TITLE

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sven Kellenberger

01.08.2018

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

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer

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 and correction 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, 8] or some machine learning algorithm [31] to find errors and possible corrections.

Source code is very sensitive to syntax, semantic and logical errors (see Table 1.1 for definition) and therefore a similar functionality is desirable for code editors. The syntax of a programming language is strictly defined which enables integrated development environments (IDEs) to detect syntax errors before the program is even run. Of course the possibilities of an IDE are limited by the properties of the programming language, e.g. is it strongly typed or weakly typed. However, the error detection in source code is mostly limited to syntax errors, while semantic and logical errors show only at runtime or sometimes go completely unnoticed. These kinds of errors are also the hardest ones to fix. In a strongly typed language like Java, a lot of possible errors in naming and accessing attributes can be eliminated, because each variable has to be initiated before it is used and the type of the variables is known at all times and therefore also their available attributes and methods. In weakly typed languages like Ruby however, one can not determine what type of object a variable holds before runtime. This creates additional sources of runtime errors.

While syntax errors can be detected by a suitable algorithm, traditional algorithms can only hope to help prevent semantic and logical errors. This is where machine learning algorithms could step in. In the past years, deep neural networks have proven to be very effective in learning generalised concepts and applying them to single cases. For example in [26] a network is trained to transfer the style of a painting to a video sequence. That's why it should also be possible to train a network to recognize and correct certain logic errors in source code.

The aim of this project is it to train a character based sequence-to-sequence model on the task of source code correction. The implementation of the model is based on the neural machine translation (NMT) model provided by Tensorflow

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;`
		`}`

logical	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 logical error in the context of programming languages.

[18]. As a dataset the Java Github Corpus [1] is used as a source of correct data. This data is then perturbed as random syntax, semantic and logic errors are added. The performance of different model architectures is then evaluated for the introduced errors.

1.2 Outline

This thesis is divided into five chapters, including this introduction. In the second chapter an overview of the prior work on the task of spelling checking and correction is given and also some work on error detection and correction in source code. Furthermore a short introduction to the machine learning techniques and architectures used in this thesis is given. The third chapter provides a description of the used model, the training procedure and the dataset construction. In chapter four the experiments are explained and the optained results analysed. 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 common helper in our everyday lives. However, the research on this topic has begun much earlier [23].

The original motivation for a spelling checker was to find input errors in databases. For example, in [5] the authors aim to find incorrectly spelled names, dates and places in a genealogical database. This is done by computing the frequency of trigrams (three sequential characters) in the source text and based on that the probability of a character given some context, i.e. its adjacent characters. Erroneous words are found by looking at its trigrams. If a word consist of a number of unusual character combinations, it is probably spelled wrongly. However, it is easy to see that this method is not very useful for new, rare or foreign expressions like `doppelgänger` and for typos with high probabilities of being correct. Furthermore the model is limited to the vocabulary used in the text.

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 [9] 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 [8] 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 assumed to be incorrectly spelled and then give the user some possible actions to chose from like `ignore` or `replace` (compare figure

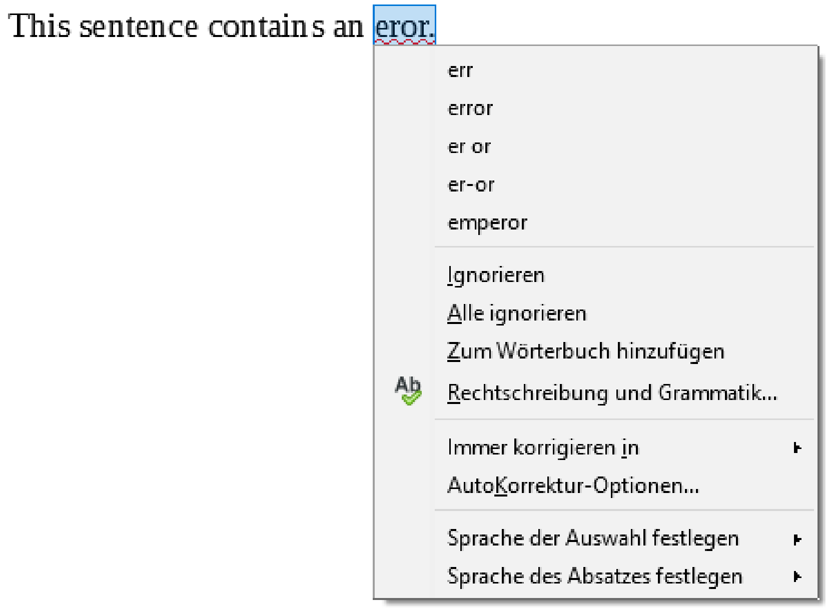

Figure 2.1: Screenshot from the LibreOffice Writer program. An example of error correction in a word processing software. The misspelled word is flagged and possible corrections are proposed to the user.

