

TITLE

Bachelorarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sven Kellenberger

01.08.2018

Leiter der Arbeit:
TITLE NAME
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, consectetur 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

1	Introduction	1
1.1	Motivation	1
1.2	Outline	2
2	Background	3
2.1	Origins of Spelling Checkers and Correctors	3
2.1.1	Modern Research	4
2.2	Automatic Correction of Source Code	4
3	Model and Training	7
3.1	Components	7
3.1.1	LSTM	7
3.1.2	The Sequence-to-Sequence Model	9
3.1.3	Attention-Mechanism	10
3.2	Dataset Construction	11
3.2.1	Preprocessing	11
3.2.2	Corruption	11
3.3	Model Implementation	13
4	Experiments and Results	14
5	Conclusion	15
5.1	Future Work	15
A	Appendix	16
	List of Tables	18
	List of Figures	18
	Bibliography	19

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 [8, 7] or some machine learning algorithm [29] 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 [24] 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.	<pre>public int add(int a, int b){ int sum := a + b; return sum; }</pre>
semantic	Incorrect usage of a variable or statement.	<pre>public void add(int a, int b){ int sum = a + b; return sum; }</pre>
logical	Failure to comply with the programs requirements.	<pre>public int add(int a, int b){ int sum = a - b; return sum; }</pre>

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

[17]. 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 obtained 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 [21].

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 [8] 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 [7] 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

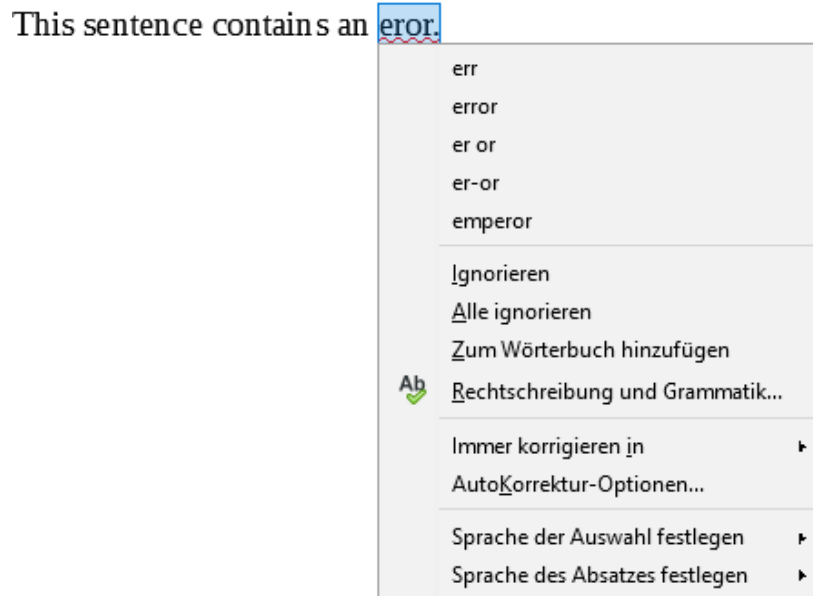


Figure 2.1: Screenshot from the LibreOffice Writer program. An example of error correction in a word processing software. The misspelled word is flagged and possible corrections are proposed to the user.

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 [10] 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 [11] or preposition errors [23].

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 [27]. In [29] the authors train an encoder-decoder neural network with an attention mechanism which operates at the character level. [9] 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

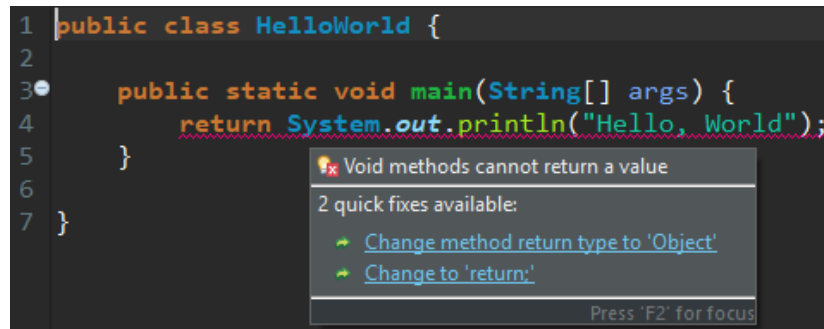


Figure 2.2: Screenshot from the Eclipse Java IDE. An example of error correction in an integrated development environment. The erroneous code is flagged and possible quick fixes are proposed to the user.

