Automatic Correction of Self-Introduced Errors in Source Code

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sven Kellenberger

16.08.2018

Leiter der Arbeit:
Prof. Dr. Paolo Favaro
Institut für Informatik

Abstract

This thesis covers the implementation of an LSTM based sequence-to-sequence model with an attention mechanism on top as well as the difficulties of introducing errors into source code that have the requirement to be as close to reality and as sophisticated as possible. The model is then trained to detect and correct these artificially introduced syntax, semantic and logic errors. This thesis also provides experimental evaluations of several architectural design choices and a detailed analysis of the introduced errors. The model achieves promising results for all generated errors and manages to gain some understanding of the semantics of source code. However, it could not be conclusively determined if the model is also able to learn to detect and correct logic errors.

Contents

Chapter 1

Introduction

1.1 Motivation

Automatic text correction is a ubiquitous technology in our today world. Every smartphone, every word processing software, every browser provides some form of spelling error detection for text input and these tools have proven to be very useful for both language learners and native speakers. These systems usually rely on a dictionary of correct words [9, 10] or some machine learning algorithm [33] to find errors and suggest possible corrections.

Source code is very sensitive to errors of all kinds and therefore a similar functionality is desirable for code editors. We distinguish between three types of errors in source code: syntax, semantic and logic errors (see Table 1.1). The syntax of a programming language is strictly defined which enables integrated development environments (IDEs) to detect syntax errors with the help of a parser or a suitable algorithm before the program is even run. However the capability of an IDE to detect more sophisticated errors is limited by the properties of the programming language, e.g. is it strongly typed or weakly typed. In general, the error detection in source code is mostly limited to syntax errors, while semantic and logic errors show only at runtime or go completely unnoticed. However, in a strongly typed language like Java, a lot of semantic errors can be detected because each variable has to be declared and initiated before it is used. The type of the variables is known at all times and therefore also their available attributes and methods. In weakly typed languages like Python on the other hand one can generally not determine what type of value a variable holds before runtime. This prevents an algorithm from detecting most semantic errors before runtime. Logic errors are in general impossible to find with a traditional algorithm because their detection requires knowledge of the purpose of the program and even humans struggle with this task. A model which is able to detect possible logic errors would therefore be extremely useful.

This is where machine learning algorithms could step in. In the past years, deep neural networks have proven to be very effective in learning generalized concepts and applying them to individual examples. For example in [28] a network is trained to transfer the style of a painting to a video sequence. Neural networks can also be trained on the tasks of object instance segmentation in images [16] or face recognition [1]. Another branch of research applies neu-

Error Type	Definition	Example

syntax	Violation of the specified syntax of the programming language.	`public int add(int a, int b){`
		`  int sum := a + b;`
		`  return sum;`
		`}`

semantic	Incorrect usage of a variable or statement.	`public void add(int a, int b){`
		`  int sum = a + b;`
		`  return sum;`
		`}`

logic	Failure to comply with the programs requirements.	`public int add(int a, int b){`
		`  int sum = a - b;`
		`  return sum;`
		`}`

Table 1.1: The definition of syntax, semantic and logic error in the context of programming languages.

ral networks to natural language processing problems such as neural machine translation [30], keyboard text decoding [11] or automatic text correction [33]. These results suggest that it should also be possible to train a network on the detection and correction of more sophisticated errors in source code.

The goal of this thesis is it to train a character based sequence-to-sequence model [30] on the task of source code correction. The model consists of an encoder network and a decoder network both of which are LSTMs [15]. On top of that, an attention mechanism [21] is used to allow the decoder to focus on important parts of the input sequence. Sequence-to-sequence is a technique originally proposed for the problem of neural machine translation but it can also be applied to any kind of problem where one sequence needs to be mapped to another sequence. The advantage of this architecture is that the model gets to see the whole input before producing any output which is necessary to correct certain errors in source code. By operating on the character level the network is also not limited to a fixed vocabulary of tokens which would be infeasible for source code.

The implementation of the model is based on the neural machine translation model provided by Tensorflow [20] and the Java Github Corpus [2] is used as a source of correct data. This data is then corrupted as random syntax, semantic and logic errors are added. A dataset of real-life examples would be preferable over these self-introduced errors but there simply is none. The difficulty for the automatic generation of these corruptions is that they have to be as sophisticated and as close to reality as possible which is not easy to do. Especially logic errors cannot be automatically generated without some knowledge of the purpose of the code. The approach selected in this thesis is to choose a corruption which has a high probability of generating logic errors and which can be introduced without some understanding of the purpose of the code.

1.2 Outline

This thesis is divided into five chapters, including this introduction. In the second chapter, an overview of the existing work on the task of spell checking and spelling correction is given and also of some research on error detection and correction in source code. The third chapter starts with a short introduction to the machine learning techniques and architectures used in this thesis. It also provides a description of the used model, the training procedure, the dataset construction and the corruption of the input. In chapter four the experiments are explained and the obtained results analyzed. Chapter five provides the conclusion and suggestions for future work.

Chapter 2

Background

2.1 Origins of Spelling Checkers and Correctors

With the emergence of word processing programs and the following digitalization of text documents, spelling checkers and correctors have become a useful helper in our everyday lives. However, the research on this topic has begun much earlier [25]. The original motivation for a spelling checker was to find input errors in databases. For example, in [6] the authors aim to find incorrectly spelled names, dates and places in a genealogical database. This was done by computing the frequency of trigrams (three sequential characters) in the source text and based on that the probability of a character occurring given its context, i.e. its adjacent characters. Erroneous words were found by looking at its trigrams. If a word consisted of a number of unusual character combinations, it was probably spelled wrong. However, it is easy to see that this method is not very useful for rare or foreign expressions like `doppelgänger` and for typos with high probabilities of being correct. Furthermore, the model is limited by the vocabulary of the dataset used to computed the trigrams.

These problems were solved by the introduction of dictionaries. A dictionary is a list of correctly spelled words which can optionally be extended by the user. For every word, the program checks if it is part of the dictionary. If it is then it is spelled correctly otherwise there is an error.

This method was enhanced by the addition of a spelling corrector. In [10] a dictionary is used to find incorrect words. It is then assumed that these words contain only one of four types of errors: one character was wrong, one extra character was inserted, one character was missing or two adjacent characters were transposed. Under this assumption, the dictionary is searched for possible corrections. In [9] the author uses a dictionary to find incorrect words as well. For the found words he uses digrams (similar to the trigrams mentioned above, just with two instead of three characters) to suggest corrections for incorrect words.

The addition of user interaction results in even more convenience. Instead of just outputting a list of incorrect words and corrections, the program would show the user the words it assumes to be incorrectly spelled and then give the user some possible actions to chose from like `ignore` or `replace` (compare Figure 2.1).

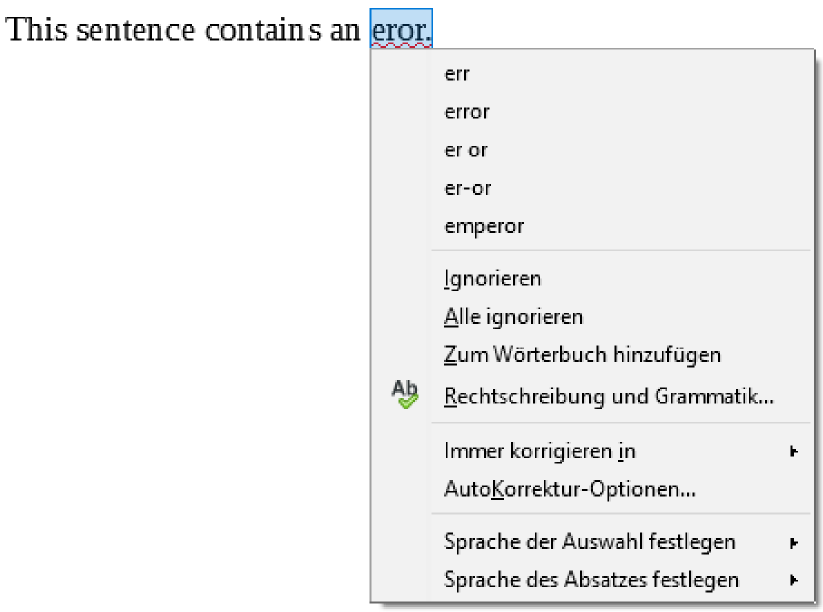

This sentence contains an eror.
err
error
er or
er-or
emperor
Ignorieren
Alle ignorieren
Zum Wörterbuch hinzufügen
Rechtschreibung und Grammatik...
Immer korrigieren in
AutoKorrektur-Optionen...
Sprache der Auswahl festlegen
Sprache des Absatzes festlegen

Figure 2.1: Screenshot from the LibreOffice Writer program. An example of error correction in a word processing software. The misspelled word is flagged and possible corrections are proposed to the user.

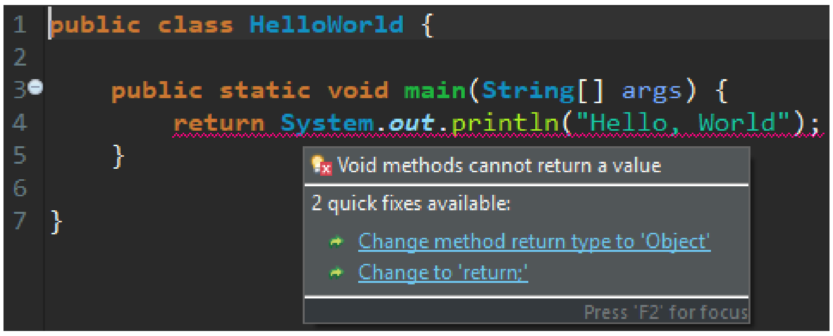
2.1.1 Modern Research

Of course, the methods described so far do still not take the context the word is in into account. For example, if `know` is misspelled as `now`, no error is indicated even though it could be concluded from the context that a verb is expected in this place. This "context-awareness" has been the topic of a lot of recent research. In [5] n-grams are used to determine the most likely replacement for an error and in [12] the authors concentrate on finding and correcting small spelling errors that result in correct words and are therefore not detected by a traditional spelling checker (e.g. `to` for `too` or `there` for `their`). Other researchers concentrated on a particular set of errors like article [13] or preposition errors [27].