2.1).

2.1.1 Modern Research

Of course the methods described so far do still not take the context of the text 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 [4] n-grams are used to determine the most likely replacement for an error and in [11] 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 [12] or preposition errors [25].

However, even though these methods perform well on the errors they were designed for, a large amount of different classifiers is needed to catch every error. This is a costly and inflexible approach. Recent research often uses statistical machine translation methods or language models and n-grams to correct errors of multiple classes [29]. In [31] the authors train an encoder-decoder neural network with an attention mechanism which operates at the character level. [10] chose a similar approach 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 errors

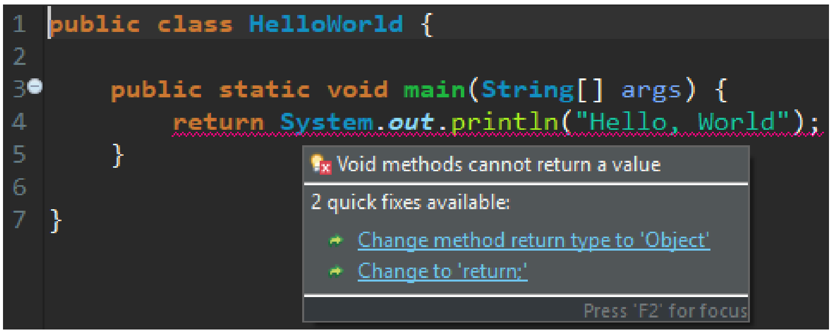

Figure 2.2: Screenshot from the Eclipse Java IDE. An example of error correction in an integrated development environment. The erroneous code is flagged and possible quick fixes are proposed to the user.

