Sequence to Sequence Machine Learning for Automatic Program Repair

Damir Vrabac and Niclas Svensson

***Abstract*—Most of previous program repair approaches are only able to generate fixes for one-line bugs, including machine learning based approaches. This work aims to reveal whether such a system with the state of the art technique is able to make useful predictions while being fed by whole source files. To verify whether multi-line bugs can be fixed using a state of the art solution a system has been created, using already existing Neural Machine Translation tools and data gathered from GitHub. The result of the finished system shows however, that the method used in this thesis is not sufficient to get satisfying results. No bug has successfully been corrected by the system. Although the results are poor there are still unexplored approaches to the project that possibly could improve the performance of the system. One way being narrowing down the input data to method level of source code instead of file level.**

***Index Terms*—Automatic program repair, neural machine translation, sequence to sequence, bug fix**

TRITA number: TRITA-EECS-EX-2019:156

I. Introduction

Bug fixing is a task inside software development that consumes a great amount of time and economical resources from companies in the industry [1]. Clearly there is an incentive to automate this task and thereby save economic resources for software development. Automatic program repair is the field of research where one attempts to fix both syntactic and semantic bugs in source code using external software.

The progress made in artificial intelligence and machine learning has provided new tools suitable for automatic software repair. One useful tool is neural machine translation (NMT). NMT has already shown its potential in similar tasks to automatic program repair, such as translating from different languages or speech recognition [2]. Recently these techniques have been applied to the field of automatic program repair, showing promising results. However, to implement these machine learning algorithms successfully a great amount of data is required. The gathering and the preprocessing of the data become one of the more challenging parts when using NMT.

A. Purpose

Until now the use of NMT in automatic program repair has been applied in the case where only small changes to the source code is necessary in order to fix a bug. In previous research mainly one-line-bugs are attempted to be resolved using NMT [3]. One-line-bugs are bugs that originate from only one line of defect code. However, never has one attempted to resolve multi-line-bugs using deep learning. Multi-line-bugs are bugs in source code that originate from multiple defect lines in the source file causing syntactic or semantic defects.

B. Problem Description

In this thesis, multi-line-bugs in Java source code are attempted to be automatically fixed using deep learning. Given a source file containing bugs, the system attempts to find and fix the bugs, both one-line-bugs and multi-line-bugs.

C. Limitations

In order to facilitate the task, certain criteria for multi-line-bugs are specified. Firstly, only bugs in Java source code are considered. The other criteria can be found in Table II. Also, the problem is attempted to be solved using deep learning, not considering any other potential method of solution.

II. Theory

In this section, the theory of automatic program repair is explained. Especially the theory behind the solution used in this work.

A. Terminology

1) Activation Function: An activation function turns a numerical value to a new value between 0 and 1 or -1 and 1. Two examples are $tanh$ (see Equation 1) and $sigmoid$ (see Equation 2).

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1}$$

$$\sigma(x) = \frac{e^x}{e^x + 1} \tag{2}$$

2) Softmax Function: A softmax function takes a vector $\mathbf{z} = \{z_1, z_2, ...z_k\}$ and normalizes it into a probability distribution $\mathbf{a} = \{a_1, a_2, ..., a_k\}$, see Equation 3. The new values will consequently full-fill the following criteria: $\sum_i^k a_i = 1$ and $0 \leq a_i \leq 1$, where $i = 1, 2, ..., k$.

$$a_i = \frac{e^{z_j}}{\sum_{i=j}^{k} e^{z_j}} \tag{3}$$

3) SOS and EOS: Two tokens $< SOS >$, indicating start of sequence, and $< EOS >$, indicating end of sequence are used by the model. These tells the decoder of the model when it should start a prediction and when a prediction has reached its end.

4) Largest Common Sequence: Let S_i be a sequence of tokens so that two files, f_1 and f_2 contains exactly the sequence S_i. The *largest common sequence* between these two files is then defined as the largest S_i that exist with respect to number of tokens. For example the following two strings have *AGBD* as the *largest common sequence*.

$$B\ C\ A\ G\ B\ D\ E\ F$$

$$A\ G\ B\ D\ H\ F$$

B. Automatic Program Repair

Automatic program repair is about modifying the source code of a defected program and creating a functioning program that works as intended. In automatic program repair, this is done without a human in the loop to help the rectification [4]. The process can be represented as $S : B \rightarrow F$, where S is the system that fixes buggy source code, B is the set of buggy source codes and F is the set of fixed source codes. The main task of automatic program repair becomes to design the system denoted by S.