Even though these methods perform well on the errors they were designed for, a large number of different classifiers would be needed to catch every error. This is a costly and inflexible approach. Therefore recent research often uses statistical machine translation methods or language models and n-grams to correct errors of multiple classes [31]. In [33] the authors train an encoder-decoder neural network with an attention mechanism which operates at the character level to correct errors in text and in [11] a similar approach was chosen but applied to keyboard decoding on smartphones.

2.2 Automatic Correction of Source Code

Of course, automatic error correction is also a useful helper for writing code. Because of the well-defined syntax of a programming language, syntactical er-



```
public class HelloWorld {

    public static void main(String[] args) {
        return System.out.println("Hello, World");
    }

}
```

Void methods cannot return a value

2 quick fixes available:

Change method return type to 'Object'

Change to 'return;'

Figure 2.2: Screenshot from the Eclipse Java IDE. An example of error correction in an integrated development environment. The erroneous code is flagged and possible quick fixes are proposed to the user.

rors are relatively easy to find using a parser or a suitable algorithm. This functionality can help novice programmers to avoid typical beginner's mistakes like missing a semicolon at the end of the line. However, semantic and logic errors can usually not be detected this easily.

Of course, the correctability of a programming language depends in part on its properties. One important distinction is between strongly typed and weakly typed programming languages [19]. In a strongly typed programming language, every variable has a fixed type and every method has a fixed return type. This enables an editor program to check if the expected and the actual type match before runtime. This error can then be flagged and shown to the user. In contrast, in a weakly typed programming language, variables don't have a fixed type. They can contain whatever value one assigns to them and similarly, methods can take arguments of any type and also return values of any type. In this case, an error stemming from a wrong parameter type shows only at runtime. For example, on the one hand, the type error in

```
public void printNumber(String n){
  System.out.println("Number: " + n);
}
printNumber(9);
```

can be detected before runtime because Java is a strongly typed programming language. The type of the parameter `n` is defined as `String` and the method invocation with an argument of type `int` is clearly wrong. On the other hand the type error in

```
def print_number(n):
  print("Number: " + n)
print_number(9)
```

shows only at runtime. Python is a weakly typed language and in general, it cannot be determined of which type a variable is allowed to be before runtime. Only when the code is executed, the `+` operator looks at its arguments and throws an error if their types don't match.

Early work used the specific properties and grammar of a programming language to develop algorithms which catch errors where ever possible [17, 26].

Modern integrated development environments (IDEs) still use such algorithms to provide error detection and correction functionality to the programmer (compare Figure 2.2).

2.2.1 Neural Network based Approaches

However, it is impossible for a traditional algorithm to find all errors in a program. Depending on the properties of the programming language it can be hard to find semantic errors and logic errors are even harder to detect because they require an understanding of the purpose of the program. Even a human struggles to detect errors in the logic of a program and that is why a model which is able to find these errors would be very useful.

There has been little research on automatic error correction in source code but the one there is mostly focuses on syntax errors and beginners mistakes. Because traditional algorithms are insufficient, the attention has recently shifted to deep neural networks. In [4] the authors train a recurrent neural network model on the task of correcting beginner's syntax errors in small programs. They train on a large corpus of student submissions for five simple programming tasks with the purpose of automatically generating useful feedback for such exercises thus removing the need for a human to do so. In [29] the authors also use a recurrent architecture to predict the exact location of a syntax error and to suggest possible corrections.

The model used in this thesis tries to go further. It is the goal to not only correct syntax errors but also semantic ones and possibly even logic ones. To reach this goal the model uses neural machine translation techniques which have already been successfully applied to the correction of text input [33].

Chapter 3

Model and Training

3.1 Components

This section aims to give a short overview of the main techniques and architectures used in this thesis. Different types of recurrent neural networks are discussed as well as the sequence-to-sequence model and the attention mechanism. All these architectures are essential parts of the model used in this thesis.

3.1.1 Types of Recurrent Neural Networks

A recurrent neural network (RNN)[32] is a special form of neural network that is used for sequential tasks. It works by having multiple copies of the network, one for each timestep. As the input proceeds in time, the network passes information to its next instance as seen in Figure 3.1. For an input sequence $(\mathbf{x}_1, ..., \mathbf{x}_n)$ a very simple RNN produces at each timestep t a hidden state vector $\mathbf{h}_t$ as follows:

$$\mathbf{h}_t = \tanh\left(\mathbf{W}\begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right)$$

However, vanilla RNNs have proven to be hard to train and to struggle with long-range dependencies [14]. In theory, they should be able to deal with these dependencies but either vanishing or exploding gradients usually prevent them from doing so. These issues were addressed with the introduction of Long Short-Term Memory networks (LSTMs) [15]. In addition to $\mathbf{h}_t$, LSTMs also pass a memory state vector $\mathbf{c}_t$ to the next instance as can be seen in Figure 3.2. The LSTM can choose at each timestep which information it wants to read or forget from the memory vector or which new one it wants to write onto the vector. This is done by using explicit gating mechanisms:

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right) \qquad \mathbf{i}_t = \sigma\left(\mathbf{W}_i\begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right)$$
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o\begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right) \qquad \mathbf{g}_t = \tanh\left(\mathbf{W}_g\begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right)$$

Here σ is the sigmoid function. $\mathbf{f}_t$, $\mathbf{i}_t$ and $\mathbf{o}_t$ can be thought of as binary gates that decide which information from $\mathbf{c}_{t-1}$ should be deleted, which information

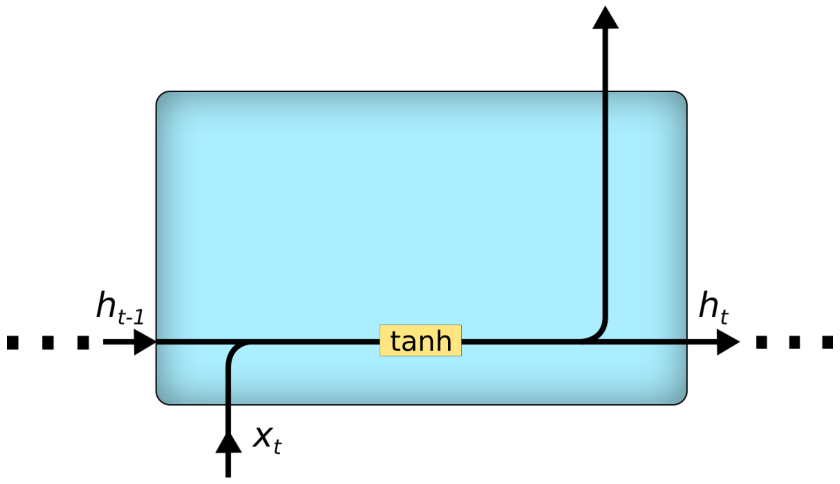


Figure 3.1: Architecture of a vanilla recurrent neural network cell. Each timestep some output and a hidden state vector $\mathbf{h}_t$ are produced by looking at the hidden state vector from the previous timestep $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$. In this simple example, $\mathbf{h}_t$ is also the output.

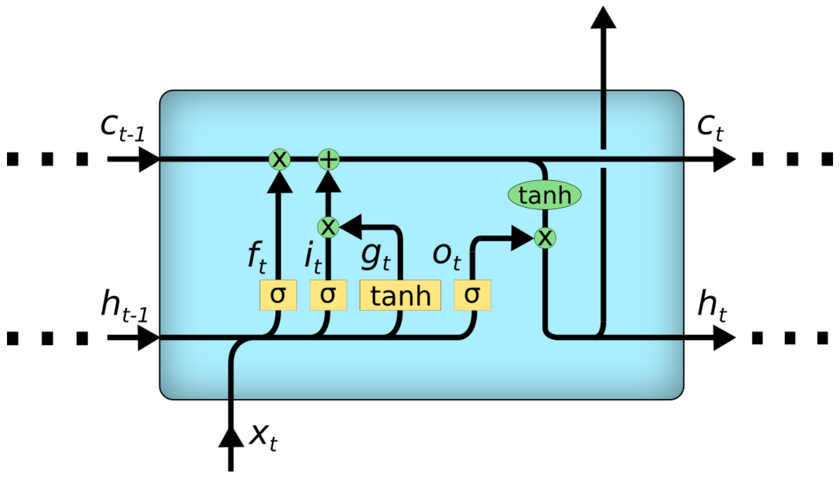


Figure 3.2: Architecture of a typical LSTM cell. In addition to the hidden state vector $\mathbf{h}_t$, a memory state vector $\mathbf{c}_t$ is passed to the next timestep. $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$ are used to compute the gates $\mathbf{f}_t$, $\mathbf{i}_t$ and $\mathbf{o}_t$ and the candidate vector $\mathbf{g}_t$. The gates are then used to add, delete and retrieve information to respectively from $\mathbf{c}_{t-1}$, subsequently generating $\mathbf{c}_t$ and $\mathbf{h}_t$.