are relatively easy to find using an algorithm. This functionality can help novice programmers to avoid typical beginner's errors like a missing semicolon at the end of the line. However, semantic and logical 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 [17]. 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 of an operation which gets an unexpected parameter type only shows at runtime. For example on the one hand, the type error in

```java
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

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

shows only at runtime. Python is a weakly typed language and in general it can not 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 [15, 24].

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

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 logical 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.

Because traditional algorithms are insufficient, the attention of recent research has shifted to deep neural networks. In [3] the authors train a recurrent neural network model on the task of correcting beginners 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 feedback for such exercises thus replacing the need of a human to do so. [27] also use a recurrent architecture to predict the exact location of a syntax error and to suggest possible corrections.

Chapter 3

Model and Training

3.1 Components

This section aims to give a short review of the main techniques and architectures used in this thesis.

3.1.1 LSTM

A recurrent neural network (RNN)[30] 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, each network passes information to it's next instance as seen in figure 3.1. For an input sequence $(\mathbf{x}_1, ..., \mathbf{x}_n)$ the 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, RNNs have proven to be hard to train, especially on long-range dependencies [13]. In theory, they should be able to deal with these dependencies but either vanishing or exploding gradients usually prevent them from doing so. To solve this issue, Long Short-Term Memory networks (LSTMs) [14] were proposed. 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 if it wants to read or forget information from the memory vector or write new information onto the vector. This is done by using explicit gating mechanisms:

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

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

where 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 of $\mathbf{c}_{t-1}$ should be updated and which information from $\mathbf{c}_t$ should be written to

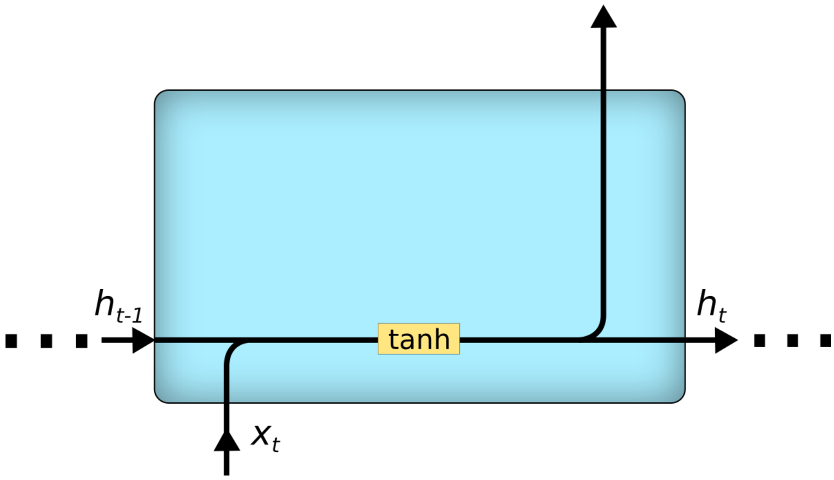

Figure 3.1: Architecture of a vanilla recurrent neural network cell. Each timestep some output and a hidden state vector $\mathbf{h}_t$ are produced by looking at the hidden state vector from the previous timestep $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$. In this simple example, $\mathbf{h}_t$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

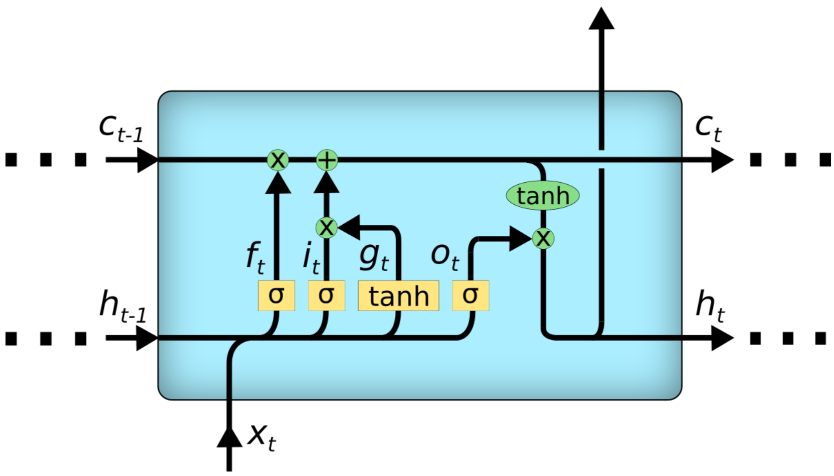

Figure 3.2: Architecture of a typical LSTM cell. In addition to the hidden state vector $\mathbf{h}_t$, a memory state vector $\mathbf{c}_t$ is passed to the next timestep. $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$ are used to compute the gates $\mathbf{f}_t$, $\mathbf{i}_t$, $\mathbf{g}_t$ and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

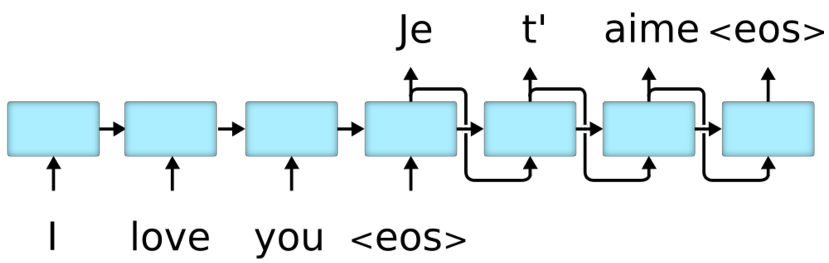

Figure 3.3: The Sequence-to-Sequence Model applied to a translation example. The English source sentence is fed to the model word by word. After the input of an end-of-sequence token (`<eos>`), the network starts producing the output sentence in French. For this the produced output tokens are fed back to the network at the next timestep. The network signals the end of the sequence by outputting another `<eos>` token.

$\mathbf{h}_t$. Finally $\mathbf{g}_t$ is a vector of possible values that (gated by $\mathbf{i}_t$) can be added to $\mathbf{c}_{t-1}$ and because of the tanh in the equation its values may range from `-1` to `1`. The state vectors are then updated as follows:

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

Almost all state-of-the-art results today are achieved using either LSTMs or networks with a similar architecture like Gated Recurrent Units (GRUs) [6] because they are easier to train and excel at capturing long range dependencies.

3.1.2 The Sequence-to-Sequence Model

Traditional Deep Neural Networks (DNNs) process the whole input 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 and only then start generating an output sequence. These problems are called sequence to sequence problems.

In [28] 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 3.3. First the whole input sequence is fed into the network and the output is ignored. Then we input an end-of-sequence token `<eos>` which signals the network to start producing the output. From there on the produced output tokens are fed to the network until the 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 actual 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 a vector which is then used to initialize the second network. It is first fed a start token `<GO>` and then the generated output until the end of the sequence is reached.

The network usually operates at word level and uses some word embedding like word2vec [20]. 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. These 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 [21] attention is used on the task of high resolution image classification. These kind 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 [19, 2]. 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 mean for the decoder to peek at previous hidden states of the encoder. This is done via 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 of the encoder $\bar{\mathbf{h}}_s$ are considered. For this an alignment vector $\mathbf{a}_t$, whose size equals the input sequence length, is calculated 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 then normalized using the softmax function.

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

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

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

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 [18]. The model consists of an encoder and a decoder with an implementation of the Luong attention mechanism [19] wrapped around the latter one. 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. 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. This 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 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. This was done to optimize training because the token which would have been the correct output is fed to the decoder instead of the possibly wrong actual output.

During backpropagation the gradients were clipped by a fixed norm. This technique is used to prevent the gradient from exploding [22]. 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 yet in part because the correction of erroneous code is a long and tedious work.

The best alternative is for the errors to be self introduced to a dataset of correct code. 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 [1] 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. Also, 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 [16]. 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. Thus 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 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 on their corruptibility. This means, that all corruptions (see subsection 3.3.2) had to be able to be applied to the method. This assures that the defined corruption rate is met.

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 learnt model cannot be applied to real world examples. Also a large variety in the introduced errors would 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 level.

The corruption of the data was done randomly during training. Each corruption was applied equally often, while the percentage of uncorrupted examples varied. 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 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 are 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
public int[] seedToArray(int seed){
  int[] seeds = new int[1];
  seeds[0] = seed;
  return seeds;
}
```

