FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jakub Náplava

Natural Language Correction

Institute of Formal and Applied Linguistics

Supervisor of the master thesis:	RNDr. Milan Straka, Ph.D.
Study programme:	Informatics
Study branch:	Artificial Intelligence

Prague 2017

Title: Natural Language Correction

Author: Bc. Jakub Náplava

Department: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Milan Straka, Ph.D., Institute of Formal and Applied Linguistics

Abstract:

The goal of this thesis is to explore the area of natural language correction and to design and implement neural network models for a range of tasks ranging from general grammar correction to the specific task of diacritization. The thesis opens with a description of existing approaches to natural language correction. Existing datasets are reviewed and two new datasets are introduced: a manually annotated dataset for grammatical error correction based on CzeSL (Czech as a Second Language) and an automatically created spelling correction dataset. The main part of the thesis then presents design and implementation of three models, and evaluates them on several natural language correction datasets.

In comparison to existing statistical systems, the proposed models learn all knowledge from training data; therefore, they do not require an error model or a candidate generation mechanism to be manually set, neither they need any additional language information such as a part of speech tags. Our models significantly outperform existing systems on the diacritization task. Considering the spelling and basic grammar correction tasks for Czech, our models achieve the best results for two out of the three datasets. Finally, considering the general grammatical correction for English, our models achieve results which are slightly worse, but still comparable to the previous state-of-the-art model.

Keywords: natural language correction, deep learning, neural networks

I would like to express my deep gratitude to my supervisor, RNDr. Milan Straka, Ph.D. for the patient guidance, numerous pieces of advice and corrections, without which the thesis would not be possible.

I would also like to thank my family for their never ending support and understanding.

Contents

1. Introduction

In today's society, people write numerous texts on a daily basis. There are emails to your colleagues or text messages to your friends, sent using your smartphone either as an SMS or more likely via social networks' messengers. However, these represent only a fraction of examples as there are countless other different types of documents everyone needs to write from time to time. As a result of our lack of time, patience and sometimes even knowledge to review and correct our texts, these texts often contain multiple errors. The types of created errors range from simple errors such as missing characters (*accidently* instead of *accidentally*) to errors which are more complex as they usually depend on the word context (*he write* instead of he *writes*) and occasionally require the entire word reorganization.

Two other uses of language correction are worth mentioning. First, there is pre-processing texts for another task, which can be inter alia text translation and text analysis. The issue here lies in the fact that such tasks may show disappointing performance whilst running on uncorrected input texts as most systems assume input texts to be correct. Second, there are post-processing results of OCR (Optical Character Recognition) and speech-to-text tools. Both these tools convert their input to textual output which however often contains multiple errors.

The aim of natural language correction is to both detect and correct errors in the input text. The major difficulty in this discipline stems from the existence of multiple corrections for a single text. Imagine a simple sentence *Do he go there?*. The most intuitive correction would be to change the first word to *Does* resulting in a new sentence *Does he go there?*. However, if we know from the preceding part of the text that there were two people, it is also possible to correct the sentence as *Do they go there?*.

Natural language correction has been studied for a long time. In 1971, Ralph Gorin created the first spell checker for general English texts. In the late 1970s, spell checkers became widely available on mainframe computers, and in the early 1980s standalone spell-checking programs for personal computers appeared and were also integrated into text editors and adopted for other languages in the subsequent years. Later, first grammar checkers appeared. Today, there are spell- and grammar-checkers widely available in text editors as well as standalone applications. Natural language correction tools which are currently implemented use mainly statistical and rule-based approaches. However, in recent years, deep neural networks have been achieving remarkable results in this area. Since 2014, they have been utilized in natural text processing, achieving state-of-the-art results in machine translation, dependency parsing, named entity recognition, and in many other text processing applications. These have been the main motivation for us to further explore the usage of deep neural networks in the area of natural language correction.

1.1 Goals

Inspired by the prevailing successes of deep neural networks, we decided to explore in detail their capabilities in regards to natural language correction tasks.

The primary goal of the thesis is to design and implement architectures for deep neural networks which would be suitable for different types of natural language correction tasks. These implemented models will be then evaluated and their performance compared to existing systems.

In order to accomplish our goals, the following tasks were performed:

- we acquired and created several datasets
- we designed several models for a range of tasks from general grammar correction to specific task of diacritization
- we performed experiments and compared our systems to existing ones
- we documented and released all software used for the experiments

All these tasks were performed exclusively by the thesis author.