There are multiple ways to approach the problem of automatic program repair. One already existing example is GenProg, a generic method for automatic software repair [5]. To solve the problem one attempts to create a system S, that inputs a defect source file, or a segment of a source file, together with a set of test cases. The test cases describe how a program shall behave given certain inputs. GenProg then iteratively applies different code patches, from a large set of code patches, on the original program. If the applied code patches cause the original program to pass additional test cases, the applied patch receives a higher fitness. The iterative process is stopped when a code patch, that causes the program to pass all test cases, is found. In addition to GenProg there are several methods that build upon the same idea but with certain modifications, e.g. RSRepair and CapGen [6] [7].

In this thesis however, automatic program repair is approached using a sequence to sequence based solution that is further described below.

C. Sequence to sequence

The problem of automatic program repair is attempted to be solved using deep learning. However, deep neural networks have the disadvantage of only being usable when the in- and output can be encoded to vectors of definite size. Source code can be of varying size and bugs can be fixed by either adding or removing code. Hence it is difficult to encode these sequences to vectors of definite size. This is an obvious problem, and accordingly standard deep neural networks are not sufficient in the case of automatic program repair [2].

The problem can be broken down and described as trying to map one sequence to another where the lengths of the different sequences are indefinite. The target sequence can possibly be longer or shorter than the source sequence. This is a similar problem to translating between different languages. One way of approaching the sequence to sequence problem is to use the sequence to sequence architecture, commonly used in the field of NMT [8].

The sequence to sequence model contains two blocks, the encoder and the decoder. Each block is implemented by a recurrent neural network (RNN). The encoder takes a sequence of tokens from buggy source code as input and encodes it into vectors, using an embedding layer. The decoder then utilizes these vectors to calculate the output sequence. Figure 1 roughly illustrates how the sequence to sequence operates. Apart from the main architecture some additional layers and mechanisms are added on top of the model to further improve its performance. These being the attention layer and a copy mechanism. All building blocks of the complete system is described below. The model is trained on a data set using backpropagation. That is the process where all the parameters of the neural networks are adjusted to optimize the performance of the model.

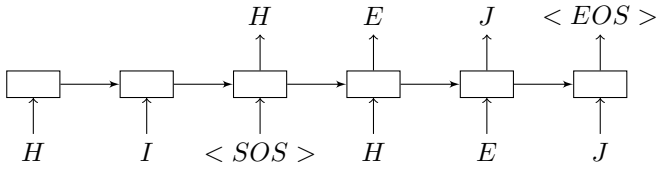

Fig. 1. A simplified illustration of a sequence to sequence model translating from English to Swedish. The horizontal arrows represent the flow of internal states of the model.

1) Embedding layer: In order to make it possible to pass tokens in to the model, they have to be translated into vectors. These vectors contain elements that can be handled by the neural network, typically floating point numbers or integers.

Word embedding is an encoding technique that learns a representation of tokens and retain their semantics. In natural language this can be thought of as the words *fork* and *knife* that have a closer semantic meaning then the words *fork* and *ball*. Thus, an embedding layer that uses word embedding allows the model to find similarities between tokens in the language and make more accurate predictions. The accuracy of the prediction is gained because the model can find similar tokens although it has not seen them as much during training as some other.

One alternative to word embedding is one-hot encoding. Each token is encoded to a vector corresponding to one row in matrix $\mathbf{O}$, see below. Where $\mathbf{O} \in \mathbb{R}^{|T|\times|T|}$ and T is the set of all tokens in the vocabulary. This method does not preserve the semantic meaning of each word since the hamming distance between two randomly chosen vectors always is 2 and the euclidean distance is always $\sqrt{2}$. More intuitively, the relation between all tokens are the same geometrically and therefore the semantic differences are the same as well.

$$\mathbf{O} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

The embedding layer is implemented as a matrix $\mathbf{E} \in \mathbb{R}^{|T|\times d}$ where T is the set of all tokens in the vocabulary and

d is a chosen dimension that the tokens are represented in [9]. Each token in the vocabulary is then mapped to a continuous sequence of positive integers, where the integer represents an index of one unique row in matrix $\mathbf{E}$. The vocabulary is commonly chosen as the most frequently appearing tokens in the training set. Tokens not included in the vocabulary set $\mathcal{V}$ is mapped to a token $< UNK >$. Each token is accordingly encoded into a vector of decimal numbers. An example of the embedding matrix $\mathbf{E}$ is shown below where the dimension of embedding space d is 3. Each token can thus be represented in 3-dimensional space as a dot, see Figure 2.

$$\mathbf{E} = \begin{bmatrix} 0.41 & 0.53 & 0.01 \\ 0.21 & 0.31 & 0.81 \\ \vdots & \vdots & \vdots \\ 0.15 & 0.90 & 0.67 \end{bmatrix}$$

The main advantage of using word embedding is that tokens with similar semantically meaning are trained to be situated close to each other in the d-dimensional embedding space, see Figure 2. All dots of one colour (or shape) are tokens with similar semantic meaning and as illustrated the euclidean distance between these points are relatively short and consequently the semantics of a token is preserved.

