier, because one of the weak points of our technique is long sequence lengths. Compression could improve the results by solving this problem and reduces the time needed to train models. For compression, we used the algorithm proposed by Ling et al. [LGH+16].

7.3.1 Training time reduction

We noticed an improvement when applying the compression to the input and output sequences. Unfortunately, Ling et al. [LGH+16] did not report on the time reduction of the training time of their models. However, we found a significant time reduction on our dataset.

7.3.2 Increasing loss

In the experiments for **RQ**1, we observed that we were unable to generate parsable code with compression. We expected that, with compression, too much information is captured in a single token. This could make predicting code harder, as every token has too much meaning. We also noticed that during training the loss increases on the validation set. This increase means that the neural network failed to learn patterns. These issues contradict the research done by Ling et al. [LGH+16]. However, we were unable to generate test code without compression. Ling et al. [LGH+16] managed to generate working code without applying compression. Therefore, it could be that compression only works when the neural network can capture the problem. The problem why we cannot generate working code could be caused by our dataset, because the dataset may exist out of barely related training examples. Compression could make them even more unrelated, what could have a negative impact. When the training set would have contained training examples that are very related, then reducing this could give only a small penalty. The positive impact of compression (reducing sequence length) could be greater than the penalty of making the training examples more unrelated.

7.4 Limitations

The scope of this project is to find out if neural networks can be used for unit test generation. We still left many questions and problems unanswered, of which some are listed in this chapter.

7.4.1 The used mutation testing tool

To calculate the mutation score of our baseline, we used an alternative version of PIT Mutation Testing to calculate the mutation testing score. During our study, this tool achieved the most reliable result to measure test suite effectiveness. However, new versions of the original tool were released and could outperform this version. The mutation generation ability of the original tool should be checked in future research and used when it performs better. Our research is not invalid when the original tool performs better, because we did not perform comparisons on the baseline yet. Even if we did, it would not have made a difference. The result should only be more accurate when comparing to real faults.

7.4.2 JUnit version

During the start of the project, Junit 4 was the latest version of this test framework. However, during the project Junit 5 was released [Junc]. We did not restart our gathering process, because a recently released version will be used by fewer projects than a version which has been around for 12 years [Junb].

7.4.3 More links for AST analysis depends on data

We noticed that for our research AST analysis could create more links than bytecode analysis, because it supports more projects. We did not perform a statistical check to validate this, because it was not of importance whether one method was better compared to another. More in-depth research should be performed when these techniques are compared in future research.

7.4.4 False positive links

In section 6.1.3, we mentioned the number of links we could create on a set of projects. However, it is possible that for some tests an incorrect class under test is used. This could have resulted in false positive matches. Many of them could be eliminated with backward slicing, as described in section 3.2.1. We expect that most false positives are eliminated when we link based on the statements returned by this method.

7.5 The dataset has an impact on the test generators quality

In our research, we only considered a selection of open-source projects. The result could be different when other open or closed-source projects were used.

7.5.1 Generation calls to non-existing methods

The unit tests in our training examples use helper functions developed for the tests, while we do not include these in the training data. Thus, our generated tests will contain calls to non-existing methods. We limited our research by not looking into this problem. Support for these cases has to be addressed in further research.

7.5.2 Testing a complete method

In an average software project, multiple unit tests exist for a single method. Every unit test is written for a small part of the method code. We did not embed this relation in our dataset. Therefore, our machine learning model has to learn this relation on its own. It has to learn from the training data that some parts of the method translate to a single unit test. In future research, a more direct translation should be performed because it makes learning easier. It will be easier since this will limit the sequence length and will relieve the machine learning algorithm from unnecessary complexity.

7.5.3 The machine learning model is unaware of implementations

During training, the machine learning model learns the usage of many methods. When the model is applied for making predictions on unseen data, it will see new method calls which it has not encountered before. The model could know what the call does based on its naming and context. However, it would be easier for the machine learning algorithm to have more detailed information such as the implementation of the method. Thus, the information in our training examples is limited. Adding more detail could make predictions easier.

7.5.4 Too less data for statistical proving our models

In most of our statistical analysis, we used five samples per group. In future research, the sample size should be increased to validate our findings.

7.5.5 Replacing manual testing

A potential risk is introduced when our method is replacing humans for writing tests. Our generator could be unable to cover all mutants that would have been captured by a human. This will result in less test suites effectiveness.