Lets assume that `seed` is to be corrupted. One possible misspelling would be to add a random character. If per chance 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 different type of error and the model shouldn'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 character are selected it would have no effect and therefore no error would be added. However these two scenarios are very specific and 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 imposes sometimes unsolvable problem to the network. One case is pretty simple. If the return type of the method is changed from `void` to something different, the model needs only check if a return statement is present and if not, the return type should be changed to `void`. The second case is more complicated as the return type is changed from one type to `void`. Again the model can determine if the return type `void` is correct by looking if a return statement is present. 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:

```
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. This is the tradeoff of only looking at methods 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. 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 a logic error, but it still didn't guarantee it. Consider the following example:

```
public int squareSum(int a, int b){
  int squareA;
  int squareB;
  squareA = a * a;
  squareB = b * b;
  return squareA + squareB;
}
```

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 do its purpose which was confirmed by the experiments.

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) is then selected and misspelled. Possible misspellings are: removal of a random character, insertion of a random character, switch 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 methods 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	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 all corruptions.

Chapter 4

Experiments and Results

4.1 Architecture and Training

4.1.1 Corruption Rate

Because the source sequence and target sequence are almost the same and the errors are self-introduced, it is fair to ask 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% posts the best result. The models with lower percentages don't pick up on the errors as well while the model with the 100% corruption rate lacks the ground truth. In general the model learns to correct all of the introduced errors. It performs especially well when inputting a sequence missing a single semicolon which is of course also the simplest task to solve. However the model is also able to correct the other errors reasonably well. A missing bracket or an incorrect return type are corrected most of the time while the model has a little more trouble correcting a misspelled variable or realigning switched lines. A more detailed analysis for the different error types is given in section 4.2.

What's also interesting to see is how well the model performs on uncorrupted sequences. Even the model with 100% corruption rate, i.e. which never gets an uncorrupted sequence as input, manages 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 only corrects where necessary.

4.1.2 Attention Mechanism

Of course it can also be asked if the attention mechanism is necessary for the model, after all it increases complexity and training duration. To test this, the model was twice trained without an 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 [28], 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 at any given timestep.

	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: Results

The experiments 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 (see table 4.1b). These models never learnt 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 learnt 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 passages they remembered from training. However after about 4,000 iterations the model with the attention mechanism learnt to utilise 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 and perform a little bit better 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 correction of source code, 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

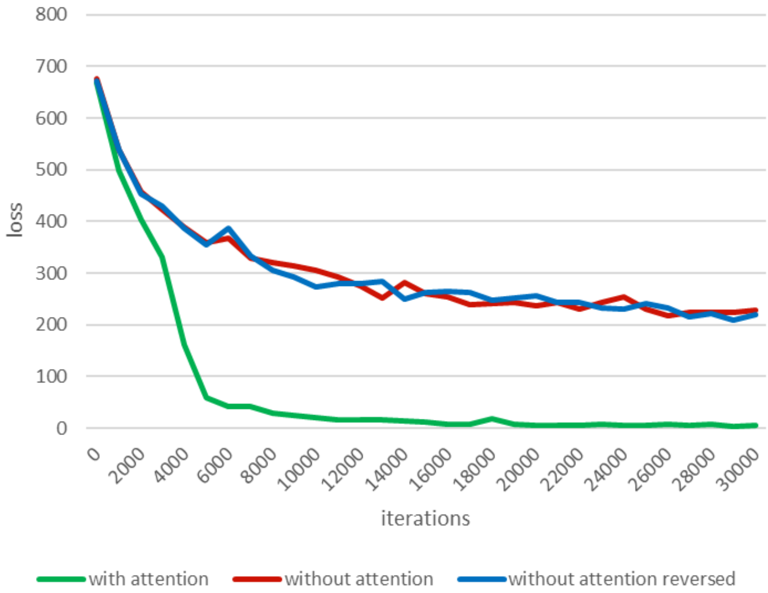

Figure 4.1: Line chart showing the loss of different models over time.

Figure 4.2: Line chart showing the loss of different models over time.

found in table 4.1c.

As could be expected the vanilla RNN performed the worst. As explained earlier, these networks struggle to learn 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 [7]. 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 are 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 on the lower boundary this 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 analysed. 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 interesting 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 (only one off), 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.