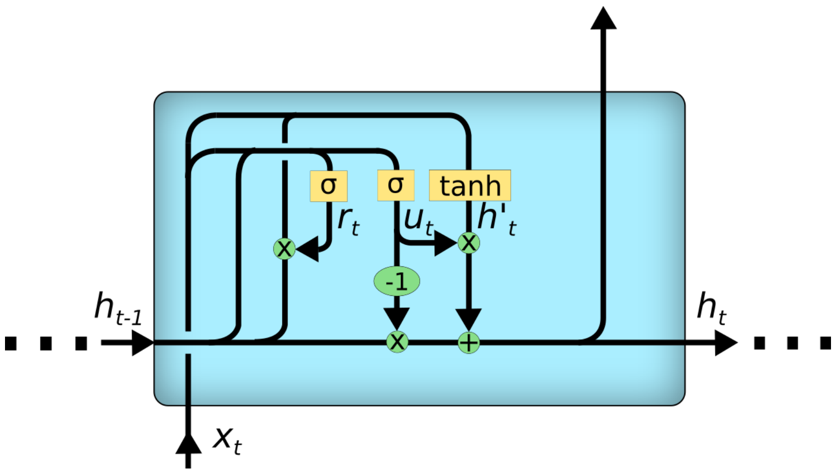


Figure 3.3: Architecture of a typical GRU cell. $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$ are used to compute the reset gate $\mathbf{r}_t$ and the update gate $\mathbf{u}_t$. These gates are then used to compute a candidate vector $\mathbf{h}'_t$ which in turn is then used to compute the new hidden state vector $\mathbf{h}_t$.

of $\mathbf{c}_{t-1}$ should be updated and which information from $\mathbf{c}_t$ should be written to $\mathbf{h}_t$. Finally, $\mathbf{g}_t$ is a vector of possible values that (gated by $\mathbf{i}_t$) can be added to $\mathbf{c}_{t-1}$. The state vectors are then updated as follows:

$$\begin{aligned}\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)\end{aligned}$$

A more recent approach to optimizing RNNs are Gated Recurrent Units [7]. They have fewer parameters than LSTMs and work without an additional memory vector and are therefore faster to train and more light-weight (see Figure 3.3). Recent research suggests that the performances of LSTM networks and of GRU networks are comparable [8].

In contrast to an LSTM, a GRU cell only has two gates, a reset gate $\mathbf{r}_t$ and an update gate $\mathbf{u}_t$. They are computed in a similar way as the gates of an LSTM:

$$\begin{aligned}\mathbf{r}_t &= \sigma\left(\mathbf{W}_r \begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right) \\ \mathbf{u}_t &= \sigma\left(\mathbf{W}_u \begin{pmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{pmatrix}\right)\end{aligned}$$

These gates are than used to calculate a candidate vector $\mathbf{h}'_t$ of possible new values which is then used to compute the updated hidden state vector $\mathbf{h}_t$.

$$\begin{aligned}\mathbf{h}'_t &= \tanh\left(\mathbf{W}_{h'} \begin{pmatrix}\mathbf{x}_t \\ \mathbf{r}_t \odot \mathbf{h}_{t-1}\end{pmatrix}\right) \\ \mathbf{h}_t &= (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \mathbf{h}'_t\end{aligned}$$

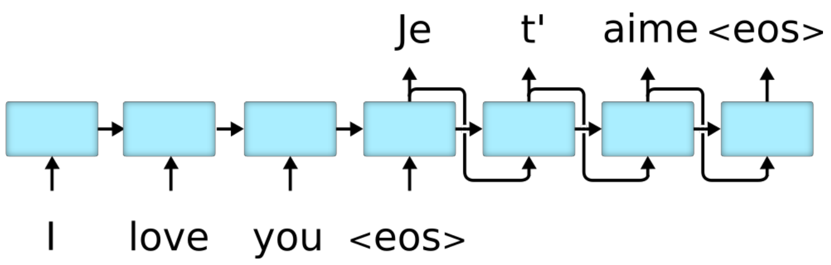


Figure 3.4: The sequence-to-sequence model applied to a translation example. The English source sentence is fed to the model word by word. After the input of an end-of-sequence token (`<eos>`), the network starts producing the output sentence in French. For this, the produced output tokens are fed back to the network at the next timestep. The network signals the end of the sequence by outputting another `<eos>` token.

Almost all state of the art results today are achieved using either LSTMs, GRUs or similar networks with some form of gating mechanism because they are easier to train than vanilla RNNs and excel at capturing long-range dependencies.

3.1.2 The Sequence-to-Sequence Model

Traditional deep neural networks process the whole input in one step and then calculate some output, e.g. process an image and then classify it. This works well for problems where the input and the output are of a fixed dimension, however, it is not suitable for problems where the input and the output are sequences of variable length. An example would be the input of a question and the network should produce an answer. We have seen that we can use LSTMs to process input sequences of variable length. However, in this case we want to process the whole input sequence and all the information that comes with it first and only then start generating an output sequence. These problems are called sequence to sequence problems.

In [30] the Sequence-to-Sequence Model is introduced as a solution to these problems. The model was applied to the task of neural machine translation (NMT) and has since become the state of the art architecture in this field. The main concept can be seen in Figure 3.4. First, the whole input sequence is fed into the network and the output is ignored. Then an end-of-sequence token `<eos>` is input which signals the network to start producing the output. From there on the produced output tokens are fed to the network until another an end-of-sequence token is generated, thus signaling the end of the sequence. To speed up training the expected output is fed back to the network and not the actually produced output.

This architecture is further improved by splitting the network into two separate LSTMs. The first network takes all the input and encodes it into its hidden state vector which is then used to initialize the second network which is first fed a start token `<GO>` and then the generated output until the end of the sequence is reached.

These networks usually operate at word level and use some word embedding like word2vec [22]. This method has the advantage of giving the input words some meaning through the embedding instead of just inputting a meaningless encoding of the word. While this is very effective for translation tasks, there are some limitations to this method. Embeddings work on a fixed size vocabulary which means that out of vocabulary words (OOV) can't be handled. Also special character sequences like `:)` pose a problem.

3.1.3 Attention-Mechanism

Attention is a relatively new concept for neural networks. The idea is to allow the network to chose on which information to focus at any given moment. For example in [23] attention is used on the task of high resolution image classification. These kinds of networks often struggle with memory constraints and attention can help them to only load the significant part of the image into the memory.

Attention has subsequently been applied to NMT [3, 21]. The vector into which the input is encoded in the sequence-to-sequence model has been identified as a bottleneck which cuts down performance because of its limited capacity. After all the vector is of fixed dimensionality and needs to encode information about the whole input sequence. Because of that attention is used as a way for the decoder to peek at previous hidden states of the encoder. This is done with a context vector $\tilde{\mathbf{c}}_t$ which is combined with the current hidden state of the decoder $\mathbf{h}_t$. The resulting attentional hidden state $\tilde{\mathbf{h}}_t$ is then used by the decoder to generate the next output.

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_c \begin{pmatrix} \tilde{\mathbf{c}}_t \\ \mathbf{h}_t \end{pmatrix}\right)$$

For the derivation of the context vector $\tilde{\mathbf{c}}_t$ all hidden states $\bar{\mathbf{h}}_s$ of the encoder are considered. For this, an alignment vector $\mathbf{a}_t$ whose size equals the input sequence length is derived from the current decoder hidden state $\mathbf{h}_t$ and the encoder hidden states $\bar{\mathbf{h}}_s$. The values of $\mathbf{a}_t$ are normalized using the softmax function.

$$a_{ts} = \frac{\exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'} \exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))}$$

Here, score is a content-based function used to compare the decoder hidden state $\mathbf{h}_t$ with each one of the encoder hidden states $\bar{\mathbf{h}}_s$. There are various possible choices for this function, for example:

$$\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^\intercal \mathbf{W}_a \bar{\mathbf{h}}_s \\ \mathbf{v}_a^\intercal \tanh\left(\mathbf{W}_a \begin{pmatrix} \mathbf{h}_t \\ \bar{\mathbf{h}}_s \end{pmatrix}\right) \end{cases}$$

The context vector $\tilde{\mathbf{c}}_t$ is then calculated as the weighted average over the encoder hidden states.

$$\tilde{\mathbf{c}}_t = \sum_{s'} a_{ts'} \bar{\mathbf{h}}_{s'}$$

3.2 Model Implementation

The implementation of the model used in this thesis is mostly based on the NMT model from Tensorflow [20]. The model consists of an encoder and a decoder with an implementation of the Luong attention mechanism [21] on top. The encoder and the decoder are both an LSTM cell consisting of 4 layers à 256 units each. No embedding was used because the model operates at the character level instead of the word level and a single character doesn't have much meaning without its context. This change was necessary because programming languages don't have a fixed vocabulary. The programmer is not restricted in the naming of variables, methods or the like and thus it makes no sense to restrict the model to a fixed vocabulary. Otherwise, it would only result in a lot of OOV tokens. Therefore the input is fed one character at a time to the model with the encoding of the character simply being its ASCII code, i.e. a number between 0 and 127.

For inference, the decoder was first fed a start-of-sequence token and after that the produced output was fed back as an input to the decoder until an end-of-sequence token was output. During training, however, the correct target sequence was fed to the decoder, left padded by a start-of-sequence token to optimize training.

During backpropagation, the gradients were clipped by a fixed norm. This technique is used to prevent the gradients from exploding [24]. The norm chosen for this thesis is 5 but other values would also be possible (1 would be another common choice).

The loss was measured with the cross-entropy loss function. Each timestep the decoder produces some output vector $\mathbf{y}'^{\intercal} = \begin{pmatrix} y'_1 & \dots & y'_n \end{pmatrix}$. This vector is then normalized using the softmax function to get probabilities p_i for each possible output.

$$p_i = \frac{\exp(y'_i)}{\sum_j \exp(y'_j)}$$

These probabilities are then used to compute the cross-entropy loss:

$$l = -\sum_j y_j * \log(p_j)$$

Here $\mathbf{y}^{\intercal} = \begin{pmatrix} y_1 & \dots & y_n \end{pmatrix}$ is a one-hot target vector with $y_i = 1$ for the desired output i and $y_j = 0$ everywhere else.