Chapter 8

Related work

In our research, we translate source code (method body) to source code (unit test). We are unaware of research that does a similar translation. However, in research, many projects apply machine learning to translate from or to source code. An overview of recent projects can be found in Table 8.1. The difference between these projects and our work is that they do not perform a translation to test code. The work of Devlin et al. [DUSK17] is closely related because they address the issue of finding software faults. However, compared to our work, they developed a static analysis tool instead of a test suite generator. They try to find locations in the code that could have bugs while we generate code that tests if the behavior of a piece of code is still the same then when the test was written.

Table 8.1: Machine learning projects that translate to or from code

Researcher	**Project**	**Description**
Beltramelli et al. [Bel17]	Pix2code: Generating Code from a Graphical User Interface Screenshot	Used machine learning to translate a mock-up (image) into HTML. Instead of directly translating into HTML, they targeted a domain-specific language (DSL) with a limited vocabulary to reduce the complexity
Ling et al. [LGH+16]	Latent Predictor Networks for Code Generation	Translates game cards into code. For this, they developed a custom machine learning algorithm to make predictions on tokens that are not very common in the training data
Parr et al. [PV16]	Towards a Universal Code Formatter through Machine Learning	Developed a universal code formatter that learns the code style from a code base
Karaivanov et al. [KRV14]	Phrase-based statistical translation of programming languages	Used phrase tables to translate one programming language into another
Devlin et al. [DUSK17]	Semantic Code Repair using Neuro-Symbolic Transformation Networks	Developed a solution that repairs syntactic errors and semantic bugs in a code base
Zheng et al. [ZZLW17]	Translating Code to Comments by Exploiting Domain Features	Translates source code to comments
Yin et al. [YN17]	A Syntactic Neural Model for General-Purpose Code Generation	Developed an algorithm that translates natural language into source code
Hu et al. [HWLJ18]	Deep Code Comment Generation	Developed an algorithm that translates code into a natural language

Chapter 9

Conclusion

Incorporating machine learning in a test generator could lead to better test generation. A generator like this cannot only be used to generate a complete test suite; it can also be used to aid software engineers in writing tests. Nevertheless, more work has to be done to get to this point.

To our knowledge, this research is the first study towards test generation with the application of machine learning. We were able to create a training set for this problem, and managed to train neural networks with this dataset. We extracted more than 52,000 training examples from 1,106 software projects. When training machine learning models, we noticed that the quality of our models improved when configurations are tweaked and when the training data was filtered. Our first results are promising, and we see many opportunities to continue the development of the test generator. We believe that our results could be further improved when a machine learning algorithm specific for code translation is used, if the neural network is trained on how small parts of the code are tested instead of whole methods, and when the neural network has more detailed implementation information. Our results can be used as a baseline for future research.

We also experimented with machine learning approaches that improved the results of other studies. We applied compression, SBT, and BPE on our training set. A correlation was found that indicates that the time needed to train a model is reduced when compression is applied. An improved compression algorithm or a dataset optimized for the compression technique could be used to create a model that can make good predictions in less time, what will make research in this field more efficient. We also improved the application of SBT in translations to code. We developed an algorithm that can translate an SBT representation back to code (Appendix A). This tool could already be used for experiments when the output length is limited. BPE was also applied, to let our model use additional information on how parts of the training data are connected. In our experiment, it did not have a positive effect. However, optimizing the number of connections could work for our datasets when only connections that add relevant information are included.

All steps from finding suitable GitHub projects until the calculation of the parsable score are automated in scripts. A copy can be found in Appendix A. With these scripts, all results of our experiments can be reproduced. The first part of the scripts can be executed to retrieve training examples or the provided backups with the used training examples can be restored. When it is preferred to download all projects, then the overview of all used repositories should be used. We included the used commits in the GitHub links. The second part is to configure the dataset generator according to the parameters in our experiments. From this point, the rest of the scripts can be used to execute the rest of the pipeline in order to calculate the parsable score.

To summarize, with this research we contribute an algorithm that can be used to generate training examples (for method to test translations), the training sets used in our experiments, SBT to code implementation, and a neural network configuration that can learn basic patterns between methods and test code. Finally, we also contribute software that takes GitHub repositories as input and a model that can be used to predict tests with machine learning as output.

Chapter 10

Future work

There are still some additional experiments which could add value to the presented work. We did not manage to do experiments with SBT, and we did not investigate further if the percentage of common subsequences has an impact on the results of the neural network. Additionally, we found some areas that are promising for a machine learning based test generator that we did not address.