1.2 Thesis structure

In the following chapter, we describe the most important tasks of natural language correction including tasks focused solely on Czech. Thereafter, we describe several approaches of tools which are currently used and cover utilized datasets. In Chapter 3, 4 a 5 we discuss our three models. In Chapter 6, we perform several experiments and present their results together with comparison to existing systems. Chapter 7 contains a description of the implementation together with a basic manual on how to re-run the experiments. Finally, Chapter 8 concludes this thesis with a summary of achieved outcomes.

2. Problem analysis

In this chapter, we explore existing resources for natural language correction. First of all, we describe the most important tasks of natural language correction. Then, we inspect approaches of several common tools that solve these tasks nowadays. After that, we analyze papers containing new interesting approaches to challenges in this field. Finally, we discuss existing language correction datasets.

2.1 Natural language correction tasks

Natural language correction is a task whose goal is to correct an erroneous text. Consider for example the following sentence:

He come from Czech Republci.

There are three errors in the given example: subject-verb agreement error (*He come* instead of *He comes*), article error (missing article *the*) and a misspelled word *Republic.*

Given these errors, one can identify two main problems of natural language correction: spell checking (spelling correction) and grammar checking (grammatical error correction). Spell checking basically attempts to detect and provide spelling suggestions for incorrectly spelled words, whereas grammar checking tries to detect and correct grammatical errors in the text. In comparison, the grammar checking task is obviously a substantially more difficult, because it includes language errors such as wrong word order, verb tense errors or already mentioned subject-verb agreement errors.

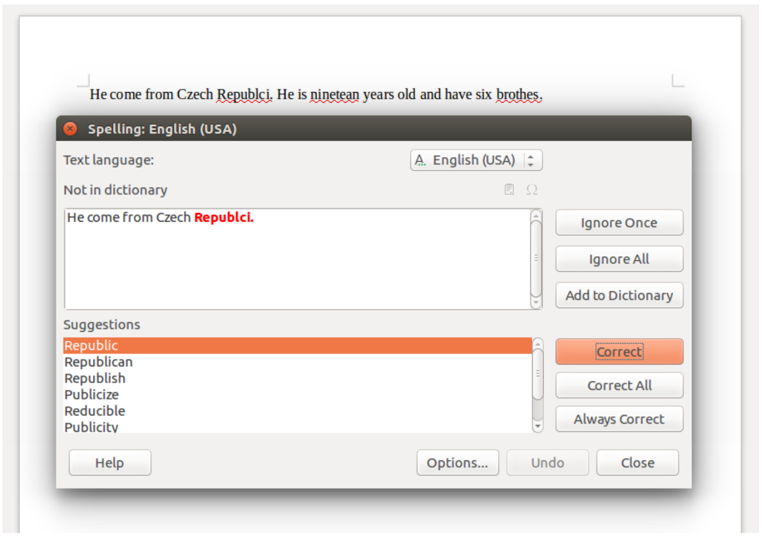
We also mention the task of diacritics generation (diacritization) for Czech texts in this thesis. The Czech language contains several special letters that are not contained in the standard ASCII table. These are composed of a standard letter (e.g. *a, e*) with a diacritic sign (e.g. acute or caron). As mentioned by Vrána [2002], Czechs often write without diacritics, which can be seen as an "intended creation" of errors. There are several reason for this phenomenon:

- Many Czechs use English keyboard, which does not contain characters with diacritics.
- Smartphones (tablets) keyboards contain these characters but almost always in different keyboard layouts.

- Before UTF-8 encoding became widespread, there were encoding problems when moving text between different platforms and many people still have this in mind.

2.2 Existing tools

While writing on a computer, many people use word processors such as Microsoft Office Word or LibreOffice Writer. These programs often have integrated spell checkers and sometimes also basic grammar checkers. To warn users that a certain word may be misspelled, these programs usually underline the word with red colour and show candidate corrections when right clicked. When a grammatical error is detected, the whole segment in question is underlined with green color and sometimes also the grammar rule breach is reported. The error detection module of LibreOffice Writer is shown in Figure 2.1.


He come from Czech Republci. He is ninetean years old and have six brothes.

Spelling: English (USA)

Text language: English (USA)

Not in dictionary

He come from Czech **Republci.**

Suggestions

Republic
Republican
Republish
Publicize
Reducible
Publicity

Figure 2.1: LibreOffice Writer proposing correction on a sample sentence.