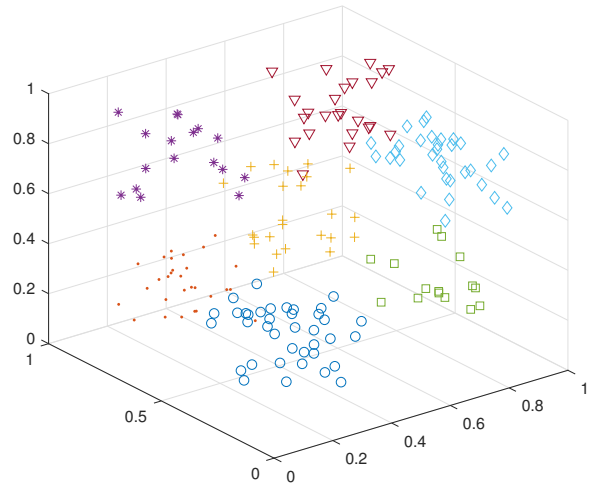

Fig. 2. The embedding illustrated in 3-dimensional space.

The final sequence to sequence model contains an embedding layer placed before the RNN. See Figure 3, where the blocks containing e represents the embedding layer. The embedding layer firstly works like an encoding technique. Another advantage of this encoding technique, compared to one-hot encoding, is that the vocabulary of the model can be increased without having to increase the size of the input-layer of the first RNN. Since the embedding vectors being fed to the model have a lower dimension than one-hot encoded vectors, it means that the model do not have to be as big. This makes the embedding technique more memory efficient and it requires less calculations when predicting an output sequence.

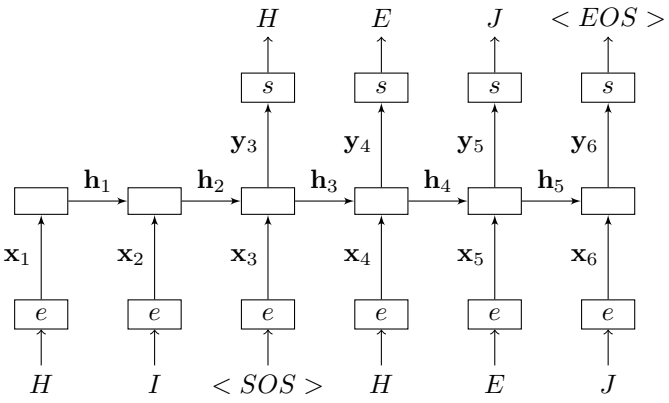

Fig. 3. Embedding layer added to the sequence to sequence model.

2) Recurrent Neural Networks: A recurrent neural network is a neural network that computes an output sequence of tokens $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_m\}$, given an input sequence of tokens $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$. Where m and n do not necessarily have to be equal.

The calculations performed by an RNN is represented by the matrices W^{hx}, W^{hh} and W^{yh}. Once the network has been trained, the vectors $\mathbf{x}_i$ are then passed to the RNN one by one. For each vector $\mathbf{x}_i$ the RNN computes a hidden state vector $\mathbf{h}_i$ according to Equation 4, using the previous hidden state vector $\mathbf{h}_{i-1}$ and the current token in the sequence $\mathbf{x}_i$ [8]. One can say that the hidden state vector contains information about the previous tokens in the sequence. This causes the prediction to not only be conditioned on the previous token, but also on the entire previous sequence. The output vector $\mathbf{y}_i$ is then calculated from the hidden state vector $\mathbf{h}_i$ according to Equation 5 [8].

The function ϕ is an activation function applied element wise, see Equation 1 or 2.

$$\mathbf{h}_i = \phi(W^{hx}\mathbf{x}_i + W^{hh}\mathbf{h}_{i-1}) \tag{4}$$

$$\mathbf{y}_i = W^{yh}\mathbf{h}_i \tag{5}$$

This process is then repeated according to Figure 4, where the blocks containing r represents the RNN. By utilizing the latest output as the next input, a sequence can be predicted given an input sequence, see Figure 1.

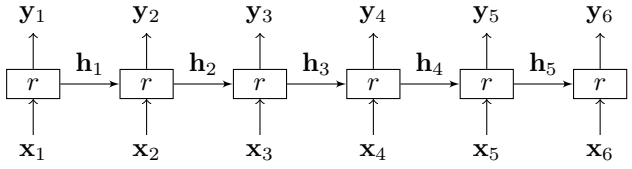

Fig. 4. The flow of information in a recurrent neural network.