10.1 SBT and BPE

Both BPE and SBT did not positively affect our results. We could not generate valid trees with SBT because of hardware/software limitations. The concept of SBT is very promising and should be addressed in future research. BPE was also applied on our training data. In future research, it should be tested if optimizing the number of connections improves results by only introducing a connection that adds relevant information.

10.2 Common subsequences

We performed experiments where we filtered training examples based on the level of common subsequences. We only kept training examples where more than 80% of the training example are common under other training examples. An optimized percentage could result in better predictions. In future research, the rate of common subsequences could be tuned to find out what the effect of varying the percentage is.

10.2.1 Filtering code complexity with other algorithms

During our research, we filtered complex training examples (Section 6.2.2). We only checked if concrete classes and default methods were used. However, more criteria could be tested. For instance, training examples that have a too high cyclomatic complexity could be excluded.

10.3 Promising areas

Apart from future work, there are many areas that we did not address in our research while they have the potential to improve our results. The time needed to train models could be reduced to allow future research to train more models in the same amount of time and have a shorter feedback loop. Additionally, an optimized machine learning algorithm could be developed for code-to-code conversion.

10.3.1 Reducing the time required to train models

To find out if a change in the machine learning configuration had a positive or negative impact on our results, we had to wait for 12-24 hours. This time could be reduced when a mechanism is incorporated

that can eliminate predictions when they are only partially processed. Predictions could be eliminated when they are not compliant with the language's grammar. This could have a positive impact since models are applied to the validation set during training.

10.3.2 Optimized machine learning algorithm

To our knowledge, there are no machine learning algorithms that target code-to-code translations. An algorithm could be developed that is specialized in these code-to-code problems. This algorithm could be used by the unit test generator, while it could also be used for other general translation problems. The algorithm target the challenges that we encountered during our research. For example, the difficulty to make predictions over a long sequence length, and the difficulty to learn code structure.

Bibliography

[Abd10] Hervé Abdi. The greenhouse-geisser correction. *Encyclopedia of research design*, 1:544–548, 2010.

[AHF+17] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, 2017.

[APS16] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, 2016.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL: http://arxiv.org/abs/1409.0473, arXiv:1409.0473.

[BCS+15] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. *CoRR*, abs/1508.04395, 2015. URL: http://arxiv.org/abs/1508.04395, arXiv:1508.04395.

[BdCHP10] Jose Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pages 93–102, Washington, DC, USA, 2010. IEEE Computer Society. URL: http://dx.doi.org/10.1109/SEFM.2010.18, doi:10.1109/SEFM.2010.18.

[Bel17] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017. URL: http://arxiv.org/abs/1705.07962, arXiv:1705.07962.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[CGCB14] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. URL: http://arxiv.org/abs/1412.3555, arXiv:1412.3555.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[DUSK17] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. *CoRR*, abs/1710.11054, 2017. arXiv:1710.11054.

[FRCA17] Gordon Fraser, José Miguel Rojas, José Campos, and Andrea Arcuri. Evosuite at the sbst 2017 tool competition. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, 2017.

[FZ12] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2), 2012.

[GAG+17] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122, 2017. URL: http://arxiv.org/abs/1705.03122, arXiv:1705.03122.

[Git] Github. Rest api v3 search. https://developer.github.com/v3/search/. Accessed: 2018-06-04.

[HWLJ18] Xing Hu, Yuhan Wei, Ge Li, and Zhi Jin. Deep code comment generation. *ICPC 2018*, 2018.

[HWZD17] Po-Sen Huang, Chong Wang, Dengyong Zhou, and Li Deng. Neural phrase-based machine translation. *CoRR*, abs/1706.05565, 2017. URL: http://arxiv.org/abs/1706.05565, arXiv:1706.05565.

[JJI+14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[Juna] Junit. Junit 4 download and install. https://github.com/junit-team/junit4/wiki/Download-and-Install. Accessed: 2018-06-04.

[Junb] Junit. Junit 4 release notes. https://sourceforge.net/projects/junit/files/junit/4.0/. Accessed: 2018-06-04.

[Junc] Junit. Junit 5 release notes. https://junit.org/junit5/docs/snapshot/release-notes/. Accessed: 2018-06-04.

[KC17] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[KRV14] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014.

[LGH+16] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *CoRR*, abs/1603.06744, 2016. URL: http://arxiv.org/abs/1603.06744, arXiv:1603.06744.