For each setting, training was done for 30,000 iterations while the batch size was set to 64. The max sequence length was set to 300 meaning that only examples of 300 characters or less were used to train and evaluate the model.

3.3 Dataset Construction

To train the model, a big dataset of erroneous code examples produced by real programmers including their respective corrections would be ideal. This would assure a large variety of errors and real-life examples. However, such a dataset does not exist in part because the correction of erroneous code is a long and tedious work which has to be done by hand.

The best alternative is to take a dataset of correct code and introduce the errors artificially. This task is no trivial one and several difficulties have to be taken into account and weighed up against each other. For one, the more sophisticated an error is the harder it is to introduce it consistently, but training a network on only easy errors (like missing semicolons) doesn't produce any added value. Another issue is the artificiality of the errors. One runs the risk of the model picking up on the error generation patterns and thus performing poorly on non-artificial examples. These problems are further discussed in Subsection 3.3.2.

For this thesis, the data from the Java Github Corpus [2] was chosen. As elaborated in Section 2.2, a weakly typed programming language like Python would be preferable over a strongly typed one like Java because a lot of errors in Java can already be found algorithmically. However, there is a general lack of large, diverse datasets of source code thus the selection of the Java Github Corpus. Furthermore, this thesis doesn't aim at building a fully polished "code corrector" but rather tries to test the boundaries of the possible. The model knows nothing of the structure and rules of the programming language and therefore the capability of the model to grasp certain concepts can still be tested.

The dataset was crawled from Github and includes only projects which were forked at least once to assure a certain measure of quality. It consists of around 15,000 projects which amount to approximately 15GB of data.

3.3.1 Preprocessing

Before the data was used, some preprocessing had to be done. While LSTMs work better than vanilla RNNs on long-range dependencies they still have their limits when it comes to input length [18]. Because the input is fed to the network character-by-character rather than word-by-word, the input sequence can get quite long rather quickly. Therefore the decision was made, to concentrate on method declarations because they are relatively self-contained and complex enough to introduce advanced errors while also being of manageable length.

The preprocessing was done for each Java file in the dataset separately and consisted of the following steps:

1. All comments were removed from the file because they are irrelevant for error detection and only increase the sequence length.

2. Line breaks were replaced by an end-of-line token.

3. All unnecessary whitespaces were removed. This was also done to reduce sequence length because Java is a whitespace insensitive programming language, i.e. a Java program is still valid (albeit harder to understand) if its indentation is removed.

4. If the file still contained non-ASCII characters, it was discarded. The purpose of this was to get rid of very rare characters, to reduce the input and output space and most importantly to avoid encoding errors.

5. All method declarations were extracted from the file and checked for their corruptibility. This means, that all corruptions (see Subsection 3.3.2) had to be able to be applied to the method. This way the rate at which the corruptions were introduced could be controlled better.

6. All suitable methods were then written to new files (ca. 100MB each), one method per line.

This resulted in around 1.7GB of train data.

3.3.2 Corruptions

As mentioned earlier, the generation of artificial errors is a challenging task. These errors need to be as sophisticated and as close to reality as possible else the learned model cannot be applied to real world examples. A large variety in the introduced errors would also be preferable. However, these guidelines are not easy to implement especially for more sophisticated errors. While it is quite easy to introduce syntax errors such as a missing semicolon, the task of automatically and unfailingly generating logic errors is very challenging. That's why this thesis concentrates on five different errors of variable difficulty.

The corruption of the data was done randomly during training. Each corruption was applied equally often, while the percentage of uncorrupted examples could be controlled. Of the possible corruptions, two produced syntax errors, two semantic errors and one logic errors. The syntax errors consisted of removing a bracket or a semicolon, for the semantic errors a variable was misspelled or the return type of the method changed and the logic errors were produced by switching the order of two statement lines. Examples and implementation details of the different corruptions can be found in Table 3.1.

Of the five possible corruptions, the syntax errors are the easiest to introduce and come relatively close to reality. They are typical errors that a novice programmer would produce and they are also the easiest ones to correct because the placement of semicolons and brackets follows strict rules. The other three corruptions produce more sophisticated errors but each one has some downsides.

The misspelling of a random variable works generally well. As a convention, the declaration of the variable is considered the "ground truth" and thus only occurrences of the variable after its declaration were misspelled. This is similar to how a traditional error checker would search for misspelled variables. However, there are some cases where the corruption doesn't work as intended. Consider the following code snippet:

```java
1 public int[] seedToArray(int seed){
2   int[] seeds = new int[1];
3   seeds[0] = seed;
4   return seeds;
5 }
```

Let's assume that `seed` is to be corrupted. One possible misspelling would be to add a random character. If perchance an 's' is added to the end of the occurrence of `seed` in the third line, it produces another valid variable, namely `seeds`. Of course, the error could still be detected but it is now a more challenging one and the model probably doesn't catch up on this unless it encounters such errors more frequently. It is also possible, that the corruption switches two adjacent characters in `seed`. If the second and third characters are selected it would have no effect and therefore no error would be added. However, these two scenarios are very rare and therefore shouldn't impact the ability of the model to learn to find misspelled variables and correct them.

The other semantic error, the changing of the return type of the method, can be generated consistently but the generation sometimes imposes unsolvable problems to the network. The first case is pretty simple. If the return type of the method is changed from `void` to something different, the model only has to check if a value is returned and if not, the return type should be changed to `void`. The second case is more complicated as the return type is changed from a different type to `void`. Again the model can determine if the return type `void` is correct by looking if a value is returned. However, if it determines that it is incorrect the model still needs to derive the correct return type from the given context which is not always possible. For example, if the following source is given:

```java
public void incrementAndGetValue(){
  this.value += 1;
  return this.value;
}
```

the correct return type is not identifiable because `this.value` is not defined in the context of this method. It could be any numeric type. There is also the problem of the return type being a superclass or an interface of the returned object. This is the trade-off of only looking at methods as opposed to whole files. However, even with the full file context, the return type could still be defined in another file, for example, if the return type is determined by the return type of a method belonging to a different class. Despite this unsolvable problem the corruption can still be used to test the limits of what the model can learn.

Lastly, logic errors are the most challenging ones to automatically generate because they require some form of understanding of the source code. To keep the errors relatively realistic, while also keeping the corruption as simple and as accurate as possible, the switching of the order of two adjacent lines in the method was chosen. However, this corruption is not guaranteed to always produce a logic error and often times produces a semantic one instead. To further increase the probability of generating an error, some restrictions were put into place. Firstly, only variable declarations, assignments or method invocations were considered and secondly, only two adjacent lines which were of a different type could be switched. This increased the probability of the occurrence of an error, but it still didn't guarantee it. Consider the following example:

```java
1 public int squareSum(int a, int b){
2   int squareA;
3   int squareB;
4   squareA = a * a;
5   squareB = b * b;
6   return squareA + squareB;
7 }
```

Here the only lines that can be switched according to the restrictions listed above are line 3 and 4 but no logic error is produced in doing so. However to produce logic errors more consistently a deeper understanding of the code would be necessary which is not possible if the errors are to be generated artificially. Having said that, the corruption with its restrictions was still deemed "good enough" to gain some insights into what the model is able to learn and what not.

Corruption	Error Type	Explanation	Example

missing bracket	syntax	One random bracket (regular, curved or squared) is selected and removed from the source.	`public int add(int a, int b{`
			`  int sum;`
			`  sum = a + b;`
			`  return sum;`
			`}`

missing semicolon	syntax	One random semicolon is selected and removed from the source.	`public int add(int a, int b){`
			`  int sum;`
			`  sum = a + b`
			`  return sum;`
			`}`

misspelled variable	semantic	A random variable which is being declared in the source is selected. A random occurrence of the variable (except the one in the declaration) it then misspelled. Possible misspellings are the removal of a random character, the insertion of a random character or the switching of two adjacent characters.	`public int add(int a, int b){`
			`  int sum;`
			`  sum = a + b;`
			`  return summ;`
			`}`

incorrect return type	semantic	The return type of the method is changed. There are only two possibilities. If the return type is `void`, it is changed to one of Java's primitive data types (`int`, `float`, etc.). In any other case, the return type is changed to `void`.	`public void add(int a, int b){`
			`  int sum;`
			`  sum = a + b;`
			`  return sum;`
			`}`

switched lines	semantic/ logic	Two adjacent statement lines are switched in their order. As statement lines qualify variable declarations, assignments and method invocations. To increase the probability of an error, only lines of different types can be switched.	`public int add(int a, int b){`
			`  sum = a + b;`
			`  int sum;`
			`  return sum;`
			`}`

Table 3.1: Implementation details and examples of the used corruptions.

Chapter 4

Experiments and Error Analysis

To get good results some experiments were conducted first to find the best possible architecture. After that, the obtained results were analyzed on in regard of the different corruption types.

4.1 Experiments

4.1.1 Corruption Rate

Because the source sequence and target sequence are almost the same and the errors are self-introduced, it is an interesting question what the optimal corruption rate for the input is. To test this the model was trained with four different corruption rates, 100%, 75%, 50% and 25%. The results can be seen in Table 4.1a.

For almost all corruptions the model trained with a corruption rate of 75% posted the best result. The models with lower percentages didn't pick up on the errors as well while the model with the 100% corruption rate didn't get as good an understanding of when the code is correct. In general, the model learned to correct all of the introduced errors. It performed especially well when inputting a sequence missing a single semicolon which is of course also the simplest task to solve. However, the model was also able to correct the other errors reasonably well. A missing bracket or an incorrect return type were corrected most of the time while the model had a little more trouble correcting a misspelled variable or realigning switched lines. A more detailed analysis of the different error types is given in Section 4.2.