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 [16]. 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

```
1 public void printNumber(String n){
2     System.out.println("Number: " + n);
3 }
4 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

```
1 def print_number(n):
2     print("Number: " + n)
3 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, 22].

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. [25] 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)[28] 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 $(\mathbf{x}_1, \dots, \mathbf{x}_n)$ 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 [12]. 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) [13] 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:

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

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

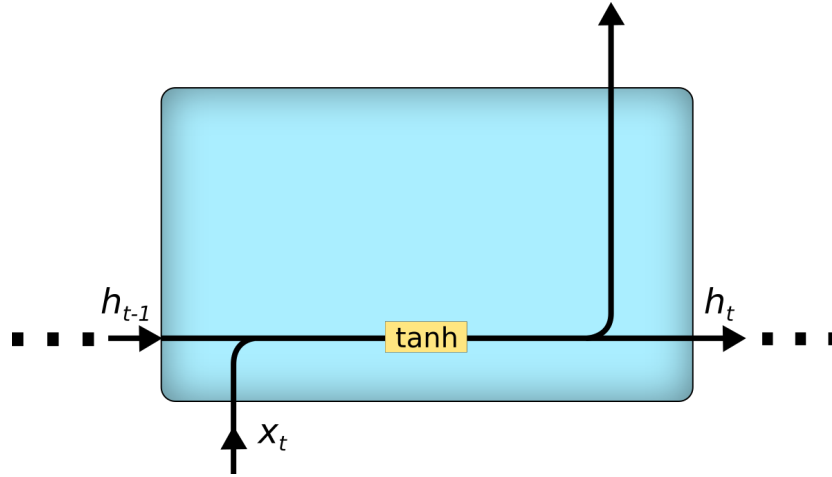


Figure 3.1: Architecture of a vanilla recurrent neural network cell. Each timestep some output and a hidden state vector \mathbf{h}_t are produced by looking at the hidden state vector from the previous timestep \mathbf{h}_{t-1} and the current input \mathbf{x}_t . In this simple example, \mathbf{h}_t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

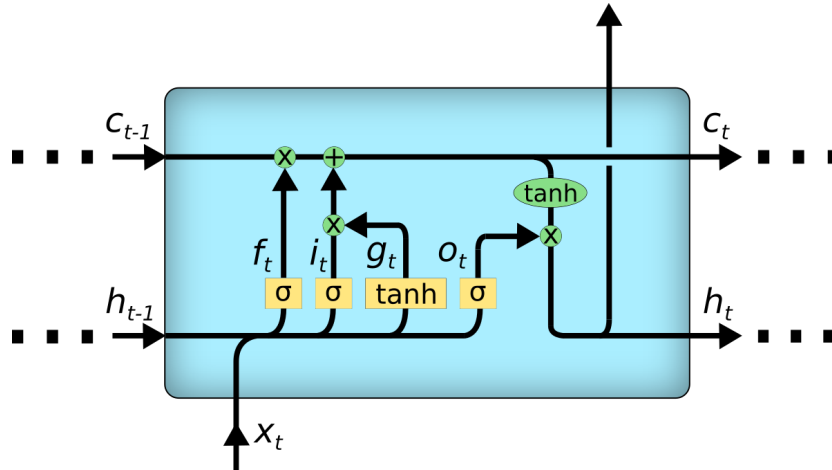


Figure 3.2: Architecture of a typical LSTM cell. In addition to the hidden state vector \mathbf{h}_t , a memory state vector \mathbf{c}_t is passed to the next timestep. \mathbf{h}_{t-1} and the input \mathbf{x}_t are used to compute the gates \mathbf{f}_t , \mathbf{i}_t , \mathbf{g}_t and \mathbf{o}_t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

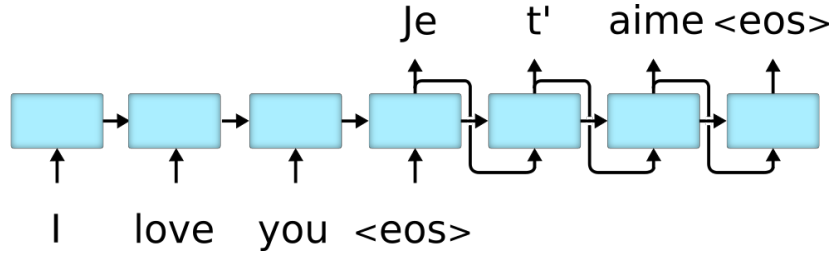


Figure 3.3: The Sequence-to-Sequence Model applied to a translation example. The English source sentence is fed to the model word by word. After the input of an end-of-sequence token (<eos>), the network starts producing the output sentence in French. For this the produced output tokens are fed back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