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 a loop and output some parts or lines of the input sequence multiple times. This is something that can often be observed in 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 from 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. While the correction of a missing semicolon scores already good with 71.8% accuracy it is also worth to look at the results of an evaluation with the same "one off tolerance" 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 missing 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 in inserting missing brackets into the sequence but the test accuracy of 51.3% seems to contradict this assumption. However a closer look at the results reveal that the task of inserting a missing bracket isn't that trivial. Consider the following example:

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

Here the opening brackets 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 because there is not white space indicating the location of the missing bracket.

To evaluate 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. The brackets were also correctly nested 76.9% of the time which indicates that the model did learn the concept of brackets well.

4.2.4 Misspelled Variable

Including corrected wrong instance: 45.6%

4.2.5 Wrong Return Type

different percentages

4.2.6 Line Switch

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

Table 4.2: VC = Variable Declaration, MI = method invocation, AS = assignment.

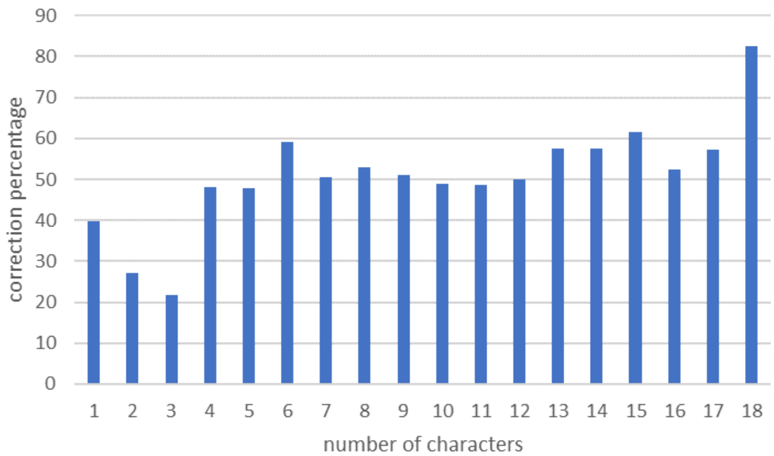

Figure 4.3: Correction percentage for variables evaluated for different lengths. Only lengths with more than 10 examples were considered.

Chapter 5

Conclusion

5.1 Future Work

Appendix A

Used Notation

A.1 Naming Conventions

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

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

If no other indication of the nature of the variable is given, it is a vector resp. 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.

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

The concatenation of vectors is abbreviated as follows:

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

Appendix B

Showcase

This is a showcase

Uncorrupted Input	Output

<pre>public final void enable() {
  FileConfiguration config = new
       FileConfiguration(NoLagg.plugin);
  config.load();
  this.enable(config);
  config.save();
}</pre>
	<pre>public final void enable() {
	 FileConfiguration config = new
	      FileConfiguration(NoLagg.plugin);
	 config.load();
	 this.enable(config);
	 config.save();
	}</pre>

<pre>private Vec4 parseVec4(String s) {
 Scanner snr = new Scanner(s);
 Vec4 res = new Vec4();
 res.x = Float.parseFloat(snr.next());
 res.y = Float.parseFloat(snr.next());
 res.z = Float.parseFloat(snr.next());
 res.w = Float.parseFloat(snr.next());
 return res;
}</pre>
	<pre>private Vec4 parseVec4(String s) {
	 Scanner snr = new Scanner(s);
	 Vec4 res = new Vec4();
	 res.x = Float.parseFloat(snr.next());
	 res.y = Float.parseFloat(snr.next());
	 res.y = Float.parseFloat(snr.next());
	 res.w = Float.parseFloat(snr.next());
	 return res;
	}</pre>

Table B.1: Example

<table>
<tr><th>Brackets Input</th><th>Output</th></tr>
<tr><td><pre>int pop(int numBits) _
 int i = getLeadingAsInt(numBits);
 truncate(numBits);
 return i;
}</pre></td><td><pre>int pop(int numBits) {
 int i = getLeadingAsInt(numBits);
 truncate(numBits);
 return i;
}</pre></td></tr>
<tr><td><pre>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));
}</pre></td><td><pre>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));
}</pre></td></tr>
</table>

Table B.2: Example

<table>
<tr><th>Semicolons Input</th><th>Output</th></tr>
<tr><td><pre>@Override
public Method run() {
 try {
  final Method mtd = clazz.getMethod("writeReplace")_
  mtd.setAccessible(true);
  return mtd;
 } catch (NoSuchMethodException e) {}
 return null;
}</pre></td><td><pre>@Override
public Method run() {
 try {
  final Method mtd = clazz.getMethod("writeReplace");
  mtd.setAccessible(true);
  return mtd;
 } catch (NoSuchMethodException e) {}
 return null;
}</pre></td></tr>
<tr><td><pre>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())_
}</pre></td><td><pre>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());
}</pre></td></tr>
</table>

Table B.3: Example

Variable 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);`
	`}`

Table B.4: Example

<table>
<tr><th>Return type Input</th><th>Output</th></tr>
<tr><td>