What is also interesting to see is how well the model performed on uncorrupted sequences. Even the model with 100% corruption rate, i.e. which never got an uncorrupted sequence as input, managed to not introduce any new errors into an uncorrupted sequence most of the time. This indicates that the model obtains some understanding of the input and how code works and only corrects where necessary.

Going forward all experiments were done with a corruption rate of 75% and the error analysis was also done on the results of this model.

	NC	MS	MB	VAR	RET	SL
100%	60.0	64.0	48.2	**37.3**	54.3	26.7
75%	**70.4**	**71.8**	**51.3**	34.4	**55.1**	**42.6**
50%	62.0	61.2	43.7	25.0	47.4	26.1
25%	67.9	65.6	41.6	19.6	50.3	33.3

(a) Performance of models with different corruption rates.

	NC	MS	MB	VAR	RET	SL
with attention	**70.4**	**71.8**	**51.3**	**34.4**	**55.1**	**42.6**
without attention	0.0	0.0	0.0	0.0	0.0	0.0
without attention, input reversed	0.0	0.0	0.0	0.0	0.0	0.0

(b) Performance of models with or without an attention mechanism.

	NC	MS	MB	VAR	RET	SL
LSTM	**70.4**	**71.8**	**51.3**	**34.4**	**55.1**	**42.6**
GRU	0.2	0.2	0.2	0.2	0.2	0.2
vanilla RNN	0.0	0.0	0.0	0.0	0.0	0.0

(c) Performance of different RNN types.

Table 4.1: Evaluation of the performance of the models on the test set with different corruptions: uncorrupted (NC), missing semicolon (MS), missing bracket (MB), misspelled variable (VAR), wrong return type (RET) and line switch (SL).

4.1.2 Attention Mechanism

Another interesting question is if the attention mechanism is essential for the model. After all, it increases complexity and training duration. To test this, the model was twice trained without the attention mechanism on top of the decoder, once with the same input the regular model got and once with the input reversed. The reversion of the input is a technique proposed in [30], the idea being to introduce more short-term dependencies while the average distance of the dependencies stays the same. This is not necessary for the regular model because the attention mechanism allows the model to take a peek of the encoder state many timesteps ago.

The experiment revealed the attention mechanism to be an essential part of the model because the models without the mechanism weren't able to solve the given task at all (see Table 4.1b). These models never learned to repeat the input sequence probably because they couldn't pass all information from the encoder to the decoder in a single vector.

For the first couple of thousand iterations, all models learned roughly the same things, namely the general structure of the desired output. The models would start to begin the output sequence with `public ...(...){` and end it with `}<eos>`. In between, they added mostly snippets that look like code but don't make any sense. However, after about 4,000 iterations the model with the attention mechanism learned to utilize the mechanism to its full potential and started repeating the input sequence. This resulted in rapid performance improvement. The training loss of the different configurations as a function over iterations can be seen in Figure 4.1.

The reversion of the input sequence helped the model to learn a little bit faster but overall it made almost no difference. The model was still not able to solve the task.

4.1.3 RNN type

In Subsection 3.1.1 three types of RNNs were listed: vanilla RNNs, LSTMs and GRUs. To test which type works best for the given task, a model was trained for each RNN type. The training loss of the models as a function over iterations can be seen in Figure 4.2. The evaluation on the test set can be found in Table 4.1c.

As could be expected the vanilla RNN performed the worst. As explained earlier, these networks have trouble with long-range dependencies and struggle with the problem of vanishing gradients. The vanilla RNN is also the simplest type and thus the one with the least amount of trainable parameters.

More surprising was the performance gap between the GRU and the LSTM, because recent research suggests that these two network types have a comparable performance [8]. However similar to the models without the attention mechanism, the GRU network never fully learns to repeat the input sequence. One possible explanation for this is the fewer parameters of the GRU. An LSTM computes three gates at each timestep while also passing a memory vector to the next timestep. A GRU only has two gates and doesn't have a second vector in addition to the hidden state vector. Because the 256 units per layer are relatively few the lack of trainable parameters could prevent the GRU from learning as well as the LSTM.

4.2 Error analysis

In this section, the performance of the model on uncorrupted input and on each of the five corruptions is analyzed. For examples from the test set see appendix B.

4.2.1 Uncorrupted

The model works reasonably well on uncorrupted input with a 70.4% success rate. However, it is still of interest to see, what kind of errors are introduced by the network, i.e. where and why it gets confused. One mistake the model makes repeatedly is a "one-off error". Here the model would output a wrong character whose ASCII encoding is just by one off of the encoding of the correct character. For example, sometimes an asterisk whose ASCII encoding is 42 is output while the correct character would be a plus (ASCII encoding 43). If the results of the model are evaluated with a tolerance for these errors (i.e. if a character is just by one off it is not seen as an error), the accuracy of the model increases to 79.6%. That's an almost 10% performance increase and the model should be able to learn to avoid these mistakes with more training.

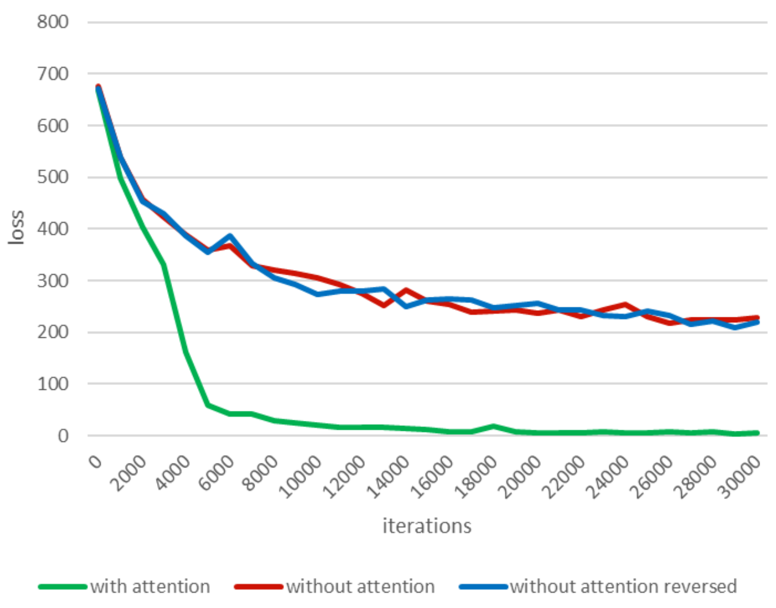


Figure 4.1: The training loss of models with or without an attention mechanism as a function over iterations.

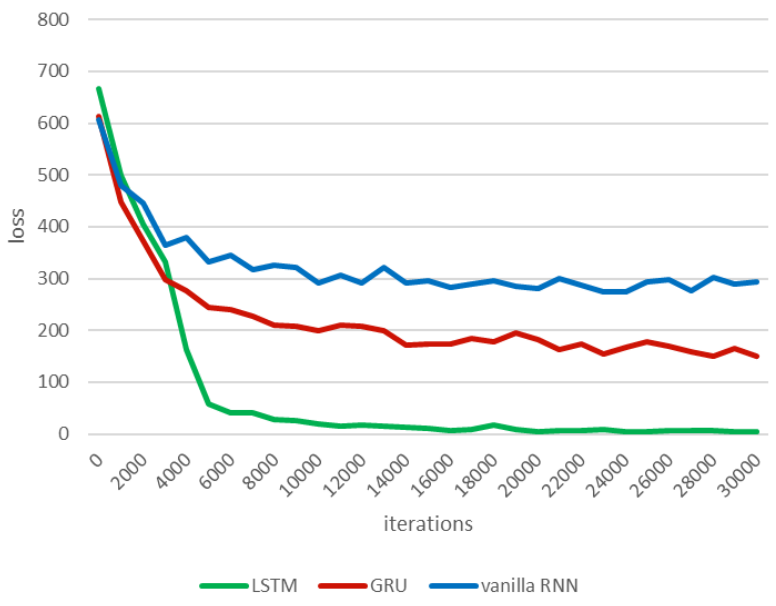


Figure 4.2: The training loss of different recurrent network types as a function over iterations.

Another common error is the random switching of lines. An additional 4.8% of the test set would have been correct if it wasn't for an incorrect line switch. This suggests that the model didn't learn to correct this corruption that well. This problem is elaborated further in Subsection 4.2.6.

The last thing that was noticeable in the test set was that sometimes the model would get stuck in some kind of loop and output some parts or lines of the input sequence multiple times. This is something that can often be observed in the early stages of training which suggests that the model should be able to avoid these mistakes with more training.

4.2.2 Missing Semicolon

The same mistakes that were observed on uncorrupted input of course also apply to the correction of corrupted input. The correction of a missing semicolon should be the easiest error to correct and while the accuracy is already good with 71.8% it is also worth to look at the results of an evaluation with the same "one-off tolerance" as in the evaluation of the uncorrupted results in which case the model is 81.7% accurate. In addition to that 3.2% of the time, an error was introduced by an incorrect line switch.

What's also interesting to see is that the output contains the correct number of semicolons in 97.0% of the time. This means that the absence of a semicolon is detected and corrected nearly every time, there are just new mistakes that are introduced by the model into the output.

4.2.3 Missing Brackets

The main advantage of LSTMs is their ability to remember long-range dependencies which should be very useful for detecting and correcting missing brackets but the test accuracy of 51.3% seems to contradict this assumption. However, a closer look at the results reveals that the task of inserting a missing bracket isn't that trivial. Consider the following example:

```java
public int addint a, int b){
  int sum = a + b;
  return sum;
}
```

Here the opening bracket between the method name and the parameters was removed. The task for the model is now to not only detect that a bracket is missing but also to find the correct spot to reinsert it which is even more difficult when there is no white space indicating the location of the missing bracket.