Using natural language correction tools integrated in word processors is a great advantage while writing text, however for handling already written text it may be better to use standalone applications. These applications are often of a much smaller size, have command-line or web interface and sometimes also perform better.

Probably the most used open-source spell checker is Hunspell.[1] It is integrated in a variaty of programs such as already mentioned Libre Office, Open Office, Mozzila Thunderbird or Google Chrome. Hunspell is a C++ tool released under GPL/LGPL/MPL tri-license. Amongst other spell checkers, it is definitely worth

[1]`http://hunspell.github.io/`

mentioning GNU Aspell[2] and Korektor,[3] which was a state-of-the-art spell checker and diacritics generator for Czech texts in 2012 [Richter et al., 2012].

Grammar correction modules contained within current word processors are mostly rule-based. They contain a set of grammar rules that are checked at runtime. Although some grammar checkers claim to be using more advanced approaches like machine learning, they still correct only a set of error types. The development of a well performing general grammar checker is definitely at the forefront of current research.

2.3 Spelling correction

In this section, we first describe a rather simple approach to spelling correction. Subsequently, we extend its ideas and explain how a state-of-the art spelling corrector for Czech language works.

2.3.1 Simple statistical spelling corrector

In 2007, Petr Norvig wrote an excellent blogpost *How to Write a Spelling Corrector*.[4] The motivation for it was a simple spell checker implemented in Google's search engine which was capable of detecting errors and offering most probable replacements. Therefore, the goal of this task was to find the most probable correction of a single word.

To formalize the task of finding most probable correction, it is useful to consider it as a sequence decoding problem. In natural language processing (NLP), this is often formulated in terms of noisy channel model.[5] To understand ideas behind this theory, imagine we have a transmitter sending a sequence of symbols (in this case a single word) to a receiver. The transmission channel is, however, imperfect and, thus, the transmitted sequence may be modified during the transfer. This potentially distorted sequence is then received by the receiver, who wants to decode the original message. To decode the message, the receiver may use the knowledge of transmission channel properties, e.g. probabilities of specific errors, and of the input data properties, e.g. single word occurrences.

Let w be a distorted received word. Then we're looking for a word c, out of all possible candidate corrections, that maximizes that c is a desired correction,

[2]http://aspell.net/

[3]http://ufal.mff.cuni.cz/korektor

[4]http://norvig.com/spell-correct.html

[5]https://en.wikipedia.org/wiki/Noisy_channel_model

given the word w:

$$c = \operatorname*{argmax}_{c \in candidates(w)} P(c|w)$$

Using the knowledge of Bayes' Theorem, we can rewrite this as:

$$c = \operatorname*{argmax}_{c \in candidates(w)} \frac{P(w|c) \cdot P(c)}{P(w)}$$

Since P(w) is the same for each possible candidate word c, this can be further simplified by omitting this probability:

$$c = \operatorname*{argmax}_{c \in candidates(w)} P(w|c) \cdot P(c)$$

We can identify four main components in the final expression:

1. Selection mechanism – We select a candidate correction with the highest probability.

2. Candidate model – A list of all candidates that will be considered. It is common to generate candidates that are valid words within certain edit distance.[6] Petr Norvig in his work uses the Damerau–Levenshtein distance, wherein the set of allowed operations is character insertion, deletion, substitution and transposition of two adjacent letters. The choice of this metric was probably motivated by Damerau [1964]. In this work F. J. Damerau stated that these four operations cover more than 80 percent of all human misspellings.

3. Error model $P(w|c)$ – A probabilistic model that estimates a noisy channel transmission posterior probability for all possible values of w and c is ussualy denoted as an error model. These probabilities should be estimated from data, however, for simplicity, Petr Norvig uses edit distance to approximate them. In this way, the fewer operations is required to transform the received word w to candidate word c, the higher the probability $P(w|c)$.

4. Language model $P(c)$ – A probabilistic model that estimates a priori probability of input word c is called language model. In the blogpost, this probability was estimated on the word level. Therefore, $P(c)$ is estimated as the number of occurrences of word c in some, ideally large, corpus divided by a total number of words.

The accuracy of such simple spelling corrector was not as high as one would want to, nevertheless, in the next section we show a way to make it better.

[6] edit distance – minimum number of operations required to transform one string into the other

2.3.2 Korektor

Korektor is a statistical Czech spell checker, which started as a diploma thesis of Michal Richter [Richter, 2010]. In 2012, Korektor was the state-of-the art tool for Czech, both as the spell checker and the diacritics generator [Richter et al., 2012].