```
@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();
}
```

</td><td>

```
@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();
}
```

</td></tr>
<tr><td>

```
@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;
}
```

</td><td>

```
@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;
}
```

</td></tr>
</table>

Table B.5: Example

<table>
<tr><th>Switch Input</th><th>Output</th></tr>
<tr><td>
<pre>
protected void disposeElementInfo(Object element,
    ElementInfo info) {
 if (info instanceof ResourceSetInfo) {
  >resourceSetInfo.dispose();
  >ResourceSetInfo resourceSetInfo = (ResourceSetInfo)
      info;
 }
 super.disposeElementInfo(element, info);
}
</pre>
</td><td>
<pre>
protected void disposeElementInfo(Object element,
    ElementInfo info) {
 if (info instanceof ResourceSetInfo) {
  >ResourceSetInfo resourceSetInfo = (ResourceSetInfo)
      info;
  >resourceSetInfo.dispose();
 }
 super.disposeElementInfo(element, info);
}
</pre>
</td></tr>
<tr><td>
<pre>
private void resolveEntry(Entry < K, T > entry) {
 >entry.isResolved = true;
 >resolved.add(entry);
 resolved(entry);
}
</pre>
</td><td>
<pre>
private void resolveEntry(Entry < K, T > entry) {
 >entry.isResolved = true;
 >resolved.add(entry);
 resolved(entry);
}
</pre>
</td></tr>
</table>

Table B.6: Example

List of Tables

List of Figures

Bibliography

[1] 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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

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

[14] 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

[15] 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

[16] 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

[17] 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

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

[19] 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

[20] 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

[21] 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

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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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