To see how many times the missing bracket was detected the test results were evaluated to look if the brackets in the output were balanced, meaning if every opening bracket had a closing counterpart and vice versa. The evaluation showed that the model managed to balance the brackets in 77.2% of the time. In addition to that, the brackets were also correctly nested 76.9% of the time which indicates that the model did learn the close open brackets very well.

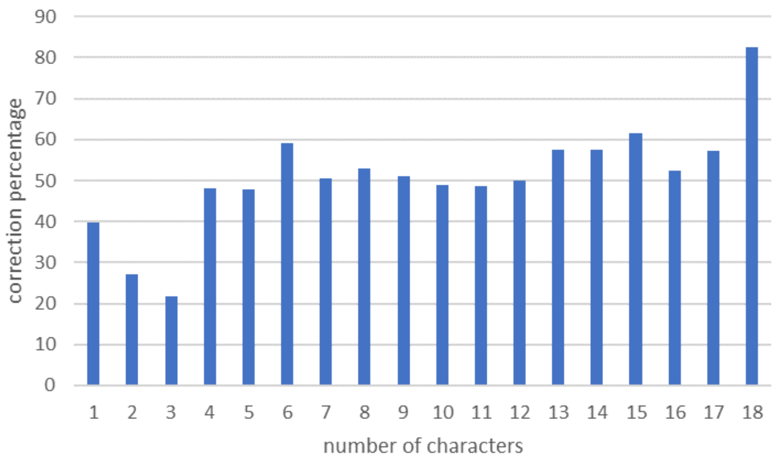


Figure 4.3: Correction percentages for variables of different lengths. Only lengths with more than 10 examples were considered.

4.2.4 Misspelled Variable

The correction of misspelled variables was the most difficult corruption to correct for the model with an overall accuracy of 34.4%. However when the test results are evaluated to see if the output contained the correct number of occurrences of the variable either all of them correctly spelled or all of them misspelled, the accuracy increases to 45.6%. This includes examples where the variable was corrected but some other error was introduced to the sequence and examples where the model corrected the wrong occurrence of the variable, i.e. it corrected the variable in the declaration to conform to the introduced misspelling later in the sequence. The latter of course also result in correct code because the variable is named consistently but it is not how the model is intended to correct variables.

The accuracy of the model was also evaluated in regard to the length of the variable name. These results can be found in Figure 4.3. As could be expected the model struggles with variables of shorter length. This suggests that the network looks for similar patterns in the sequence to determine if a variable is misspelled and then corrects it accordingly, which is, of course, easier to do if the variable name is longer. The only big exception are one-character variables but this most likely stems from the fact that these variables have a simpler corruption pattern because there are no two characters that can be switched and the deletion of a random character doesn't serve as well so the misspelling is always an additional character that has to be removed.

4.2.5 Wrong Return Type

As discussed in Subsection 3.3.2 it is not always possible to derive the correct return type from a method. Sometimes there is no indication of the correct type or the return type is specified as a superclass or interface of the object that is actually returned. In regard of this, the 55.1% accuracy scored by the model

Return Type	Accuracy %
byte(1)	100.0
State(15)	86.7
int(447)	70.5
void(5022)	68.2
String(321)	60.8
IFigure(284)	44.7
Collection(24)	33.3
boolean(168)	14.3
Double(82)	0.0
Session(1)	0.0

Table 4.2: Some return types and their correction percentages. The number in brackets indicates the number of occurrences of this type in the test set.

$\leftrightarrow$	VD	MI	AS
VD	-	53.3	**59.3**
MI	0.0	-	13.0
AS	0.0	9.8	-

Table 4.3: The correction percentages for different line switches. The rows indicate the type of the first line and the columns the type of the second line. There are three possible types: variable declaration (VC), method invocation (MI) or assignment (AS). Only lines of different types were switched.

is pretty good. To analyse this result further the model was evaluated to see which types it was able to correct and with which it struggled. A selection of these results can be found in Table 4.2.

The accuracy for different data types varies greatly. There are some data types (alas not many) that the model never manages to correct. This can have different causes for example as mentioned it could be impossible to derive the correct return type or in the case of `Double` the model can also be confused by the distinction of the primitive data type `double` and its object wrapper `Double`. In the case of a `boolean` return type, there is also the additional difficulty that often no variable of type `boolean` is returned but rather a statement that resolves to a `boolean` value, e.g. `return i == 1;`.

Even though the model generally scores better on examples that occur more often, it can also be seen that this is no necessity for the model to derive the correct return type. For example, `byte` only occurs once in the test set and probably not much more in the training data and the model still managed to derive the correct return type for the method.

4.2.6 Line Switch

At first sight, the performance of the model on the switched lines corruption appears to be pretty good with 42.6%. However, observations from the previous subsections suggest that the model still gets confused with some line switches. For this purpose, the test results were evaluated with regard to the type of the switched lines. The scores can be found in Table 4.3.

It quickly becomes apparent that the model only really learns to correct line switches where the first line is a variable declaration. These switches mostly produce semantic errors rather than logic ones because often the declared variable is used in the other line and therefore before it was declared after the switch. For example the error in:

```java
public int add(int a, int b){
  sum = a + b;
  int sum;
  return sum;
}
```

is a semantic one and not a logic one. Furthermore by going through the test results, one can see that the other switches often don't produce any error at all (examples can be found in Appendix B). Unfortunately, it is not possible to evaluate the percentage of introduced errors quantitatively because there would be no need to train a model on the task of logic error detection if it could already be done reliably.

Nevertheless, it is reasonable to draw the conclusion that the model gets confused by the line switches with low correction accuracy because it cannot figure out when and how they occur. This is probably the reason for the random line switches discussed in previous subsections. These low accuracy switches are also the ones that are not very likely to produce an error.

Chapter 5

Conclusion

In this thesis, a sequence-to-sequence model with an attention mechanism was trained on the task of automatic error correction in source code. Experiments on the architecture of the model were conducted revealing the attention mechanism to be an essential part of the model which can not be omitted. Models without the attention mechanism weren't even able to solve the given task. The experiments also showed that an LSTM network outperforms both a vanilla RNN network and a GRU network on this task most likely because it has more trainable parameters.

The errors on which the model was trained were self-introduced into a dataset of correct Java code and ranged from simple syntax errors to more sophisticated semantic ones. The problem of automatically generating logic errors was discussed and it was attempted to introduce such errors into the dataset by switching the order of two adjacent lines in the code. The model managed to achieve promising results on all of the corruptions and further analysis of the errors showed that the performance of the model could even be further improved with additional training. The analysis also showed that the model achieved some understanding of the semantics of source code and how code works.

Alas, this thesis was not able to answer the question if a neural network can detect and correct logic errors. The switching of lines proved to be an insufficient corruption and didn't manage to consistently generate logic errors. On the contrary, most of the errors introduced were semantic ones which the model learned to correct successfully. However, this corruption also confused the model because sometimes these line switches would not introduce any error and because of that the model was unable to learn how to correct these inputs.

5.1 Future Work

The model in this thesis has proven to be able to correct syntax and semantic errors. As mentioned it still remains to be seen if it can also detect and correct logic errors. To test this a different corruption has to be found which introduces logic errors more consistently into the dataset. An even better alternative would be the collection of a dataset of real-life examples. With that, the model could be evaluated on its performance on a big variety of non-artificial errors and one would most likely get a deeper insight into the possibilities and limitations of this model.

For this thesis, the model was trained on Java which is a strongly typed language. Future work could also include applying the same techniques to a weakly typed language. Errors in weakly typed languages are harder to spot which could bring some additional insights. This could also have some real-life applications because IDEs today are still not able to spot most errors in weakly typed languages.

Appendix A

Notation

A.1 Naming Conventions

Vector and matrix variables are always written in bold, vectors having lowercase names, matrices uppercase ones. For example:

$$\mathbf{a}_t, \mathbf{B}$$

If no other indication of the nature of the variable is given, it is a vector respectively a matrix of learnt parameters. These variables are often named $\mathbf{W}$ for matrices and $\mathbf{v}$ for vectors.

The individual elements of a vector are not written in bold and are indexed with a subscript.

$$\mathbf{a} = \begin{pmatrix} a_1 \\ ... \\ a_n \end{pmatrix}$$

If the vector already has a subscript, the index of the element is added as an additional subscript.

$$\mathbf{a}_i = \begin{pmatrix} a_{i1} \\ ... \\ a_{in} \end{pmatrix}$$

A.2 Vector Operations

For $\mathbf{a}^\intercal = \begin{pmatrix} a_1 & ... & a_n \end{pmatrix}$ and $\mathbf{b}^\intercal = \begin{pmatrix} b_1 & ... & b_n \end{pmatrix}$, $\odot$ depicts the elementwise multiplication of two vectors with the same dimensionality.

$$\mathbf{a} \odot \mathbf{b} = \begin{pmatrix} a_1 * b_1 \\ ... \\ a_n * b_n \end{pmatrix}$$

The concatenation of vectors is abbreviated as follows:

$$\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix} = \begin{pmatrix} a_1 \\ ... \\ a_n \\ b_1 \\ ... \\ b_n \end{pmatrix}$$

All functions such as tanh or σ are applied to a vector elementwise unless specified differently.

$$\tanh \begin{pmatrix} a_1 \\ ... \\ a_n \end{pmatrix} = \begin{pmatrix} \tanh(a_1) \\ ... \\ \tanh(a_n) \end{pmatrix}$$

Appendix B

Showcase

On the next few pages some examples of the test set with different corruptions are given. For each corruption there is a table with the input and the generated output together with a short explanation.