Once the $\mathbf{y}_i$ has been calculated, see Equation 5, the vector has to be translated into a token. This is done using a softmax function, see Equation 3. The $\mathbf{y}_i$ has the same size as the model's vocabulary. The vocabulary is the words the model is trained to recognize. The vector contains a probability

distribution telling what token in the vocabulary is the most likely to be predicted as the next token in the sequence. Thereafter it is translated to that token. In Figure 3 the boxes denoted by s represents the softmax function.

For an example, the RNN in the decoder initially gets a hidden state vector from the encoder and an input indicating start of sequence $< SOS >$, see Figure 1. Thereafter, a new hidden state vector is calculated as well as a prediction is made of a token for the repaired source code. Moreover, the predicted word is used as input for the next prediction and the decoder keeps predicting the sequence token by token until it predicts end of sequence $< EOS >$.

3) Attention layer: Until this point, when the sequence to sequence model have been trying to predict the next token in the sequence, the previous hidden state have been used [10]. However, when trying to predict the first token of the target sequence the latest hidden state might not be the most suitable to use. To predict the first token it is more appropriate to use the first hidden state vector since it contains the information of the beginning of the sequence. This is the reason for why the attention layer is introduced. Figure 5 will be used as an example to explain the theory of the attention layer. The blocks denoted with an a represents the attention layer.

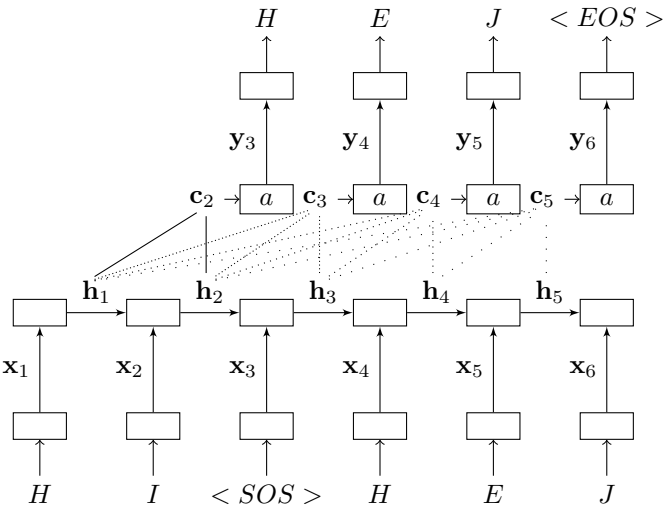

Fig. 5. Attention layer added to the sequence to sequence model.

When inputting a new token to the sequence to sequence model each previous hidden state vector $\mathbf{h}_i$ will be assigned a score s_i. The score will tell how much attention each of the previous hidden states will get when predicting the next token. In the example in Figure 5, when the $< SOS >$ token is inputted, s_1 will be bigger than s_2, the reason being that the hidden state vector $\mathbf{h}_1$ contains more information about the beginning of the sequence than $\mathbf{h}_2$. The s values together from an $\mathbf{s}$ vector.

Then, a softmax function, see Equation 3, is applied upon the $\mathbf{s}$ vector generating a vector $\mathbf{a}$ containing the attention weights a_i. A context vector is then calculated according to Equation 6 [11].

$$\mathbf{c}_n = \sum_{i=1}^{n} a_i \mathbf{h}_i \tag{6}$$

The decoder will then use the context vector, instead of the previous hidden state vector, to predict the next token in the sequence, see Equation 5.

When translating between different languages the sequence length seldom exceeds 20 words. Therefore, the need for an attention layer might not be very urgent. In the case of automatic program repair though, especially when supplying entire source code files, the amount of tokens in an input sequence can be large. The need of an attention layer is therefore greater.

4) Copy Mechanism: Source code contains several user-defined variables. However, it does not exist any well defined conventions for the variable names and these differ a lot between software projects. For a system to be able to fix bugs it has to deal with these unrecognized tokens. One solution for this problem is to use a copy mechanism. A copy mechanism is a model that processes the input sequence before the main bug-fixing system. The copy mechanism learns to decide which input tokens should be copied to the output sequence and which tokens should go through the main system.