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

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 [26] 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 $\langle GO \rangle$ 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 [19]. 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 choose on which information to focus at any given moment. For example in [20] 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 [18, 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 a_t are then normalized using the softmax function.

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

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

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

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

$$\mathbf{c}_t = \sum_{s'} a_t(s') \bar{\mathbf{h}}_{s'}$$

3.2 Dataset Construction

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, 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 Corpus 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.

3.2.1 Preprocessing

Several operations were performed during preprocessing for each Java file in the dataset separately:

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 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 3.2.2) had to be able to be applied to the method.
6. All suitable methods were then written to new files (ca. 100MB each), one method per line.

3.2.2 Corruption

The corruption of the data was done randomly during training. 75% of the data was left uncorrupted, meaning that the source sequence and the target sequence were identical. To the other 25% one of five corruptions was applied, with each one being equally frequent.

Of the possible corruptions, two produced syntax errors, two semantic errors and one logic errors. The syntax error consisted of removing a bracket or a

Corruption	Error Type	Explanation	Example
missing bracket	syntax	One random bracket (regular, curved or squared) is selected and removed from the source.	<pre>public int add(int a, int b{ int sum; sum = a + b; return sum; }</pre>
missing semicolon	syntax	One random semicolon is selected and removed from the source.	<pre>public int add(int a, int b){ int sum; sum = a + b return sum; }</pre>
misspelled variable	semantic	A random variable which is being declared in the source is selected. A random occurrence of the variable (not 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.	<pre>public int add(int a, int b){ int sum; sum = a + b; return summ; }</pre>
incorrect return type	semantic	The return type of the methods is changed. There are only two possibilities. If the return type is <code>void</code> , it is changed to one of Java's primitive data types (<code>int</code> , <code>float</code> , etc.). In any other case, the return type is changed to <code>void</code> .	<pre>public void add(int a, int b){ int sum; sum = a + b; return sum; }</pre>
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.	<pre>public int add(int a, int b){ sum = a + b; int sum; return sum; }</pre>

Table 3.1: Description of all possible corruptions.

semicolon, for the semantic errors a variable was misspelled or the return type of the source changed and the logic errors were produced by switching the order of statement lines. More detailed explanations of the different corruptions are given in table 3.1.

3.3 Model Implementation

Chapter 4

Experiments and Results

Chapter 5

Conclusion

5.1 Future Work

Appendix A

Appendix

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, consectetur 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

List of Tables

1.1	The definition of syntax, semantic and logical error in the context of programming languages.	2
3.1	Description of all possible corruptions.	12

List of Figures

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.	4
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.	5
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.	8
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 .	8
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 ($\langle \text{eos} \rangle$), 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 $\langle \text{eos} \rangle$ token.	9

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, Çağlar ; 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] 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](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](https://doi.org/10.1016/S0019-9958(68)90201-5). – ISSN 0019-9958
- [8] 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
- [9] 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>

-
- [10] 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
- [11] 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
- [12] HOCHREITER, Sepp: Untersuchungen zu dynamischen neuronalen Netzen. (1991), 04
- [13] 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
- [14] KARPATHY, Andrej ; JOHNSON, Justin ; LI, Fei-Fei: Visualizing and Understanding Recurrent Networks. In: *CoRR* abs/1506.02078 (2015). <http://arxiv.org/abs/1506.02078>
- [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] 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
- [17] LUONG, Minh-Thang ; BREVDO, Eugene ; ZHAO, Rui: Neural Machine Translation (seq2seq) Tutorial. In: <https://github.com/tensorflow/nmt> (2017)
- [18] 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>
- [19] 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>
- [20] MNIH, Volodymyr ; HEES, 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
- [21] 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

- [22] 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
- [23] 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
- [24] 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>
- [25] 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
- [26] 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>
- [27] 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
- [28] 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
- [29] 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 Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....

Ort/Datum

.....

Unterschrift