Input	Output

`@Override`
`public int hashCode() {`
` int hash = getClass().hashCode();`
` hash += hash * 31 + JodaBeanUtils.hashCode(getPaging());`
` hash += hash * 31 +`
`     JodaBeanUtils.hashCode(getMarketDatas());`
` return hash;`
`}`
	`@Override`
	`public int hashCode() {`
	` int hash = getClass().hashCode();`
	` hash += hash * 31 + JodaBeanUtils.hashCode(getPaging());`
	` hash += hash * 31 + JodaBeanUtils.hashCode(getMaging());`
	` return hash;`
	`}`

`@Test(expectedExceptions = DataNotFoundException.class)`
`public void`
`    test_removeHistoricalTimeSeries_versioned_notFoundId()`
`    {`
` UniqueId uniqueId = UniqueId.of("DbHts", "0", "0");`
` _htsMaster.remove(uniqueId);`
`}`
	`@Test(expectedExceptions = DataNotFoundException.class)`
	`public void`
	`    test_removeHistoricalTimeSeries_versioned_notFoundId()`
	`    {`
	` UniqueId uniqueId = UniqueId.of("DbHts", "0", "0");`
	` _itsMaster.remove(uniqueId);`
	`}`

`public void addCount(int sourceToken, int targetToken,`
`    int count) {`
` if (targetToken < sourceToken) {`
`  int swap = sourceToken;`
`  sourceToken = targetToken;`
`  targetToken = swap;`
` }`
` edits[sourceToken][targetToken] += count;`
`}`
	`public void addCount(int sourceToken, _ int count) {`
	` if (targetToken < sourceToken) {`
	`  int swap = sourceToken;`
	`  sourceToken = targetToken;`
	`  targetToken = swap;`
	` }`
	` edits[sourceToken][targetToken] += count;`
	`}`
	` edits[sourceToken][targetToken] += count;`
	`}`

Table B.1: In this table there are some common mistakes the model sometimes introduces in its output. In the first example the model mixes two lines up because they are very similar. In the second example it manages to reproduce the input sequence correctly but is one-off with one single character. The last example shows how the model sometimes gets stuck in some sort of loop and outputs a part of the input multiple times.

Input	Output

`@Override`
`public Method run() {`
` try {`
`  final Method mtd = clazz.getMethod("writeReplace")`
`  mtd.setAccessible(true);`
`  return mtd;`
` } catch (NoSuchMethodException e) {}`
` return null;`
`}`
	`@Override`
	`public Method run() {`
	` try {`
	`  final Method mtd = clazz.getMethod("writeReplace");`
	`  mtd.setAccessible(true);`
	`  return mtd;`
	` } catch (NoSuchMethodException e) {}`
	` return null;`
	`}`

`public void test_hashCode() {`
` ExternalIdWithDates d1a =`
`     ExternalIdWithDates.of(IDENTIFIER, VALID_FROM,`
`     VALID_TO);`
` ExternalIdWithDates d1b =`
`     ExternalIdWithDates.of(IDENTIFIER, VALID_FROM,`
`     VALID_TO);`
` assertEquals(d1a.hashCode(), d1b.hashCode())`
`}`
	`public void test_hashCode() {`
	` ExternalIdWithDates d1a =`
	`     ExternalIdWithDates.of(IDENTIFIER, _ VALID_TO);`
	` ExternalIdWithDates d1b =`
	`     ExternalIdWithDates.of(IDENTIFIER, _ VALID_TO);`
	` assertEquals(d1a.hashCode(), d1b.hashCode());`
	` assertEquals(d1a.hashCode(), d1b.hashCode());`
	`}`

`public static Message createByeRequest(Dialog dialog) {`
` Message msg = createRequest(dialog, SipMethods.BYE, null)`
` msg.removeExpiresHeader();`
` msg.removeContacts();`
` return msg;`
`}`
	`public static Message createByeRequest(Dialog dialog) {`
	` Message msg = cialog dialog, SipMethods.BYE, null);`
	` msg.removeExpiresHeader();`
	` msg.removeContacts();`
	` return msg;`
	`}`

Table B.2: These examples show that the insertion of a missing semicolon is no big problem for the model. If the output is incorrect it is only because new errors are introduced by the model similar to the ones seen in Table B.1.

Input	Output

`int pop(int numBits)`
` int i = getLeadingAsInt(numBits);`
` truncate(numBits);`
` return i;`
`}`
	`int pop(int numBits) {`
	` int i = getLeadingAsInt(numBits);`
	` truncate(numBits);`
	` return i;`
	`}`

`protected void outlineShape(Graphics graphics, Rectangle`
`     bounds) {`
` PointList pl = setupPoints_bounds);`
` graphics.drawPolygon(pl);`
` int add = graphics.getLineWidth() / 2;`
` graphics.drawOval(new Rectangle(ovalX, ovalY, ovalD +`
`      add, ovalD + add));`
`}`
	`protected void outlineShape(Graphics graphics, Rectangle`
	`     bounds) {`
	` PointList pl = setupPointsbounds();`
	` graphics.drawPolygon(pl);`
	` int add = graphics.getLineWidth() / 2;`
	` graphics.drawOval(new Rectangle(ovalX, ovalY, ovalD +`
	`      add, ovalD + add));`
	`}`

`public void partActivated_IWorkbenchPart part) {`
` workbenchPart = part;`
` refreshActions();`
`}`
	`public void partActivatedIWorkbenchPart(part) {`
	` workbenchPart = part;`
	` refreshActions();`
	`}`

Table B.3: These examples show well what the model is able to do when reinserting a missing bracket. In all three examples the missing bracket is detected and reinserted but only in one example in the right place. The second and the third example are very difficult for the model because there is no whitespace indicating the correct location of the bracket.

Input	Output
`private boolean validateOrder(InteractionOperand interactionOperand) { orderedFragments = interactionOpernd.getFragments(); computeConstraints(); return reorderFragmentsInAValidTrace(); }`	`private boolean validateOrder(InteractionOperand interactionOperand) { orderedFragments = interactionOperand.getFragments(); computeConstraints(); return reorderFragmentsInAValidTrace(); }`
`@Override public void mouseReleased(MouseEvent e) { popup.setVisible(false); String colorText = "RGB = " + buttonColor.getRed() + ", " + buttonColor.getBreen() + ", " + buttonColor.getBlue(); this.setText(colrText); this.firePropertyChange(COLOR_CHANGE, previousColor, buttonColor); }`	`@Override public void mouseReleased(MouseEvent e) { popup.setVisible(false); String colorText = "RGB = " + buttonColor.getRed() + ", " + buttonColor.getBreen() + ", " + buttonColor.getBlue(); this.setText(colrText); this.firePropertyChange(COLOR_CHANGE, previousColor, buttonColor); }`
`public void testGetCanonicalGraphSet() { String canon1 = SWPSignatureUtilities.getCanonicalGraphSet(this.set); String canon2 = SWPSignatureUtilities.getCanonicalGraphSet(this.set); assertEquals(cano1n, canon2); }`	`public void testGetCanonicalGraphSet() { String canon1 = SWPSignatureUtilities.getCanonicalGraphSet(this.set); String canon2 = SWPSignatureUtilities.getCanonicalGraphSet(this.set); assertEquals(canon2., canon2); }`

Table B.4: As can be seen in this examples, a misspelled variable is not always detected by the model. And if it is detected it can sometimes be difficult for the model to derive the correct ground truth as seen in the last example.

Input	Output

`@Override`
`public void toString() {`
` if (eIsProxy()) return super.toString();`
` StringBuffer result = new StringBuffer(super.toString());`
` result.append(" (name: ");`
` result.append(name);`
` result.append(')');`
` return result.toString();`
`}`
	`@Override`
	`public String toString() {`
	` if (eIsProxy()) return super.toString();`
	` StringBuffer result = new StringBuffer(super.toString());`
	` result.append(" (name: ");`
	` result.append(name);`
	` result.append(')');`
	` return result.toString();`
	`}`

`@Override`
`public void evaluate(final Double...ts) {`
` Validate.isTrue(ts.length == 2);`
` final double tau = ts[0];`
` final double s = ts[1];`
` final double t = maturity - tau;`
` final double temp = vol * Math.pow(s, beta) *`
`     localVol.getVolatility(t, s);`
` return -0.5 * temp * temp;`
`}`
	`@Override`
	`public Validate evaluate(final Double...ts) {`
	` >final double tau = ts[0];`
	` >Validate.isTrue(ts.length == 2);`
	` final double s = ts[1];`
	` final double t = maturity - tau;`
	` final double temp = vol * Math.pow(s, beta) *`
	`     localVol.getVolatility(t, s);`
	` return -0.5 * temp * temp;`
	`}`

`public void createFigure() {`
` ResizableCompartmentFigure result =`
`     (ResizableCompartmentFigure) super.createFigure();`
` result.setTitleVisibility(false);`
` return result;`
`}`
	`public Resizable createFigure() {`
	` ResizableCompartmentFigure result =`
	`     (ResizableCompartmentFigure) super.createFigure();`
	` result.setTitleVisibility(false);`
	` return result;`
	`}`

Table B.5: The examples show that the model is able to derive the return type of the method from the given context it just doesn't always chose the right type. Furthermore in the second example, there is also a random line switch to be seen.