With a copy mechanism there is not just a set $\mathcal{V}$ that contains tokens that the model has identified patterns for, which are usually a part of the programming language. There is now a set $\mathcal{X}$ which contains each unique token from the input sequence as well. The model then makes a prediction whether to copy the token or generate one, using Equation 7. The value p_{gen} are calculated, where $\mathbf{w}_\mathbf{c}^T$, $\mathbf{w}_\mathbf{h}^T$, $\mathbf{w}_\mathbf{x}^T$ and b_{ptr} are trainable parameters. Depending on the value of p_{gen}, the model decides whether it should copy or generate. A copy is made when the token is not in the models vocabulary. The copy-mechanism used in the project is described in [12].

$$p_{gen} = \sigma(\mathbf{w}_\mathbf{c}^T \mathbf{c}_i + \mathbf{w}_\mathbf{h}^T \mathbf{h}_i + \mathbf{w}_\mathbf{x}^T \mathbf{x}_i + b_{ptr}) \tag{7}$$

5) Backpropagation: W^{hx}, W^{hh} and W^{yh}, see Equations 4 and 5, are matrices that represent the calculations that the neural network performs. Consequently, these are generated and adjusted during the training process. The model is, during the training process, given a source sequence and a target sequence from the training set. Given the source sequence the model knows what the target sequence should be. Originally, the values of the matrices are random and hence the predictions will not be very accurate. The training starts by the model making a prediction with a source sequence from the training set. The prediction is most likely awful but since the model knows what the prediction is supposed to be it calculates the loss, a measurement of the quality of the prediction. Using the loss, a gradient can be calculated. By using a vector in the opposite direction to the gradient one can adjust the values of the matrices in the network. Training continues by working its way through the entire training set. If the settings of the model are good the loss will converge towards a value close to 0. When the loss changes marginally the training can be stopped. This process is called backpropagation.

III. Method

Underneath is the approach of solving the problem of automatic program repair described.

A. Data gathering

The approach of using a sequence to sequence model requires a great amount of training data in order to get satisfying results. However, the gathered data have to be of a certain quality and full-fill certain criteria so that the sequence to sequence model can be trained to fix the specified bugs. Therefore, the data had to be selected and filtered carefully.

The source of training data was open source projects on the version handler GitHub. The advantage of using GitHub is that the entire commit history of a certain project is saved and accessible through different API's. The initial selection of open source projects was made so that good code quality was guaranteed. The GitHub filter was set to only search for projects containing java source files. Then the projects were sorted according to their star rating. From this list the top 10 projects containing between 4.500 and 50.000 commits as well as over 70% of java source files were selected and cloned. The chosen projects are listed in Table I.

TABLE I
THE OPEN SOURCE JAVA PROJECTS CHOSEN TO COLLECT TRAINING AND TEST DATA FROM

	Open source java projects	Nr. Commits
1	elasticsearch	43437
2	ExoPlayer	5413
3	guava	4834
4	jenkins	27363
5	mockito	4744
6	presto	14970
7	realm-java	7376
8	redisson	4534
9	RxJava	4956
10	spring-boot	18296
	Total:	119523

B. Data preprocessing

After having cloned all the projects the filtering process could be started. Initially the files containing the file changes of each commit had to be extracted. This was performed using a shell script. An example of a file change is illustrated in Figure 6.

But using all commits from a project is not a good idea since every commit does not necessarily contain bug fixes. They can be large additions or subtractions from a project as well. In addition they can contain changes in non java-files. Therefore, a filter reading the commit message of each commit was created. All commits who's commit message contained the word 'fix', 'issue' or 'bug' were kept. Also the file extension of each file that had been changed in every commit was checked. If it was not a java-file it was deleted from the data set.

A decision concerning what bug types that should be targeted by our model had to be made. The bugs that are attempted to be fixed are specified in Table II.

The remaining data, a total of approximately 200k file changes, were then split into bug-fix-pairs. One file containing

```
 public class HelloWorld {
   public void main(String[] args) {
     int a = 2;
     test()
+    System.out.println("Hello World");
-    System.out.println("Hello World")
   }
   public int test() {
     return 1;
   }
 }
```

Fig. 6. An example of a file change.

```
public class HelloWorld {
  public void main(String[] args) {
    int a = 2;
    test()
    System.out.println("Hello World")
  }
  public int test() {
    return 1;
  }
}
```

Fig. 7. The source-file created from Figure 6.

```
public class HelloWorld {
  public void main(String[] args) {
    int a = 2;
    test()
    System.out.println("Hello World");
  }
  public int test() {
    return 1;
  }
}
```

Fig. 8. The target-file created from Figure 6.

TABLE II
BUGS THAT THE SYSTEM TARGETS

	Bugs
1	One line bugs. Bugs that are caused from defects on one line in the source code.
2	Multiple uncorrelated one line bugs in the same source file.
3	Continuous multi-line bugs with maximum of 10 lines.
4	Multiple continuous multi-line bugs.

the buggy code (see Figure 7) and one file containing the fixed code (see Figure 8). This was essentially the foundation of the training set.