[LKT09] Hui Liu and Hee Beng Kuan Tan. Covering code behavior on input validation in functional testing. *Inf. Softw. Technol.*, 51(2):546–553, February 2009. URL: http://dx.doi.org/10.1016/j.infsof.2008.07.001, doi:10.1016/j.infsof.2008.07.001.

[Maa97] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[MB89] Keith E Muller and Curtis N Barton. Approximate power for repeated-measures anova lacking sphericity. *Journal of the American Statistical Association*, 84(406):549–555, 1989.

[ND12] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[Ost02a] Thomas Ostrand. Black-box testing. *Encyclopedia of Software Engineering*, 2002.

[Ost02b] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002.

[PM17] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition-fifth round. In *IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, 2017.

[PMm17] Van Beckhoven P, Oprescu A M, and Bruntink m. Assessing test suite effectiveness using static metrics. *10th Seminar on Advanced Techniques and Tools for Software Evolution*, 2017.

[Poi] Yolande Poirier. What are the most popular libraries java developers use? based on github's top projects. https://blogs.oracle.com/java/top-java-libraries-on-github. Accessed: 2018-06-04.

[PV16] Terence Parr and Jurgen J. Vinju. Technical report: Towards a universal code formatter through machine learning. *CoRR*, abs/1606.08866, 2016. arXiv:1606.08866.

[REP+11] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[SHB15] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015. URL: http://arxiv.org/abs/1508.07909, arXiv:1508.07909.

[SSN12] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 2014.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

[YKYS17a] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[YKYS17b] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017. URL: http://arxiv.org/abs/1702.01923, arXiv:1702.01923.

[YN17] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017. URL: http://arxiv.org/abs/1704.01696, arXiv:1704.01696.

[ZM15] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[ZZLW17] Wenhao Zheng, Hong-Yu Zhou, Ming Li, and Jianxin Wu. Code attention: Translating code to comments by exploiting domain features. *CoRR*, abs/1709.07642, 2017. arXiv:1709.07642.

Appendix A

Data and developed software

The source-code, training examples, and everything else used to perform our experiments can be found on GitHub[1] and Stack[2]. An overview of the GitHub repository is given in Table A.1, and an overview of the data on Stack is given in Table A.2.

Table A.1: Overview of the GitHub repository

Folder	**Description**
Fetch projects	Contains code to crawl and filter Java projects hosted on GitHub
Last version of the Java linking applications	Includes compiled Java programs used in this research. The source of these programs can be found in the folder "Test extraction". Programs are included to fill a linking queue based on test reports, filter training examples, convert code to SBT, convert code to BPE, tokenize code, precaching to speed up the linking process, token to code converter, linking programs, SBT to code converter, training data exporter, and a tool to check if code is parsable
Linux script	Contains scripts that can be used to automate all experiments in the research. In the subfolder "Compile and Test" a file pipeline.sh is included which can be used to convert GitHub repository links to training examples. In trainingData.sh the output format can be specified (SBT/BPE/tokenized)
Machine learning	The script run.sh takes training sets as input and gives a machine learning model as output. The script predict.sh script can be used to make predictions with a model

[1] https://github.com/laurenceSaes/Unit-test-generation-using-machine-learning
[2] https://lauw.stackstorage.com/s/SdX5310wWLUvEQs

Table A.2: Overview of the Stack directory

Folder	**Description**
All files for linking server	This folder contains a zip archive that can be extracted on an Ubuntu 18.04 server. It contains the pipeline.sh script with all its dependencies in place
Compression loss models	The compression loss experiment data
Experiment S1-S20 random number 1234	Models S1 until S20 that are trained with the random value 1234
Extra models for statistics S1-S20	Models for S1 until S20
Other data	Database backups with all training examples, logs, and a flow diagram of the research
Predictions best model	Predictions (with input) of the best model
Raw data	All raw data used to generate diagrams in this thesis
SBT folders	All information for the SBT experiments
Seq2Seq server	A pre-configured machine learning server. The folder can be extracted on an Ubuntu 18.04 server, and the run.sh can be used to train models and parsableTest.sh to generate a Parsable score
Slicing	Early work on a slicing tool that can be used with linking
Suite validation	Mutation testing used to test the test suite effectiveness of our baseline
Training	Contains training examples
Unit test report generator	From the scripts to download GitHub repositories to scripts to generate test reports