Input	Output

`protected void disposeElementInfo(Object element,`
`    ElementInfo info) {`
` if (info instanceof ResourceSetInfo) {`
`  >resourceSetInfo.dispose();`
`  >ResourceSetInfo resourceSetInfo = (ResourceSetInfo)`
`      info;`
` }`
` super.disposeElementInfo(element, info);`
`}`
	`protected void disposeElementInfo(Object element,`
	`    ElementInfo info) {`
	` if (info instanceof ResourceSetInfo) {`
	`  >ResourceSetInfo resourceSetInfo = (ResourceSetInfo)`
	`      info;`
	`  >resourceSetInfo.dispose();`
	` }`
	` super.disposeElementInfo(element, info);`
	`}`

`private void resolveEntry(Entry < K, T > entry) {`
` >entry.isResolved = true;`
` >resolved.add(entry);`
` resolved(entry);`
`}`
	`private void resolveEntry(Entry < K, T > entry) {`
	` >entry.isResolved = true;`
	` >resolved.add(entry);`
	` resolved(entry);`
	`}`

`protected NodeFigure createMainFigure() {`
` NodeFigure figure = createNodePlate();`
` figure.setLayoutManager(new StackLayout());`
` >figure.add(shape);`
` >IFigure shape = createNodeShape();`
` contentPane = setupContentPane(shape);`
` return figure;`
`}`
	`protected NodeFigure createMainFigure() {`
	` NodeFigure figure = createNodePlate();`
	` figure.setLayoutManager(new StackLayout());`
	` >figure.add(shape);`
	` >IFigure shape = createNodeShape();`
	` contentPane = setupContentPane(shape);`
	` return figure;`
	`}`

Table B.6: As can be seen in the first example, the model is able to detect a line switch if it results in a variable being used before it is declared. However, this doesn't mean that it is always detected (see the last example). The second example shows, that a line switch doesn't always result in an error which makes it impossible for the model to detect.

List of Tables

List of Figures

Bibliography

[1] Aiman, U. ; Vishwakarma, V. P.: Face recognition using modified deep learning neural network. In: *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2017, S. 1–5

[2] Allamanis, Miltiadis ; Sutton, Charles: Mining Source Code Repositories at Massive Scale using Language Modeling. In: *The 10th Working Conference on Mining Software Repositories* IEEE, 2013, S. 207–216

[3] Bahdanau, Dzmitry ; Cho, Kyunghyun ; Bengio, Yoshua: Neural Machine Translation by Jointly Learning to Align and Translate. In: *CoRR* abs/1409.0473 (2014). http://arxiv.org/abs/1409.0473

[4] Bhatia, Sahil ; Singh, Rishabh: Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. In: *CoRR* abs/1603.06129 (2016). http://arxiv.org/abs/1603.06129

[5] Carlson, Andrew ; Fette, Ian: Memory-Based Context-Sensitive Spelling Correction at Web Scale. In: *Proceedings of the Sixth International Conference on Machine Learning and Applications*. Washington, DC, USA : IEEE Computer Society, 2007 (ICMLA '07). – ISBN 0–7695–3069–9, 166–171

[6] Carlson, Gary: Techniques for Replacing Characters That Are Garbled on Input. In: *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1966 (AFIPS '66 (Spring)), 189–193

[7] Cho, Kyunghyun ; Merrienboer, Bart van ; Gülçehre, Çaglar ; Bougares, Fethi ; Schwenk, Holger ; Bengio, Yoshua: Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In: *CoRR* abs/1406.1078 (2014). http://arxiv.org/abs/1406.1078

[8] Chung, Junyoung ; Gülçehre, Çaglar ; Cho, KyungHyun ; Bengio, Yoshua: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. In: *CoRR* abs/1412.3555 (2014). http://arxiv.org/abs/1412.3555

[9] Cornew, Ronald W.: A statistical method of spelling correction. In: *Information and Control* 12 (1968), Nr. 2, 79 - 93. http://dx.

doi.org/https://doi.org/10.1016/S0019-9958(68)90201-5. – DOI https://doi.org/10.1016/S0019–9958(68)90201–5. – ISSN 0019–9958

[10] Damerau, Fred J.: A Technique for Computer Detection and Correction of Spelling Errors. In: *Commun. ACM* 7 (1964), März, Nr. 3, 171–176. http://dx.doi.org/10.1145/363958.363994. – DOI 10.1145/363958.363994. – ISSN 0001–0782

[11] Ghosh, Shaona ; Kristensson, Per O.: Neural Networks for Text Correction and Completion in Keyboard Decoding. In: *CoRR* abs/1709.06429 (2017). http://arxiv.org/abs/1709.06429

[12] Golding, Andrew R. ; Roth, Dan: A Winnow-Based Approach to Context-Sensitive Spelling Correction. In: *Mach. Learn.* 34 (1999), Februar, Nr. 1-3, 107–130. http://dx.doi.org/10.1023/A:1007545901558. – DOI 10.1023/A:1007545901558. – ISSN 0885–6125

[13] Han, Na-Rae ; Chodorow, Martin ; Leacock, Claudia: Detecting Errors in English Article Usage by Non-native Speakers. In: *Nat. Lang. Eng.* 12 (2006), Juni, Nr. 2, 115–129. http://dx.doi.org/10.1017/S1351324906004190. – DOI 10.1017/S1351324906004190. – ISSN 1351–3249

[14] Hochreiter, Sepp: Untersuchungen zu dynamischen neuronalen Netzen. (1991), 04

[15] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-Term Memory. In: *Neural Comput.* 9 (1997), November, Nr. 8, 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735. – DOI 10.1162/neco.1997.9.8.1735. – ISSN 0899–7667

[16] Hu, Ronghang ; Dollár, Piotr ; He, Kaiming ; Darrell, Trevor ; Girshick, Ross B.: Learning to Segment Every Thing. In: *CoRR* abs/1711.10370 (2017). http://arxiv.org/abs/1711.10370

[17] Lévy, J. P.: Automatic correction of syntax-errors in programming languages. In: *Acta Informatica* 4 (1975), Sep, Nr. 3, 271–292. http://dx.doi.org/10.1007/BF00288730. – DOI 10.1007/BF00288730. – ISSN 1432–0525

[18] Li, Shuai ; Li, Wanqing ; Cook, Chris ; Zhu, Ce ; Gao, Yanbo: Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN. In: *CoRR* abs/1803.04831 (2018). http://arxiv.org/abs/1803.04831

[19] Liskov, Barbara ; Zilles, Stephen: Programming with Abstract Data Types. In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages.* New York, NY, USA : ACM, 1974, 50–59

[20] Luong, Minh-Thang ; Brevdo, Eugene ; Zhao, Rui: Neural Machine Translation (seq2seq) Tutorial. In: *https://github.com/tensorflow/nmt* (2017)

[21] Luong, Minh-Thang ; Pham, Hieu ; Manning, Christopher D.: Effective Approaches to Attention-based Neural Machine Translation. In: *CoRR* abs/1508.04025 (2015). http://arxiv.org/abs/1508.04025

[22] Mikolov, Tomas ; Chen, Kai ; Corrado, Greg ; Dean, Jeffrey: Efficient Estimation of Word Representations in Vector Space. In: *CoRR* abs/1301.3781 (2013). http://arxiv.org/abs/1301.3781

[23] Mnih, Volodymyr ; Heess, Nicolas ; Graves, Alex ; Kavukcuoglu, Koray: Recurrent Models of Visual Attention. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. Cambridge, MA, USA : MIT Press, 2014 (NIPS'14), 2204–2212

[24] Pascanu, Razvan ; Mikolov, Tomas ; Bengio, Yoshua: Understanding the exploding gradient problem. In: *CoRR* abs/1211.5063 (2012). http://arxiv.org/abs/1211.5063

[25] Peterson, James L.: Computer Programs for Detecting and Correcting Spelling Errors. In: *Commun. ACM* 23 (1980), Dezember, Nr. 12, 676–687. http://dx.doi.org/10.1145/359038.359041. – DOI 10.1145/359038.359041. – ISSN 0001–0782

[26] Reps, Thomas ; Teitelbaum, Tim ; Demers, Alan: Incremental Context-Dependent Analysis for Language-Based Editors. In: *ACM Trans. Program. Lang. Syst.* 5 (1983), Juli, Nr. 3, 449–477. http://dx.doi.org/10.1145/2166.357218. – DOI 10.1145/2166.357218. – ISSN 0164–0925

[27] Rozovskaya, Alla ; Roth, Dan: Generating Confusion Sets for Context-sensitive Error Correction. In: *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2010 (EMNLP '10), 961–970

[28] Ruder, Manuel ; Dosovitskiy, Alexey ; Brox, Thomas: Artistic style transfer for videos and spherical images. In: *CoRR* abs/1708.04538 (2017). http://arxiv.org/abs/1708.04538

[29] Santos, Eddie A. ; Campbell, Joshua C. ; Hindle, Abram ; Amaral, Jos N.: Finding and correcting syntax errors using recurrent neural networks. In: *PeerJ Preprints* 5 (2017), August, e3123v1. http://dx.doi.org/10.7287/peerj.preprints.3123v1. – DOI 10.7287/peerj.preprints.3123v1. – ISSN 2167–9843

[30] Sutskever, Ilya ; Vinyals, Oriol ; Le, Quoc V.: Sequence to Sequence Learning with Neural Networks. In: *CoRR* abs/1409.3215 (2014). http://arxiv.org/abs/1409.3215

[31] Tou Ng, Hwee ; mei, wu siew ; Briscoe, Ted ; Hadiwinoto, Christian ; Hendy Susanto, Raymond ; Bryant, Christopher: *The CoNLL-2014 Shared Task on Grammatical Error Correction*. 01 2014

[32] Werbos, P. J.: Backpropagation through time: what it does and how to do it. In: *Proceedings of the IEEE* 78 (1990), Oct, Nr. 10, S. 1550–1560. http://dx.doi.org/10.1109/5.58337. – DOI 10.1109/5.58337. – ISSN 0018–9219

[33] Xie, Ziang ; Avati, Anand ; Arivazhagan, Naveen ; Jurafsky, Dan ; Ng, Andrew Y.: Neural Language Correction with Character-Based Attention. In: *CoRR* abs/1603.09727 (2016). `http://arxiv.org/abs/1603.09727`

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname:	....
Matrikelnummer:	....
Studiengang:	…………………………………………………………………………
	Bachelor ☐ Master ☐ Dissertation ☐
Titel der Arbeit:	....
	....
	....
LeiterIn der Arbeit:	....
	....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

....
Ort/Datum

....
Unterschrift