The data then had to be tokenized into sequences. The python library Javalang[1] where used to tokenize the java source files. In addition to tokenizing the source code, the comments where removed as well, leaving a result of sequences containing the building blocks of the java programming language separated from each other. In total a training set of 191260 file changes with different lengths remained after extracting a validation set. The data set is separated into token length and illustrated in Figure 9. In Figure 10 the differences between the boxes in Figure 9 are clarified. If the amount of files are negative it means that the target box is smaller than the source box. Figure 11 describes common sequence score, see Equation 8, but between target and source file instead of between target and prediction file. This gives an indication of how similar the target and source files are and therefore it

[1]https://github.com/c2nes/javalang

illustrates how big the required changes are to fix multi-line bugs.

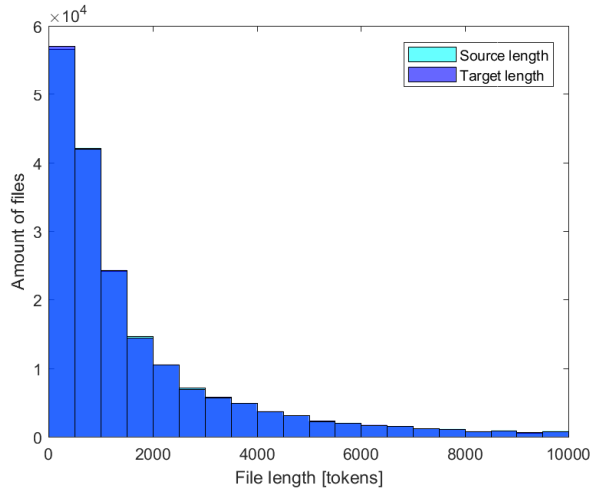

Fig. 9. A histogram of the final training data set. File changes separated into intervals of token length.

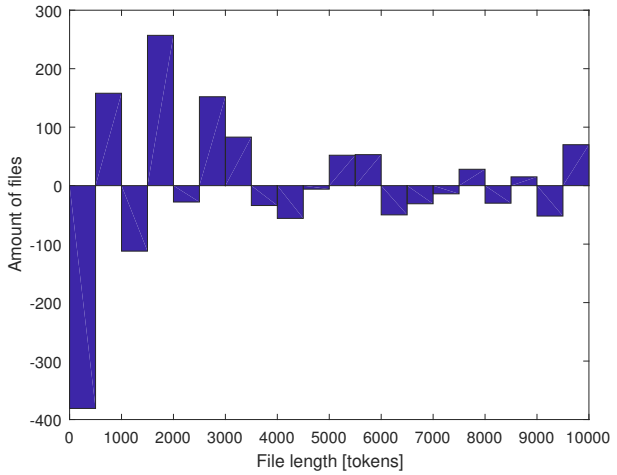

Fig. 10. A histogram illustrating the difference between the boxes in Figure 9.

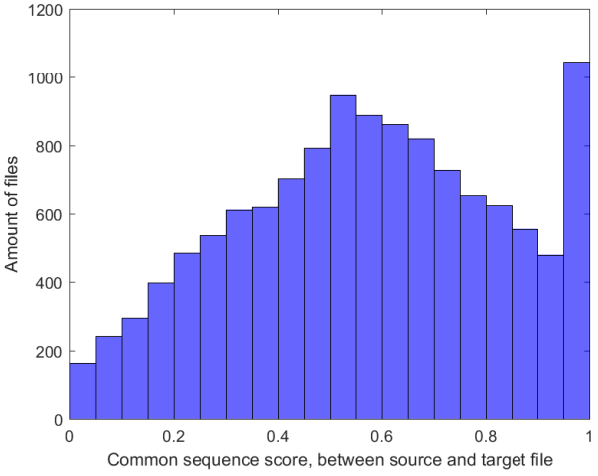

Fig. 11. Common sequence score calculated between target and source file.

Finally a max sequence length, in terms of tokens, is set for both the source sequences and the target sequences, shrinking the training set by approximately half. 98502 bug-fix-pairs remained. Only the file changes in the first three intervals in Figure 9 were kept. The maximum source length is 1000 tokens and the maximum target length is 1500 tokens. The vocabulary of the model is set as well. The 1000 most common tokens are included in the vocabulary meaning that the model is only trained to recognize these. The tokens not included in the vocabulary is handled by the copy mechanism. Since the system is not translating between two different languages the source vocabulary is set to be the same as the target vocabulary.

C. Machine learning

To implement the sequence to sequence model, described in the theory, OpenNMT-py where used [13]. This Python library is a machine learning framework built upon the PyTorch API. It facilitates the implementation of deep learning algorithms and especially NMT using sequence to sequence models.

In Table III the settings of our sequence to sequence model is specified, separated into the encoder and the decoder. Apart from these setting, the word embedding size is set to 256, meaning that each word in the vocabulary is mapped to a vector of dimension 256. The tool for adding an attention layer as well as a copy mechanism is also used.

TABLE III
SETTINGS FOR THE SEQENCE TO SEQUENCE MODEL.

Encoder	
Type	RNN
Hidden states	256
Layers	2
Embedding size	256
Decoder	
Type	RNN
Hidden states	256
Layers	2
Additinal settings	
Optimizer	Stochastic gradient decent
Copy mechanism	
Attention layer	

D. Evaluation

In order to make an evaluation of the system's performance, a test set had been created as well. From the original data set a total of 12579 files were extracted, together forming the test set. The test set contained files from all open source java projects, equally distributed. The model is not trained on the test set. A validation set was used as well. The model used it during the evaluation process of the training.

The performance of the model was then measured accordingly: Every source file in the test data set were run through the sequence to sequence model. Then the output was compared to the corresponding target file. If the target file is exactly the same as the output of the model, the test is successful. If the output is not the same as the target, the test has failed. The percentage of successful tests is then the measurement of the model's performance.

Another measurement has been used by assigning a score for each prediction that the model made on the test set. These scores were then evenly weighted. The score *Common sequence score (CSS)* was calculated according to Equation 8.

$$CSS = \frac{len(\textit{largest common sequence})}{max\{len(\textit{target file}), len(\textit{prediction file})\}} \quad (8)$$

Where *largest common sequence* is the largest common sequence that the *target file* and *prediction file* contain and len is the function that returns the number of tokens that the input contains.

To further evaluate the system a simple grammar test was made on the predicted files using the test set. The grammar was tested by controlling, in how many cases the system's predicted files follows the grammar rule of closing parentheses, brackets and braces for each time opening them.

IV. Result

The resulting scores of the system that have been obtained from this work are presented in Table IV. Each score is calculated by the results from the test set. The scores clearly show how bad the system works and that it is too difficult for it to learn from whole files and fix them.

TABLE IV
SCORES FOR THE MODEL.

Score name	Score
Amount of fixed files	0.0%
Common sequence score	1.3%
Average length difference	1920 tokens
Relative average length difference	53%

The frequency distribution of the *Common sequence score* is illustrated in Figure 12. The figure shows a peak at a CSS of approximately 0.018 with approximately 1500 files. The frequency distribution that is seen in the figure may be explained both by the system actually having learned something but also by the copy mechanism that preserves many of the user-define variable names.

The simple grammar test is presented in Table V. The results from the test can be interpreted as the system learns some syntactical rules of the Java programming language that is being used.

TABLE V
SIMPLE GRAMMAR TEST OF THE MODELS PREDICTED FILES

Amount of equally many opening as closing ...	Score
parentheses	62%
brackets	91%
braces	18%

V. Discussion

Given the results of the system we have shown that a state of the art solution to the problem is not sufficient to fix bugs when supplying a complete tokenized file to the system. The results of the final system are completely unusable since the system more or less destroys the code. The generated code contains

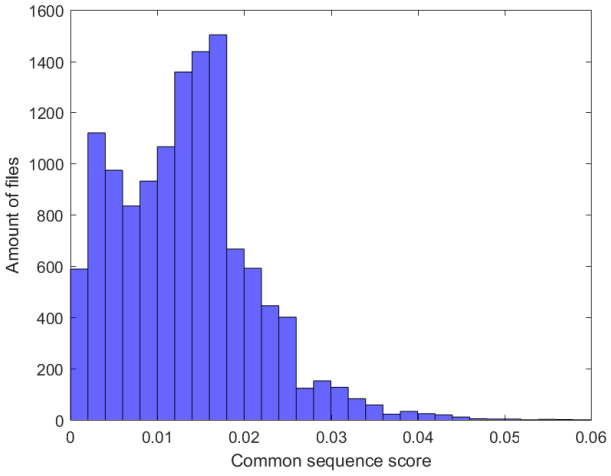

Fig. 12. Common sequence score calculated between target and prediction file.

in almost every case more bugs than the original source code, both syntactical and semantic bugs. The prediction is not close to the original code, which can be seen by the results, and will with all certainty not even compile.

The bad predictions to fix code that are made by the system is mostly explained by the enormous inputs that are being feed to the system. In the work [14], methods have been used as input to the model and a data set of 2.3 M methods to train it. That work achieved a result of 9% correctly fixed bugs. Hence, it is clear that a model that is being feed by whole files with a data set of barely 100k files, will not be able to produce any valuable results.

Although the results are bad, Table V shows that the system has learned some syntactical rules. This is especially the case for brackets. One way of explaining why the system had it the easiest for brackets and hardest for braces is that brackets are usually used for indexing. The opening as and the closing brackets are therefore close to each other and may therefore be easier to identify patterns for. While on the other hand, the braces are usually used for enclosing classes or methods. Therefore, the opening and the closing braces can be far apart which may make it harder for the system to learn the syntactical patterns for them.

Most of the work in the field of automatic program repair is being done on one-line-bugs. Some examples of this is [2], [3] and [4]. There has been some work on multi-line-bugs, in [14] methods with less than 100 tokens was considered. However, there has never been any study on multi-line-bugs using whole files. In this work, it has been shown that approaches that uses state of the art technique is not sufficient for generalizing to whole files. New architectures are needed for this to be possible.

A. Future Work

To further explore how general an automatic program repair system can become with the state of the art technique one could divide the source codes into smaller blocks. Thereby, it

would be easier for the system to identify patterns between a buggy and a fixed source code. The next level of building blocks of source code are classes. However, classes are in most cases also very big and it may be too hard for the system to learn any fixing patterns with these inputs as well. Therefore, an obvious future work is to examine how well an automatic repair system would be able to generalize and perform on method-level inputs. Evaluating automatic program repair systems on their performance with different building blocks as input is essential. Larger building blocks as input gives the model more information and the possibility to find hidden bugs that are spread through the whole system.

Another interesting future work could be to examine how the same approach as in this work would perform but with lots of more data and a greater machine learning model. A model that has more parameters to learn and that trains on a much larger data set could find more learning patterns and therefore possibly perform much better. However, this would also imply that lots of more computational resources would be needed than was used in this work.

B. Related Work

It has been done extensive amount of work in natural language processing, in particular machine translation with neural networks. This work is closely related to the work of that. Many techniques that have been applied originate from this area of research. The work done in [8] is an example of how well RNN can perform on sequential problems. The copy mechanism that has been used in this work was also first introduced in the field of natural language processing and presented in [11].

The research area of automatic program repair is fairly new and unexplored. One work that inspired this work is [14], the work shows a good approach for developing an automatic program repair system. In that work the model is being feed by method-level inputs with a data set of 2.3 M methods and the achieved result was 9% correct predictions.

The work done in [2] has shown remarkable results in the field of automatic program repair. The system that was developed in that work generated syntactically correct code in 98.7% of the time. The developers that tested the system experienced it as helpful with fixing bugs even when the fixes was not completely correct. However, the approach was different from this work and the fixes were done on one line. A work with a similar network architecture was presented in [3]. That work introduced the possibility to use the copy mechanism on source code and its effectiveness.

VI. Conclusion

The purpose of the thesis was to investigate if a state of the art system for automatic program repair is suitable for multi-line-bug-fixing. A system was implemented using data gathered from GitHub and neural machine translation tools. The result of the thesis however, clearly show that a state of the art solution to the problem of automatic program repair of multi-line bugs is not sufficient when supplying complete tokenized files to the system. In order to improve the system's performance, next step would be to investigate how the system performs when methods are supplied to the system instead of complete source files.

Acknowledgment

The computations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at the university in Umeå. Therefore, the authors would like to thank them. The authors would also like to thank the supervisor of the project Martin Monperrus and the co-supervisor Zimin Chen for their guidance. Their profound knowledge in the area has been very helpful and the project would never have been possible without their support.

References

[1] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502041

[2] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *CoRR*, vol. abs/1812.07170, Dec. 2018. [Online]. Available: http://arxiv.org/abs/1812.07170

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

[4] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *CoRR*, vol. abs/1707.04742, Jul. 2017. [Online]. Available: http://arxiv.org/abs/1707.04742

[5] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, Feb. 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6035728

[6] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, Jun. 2014, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254

[7] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, Jun 2018, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180233

[8] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, Dec. 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[9] Y. Goldberg, *Neural network methods for natural language processing*, ser. Synthesis Lectures on Human Language Technologies ; 37. S.l.]: Morgan & Claypool Publishers, 2017, vol. 10, no. 1. [Online]. Available: http://portal.igpublish.com/iglibrary/search/MCPB0000900.html

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

[11] J. Gu, Z. Lu, H. Li, and V. O. K. Li, "Incorporating copying mechanism in sequence-to-sequence learning," *CoRR*, vol. abs/1603.06393, Jun. 2016. [Online]. Available: http://arxiv.org/abs/1603.06393

[12] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *CoRR*, vol. abs/1704.04368, Apr. 2017. [Online]. Available: http://arxiv.org/abs/1704.04368

[13] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," in *Proc. ACL*, Mar. 2017. [Online]. Available: https://doi.org/10.18653/v1/P17-4012

[14] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *CoRR*, vol. abs/1812.08693, Dec. 2018. [Online]. Available: http://arxiv.org/abs/1812.08693