Korektor is the spell checker which is significantly more sophisticated than the previously discussed one, even though many concepts are still the same. The great difference is that Korektor corrects whole sentences instead of single words. This naturally brings several issues that need to be solved.

When we consider the noisy model approach, we can identify two features influencing the result: error model and language model. However, sometimes we would like to use more than just two features. Fortunately, there is a generalization of noisy channel approach called log-linear model. This method allows using any number of weighted features. To formalize, let $w_1^*...w_n^*$ be an optimal sequence of words, $w_1^{'}...w_n^{'}$ be an input sequence of words, $f_{j=1}^N$ set of feature functions and $\alpha_{j=1}^N$ their weights. Then we can define log-linear model as follows:

$$(w_1^*...w_n^*) = \underset{(w_1...w_n)}{\operatorname{argmax}} \sum_{j=1}^{N} \alpha_j \cdot f_j(w_1...w_n, w_1^{'}...w_n^{'})$$

Korektor utilizes 4 feature functions:

- Transmission channel feature – Based on error model probability $P(w|c)$, where w is the received and potentially distorted word and c is the original word form. The probabilities are now assigned with respect to the solved problem. For spelling correction task, specific typing errors are taken into account. For diacritics generation, all words that could possibly come from c by adding diacritics have uniform cost and all other an infinite cost.

- Word forms feature – Based on language model probability $P(w_i|w_{i-2}, w_{i-1})$, where w_i is the current word and w_{i-2} and w_{i-1} are the previous ones.

- Morphological lemma[7] feature – Based on language model probability $P(l_i|l_{i-2}, l_{i-1})$ and emission model probability $P(w_i|l_i)$, where l_i stands for the current word lemma and l_{i-2} and l_{i-1} for the previous ones.

- Morphological tag[8] feature - Based on language model probability $P(t_i|t_{i-2}, t_{i-1})$ and emission model probability $P(w_i|t_i)$, where t_i stands for the current word part of speech tag and t_{i-2} and t_{i-1} for the previous ones.

[7]lemma – canonical form, dictionary form, or citation form of a set of words

[8]part of speech – category of words that have a similar grammatical properties

The transmission channel feature for spelling correction task was estimated from a manually created transcript of an audio book. All other feature functions were estimated from a large text corpus.

Since Korektor corrects whole sentences, it is no longer desirable to generate all possible sentences and re-rank them afterwards. Instead, Hidden Markov model is built. The vertices in this model contain corrections up to predefined edit distance. The transition costs are then composed of morphological lemma feature, morphological tag feature and word forms feature. Emission probabilities in each state are based on an error model. To find an optimal correction in the created graph, authors utilize Viterbi algorithm [Forney, 1973].

2.4 Grammatical error correction

Since grammatical error correction (GEC) includes errors such as wrong word order or missing article, the approach used for spelling correction can not be directly applied. The reason is that the spelling correction solution assumes that there are errors mainly in the words themselves. The operation of word reorganization or word deletion is then an operation which is too unlikely and thus never considered.

Early research in GEC aimed to solve single error types' correction. An example can be found in [Knight and Chander, 1994]. In this work, the authors built a system for correcting articles on machine translated outputs. The task of correcting articles can be seen as a classification task, because the goal is to choose one article out of a fixed set.

The classification approach has been further improved in article correction [Chodorow et al., 2007, Han et al., 2006], preposition correction [Xi et al., 2010] and also verb correction [Rozovskaya et al., 2014b]. To train a classifier, most of the works use features extracted from word context. These are typically word n-grams or part of speech tags. To create a grammatical error correction system, a set of these classifiers is used in a hybrid system.

In 2006, Brockett et al. [2006] showed that using statistical machine translation (SMT) for GEC task may be a good idea. He performed experiments, in which he used the SMT system for correcting mass noun errors and reached promising results. The obvious disadvantage of SMT is that it needs a lot of data. Therefore, there were practically no SMT systems correcting multiple errors up to 2011.

In 2011, Helping Our Own (HOO) shared task comprising on GEC was announced [Dale and Kilgarriff, 2010]. It came with an annotated corpus of English as a second language learners and a separate test set. In next three years, HOO 2012 [Dale et al., 2012] and the CoNLL 2013 and 2014 [Ng et al., 2014] shared tasks were organized, all focusing on GEC tasks and containing training as well as test sets. This naturally attracted attention of many researchers.

State-of-the-art performance on these datasets was, up to 2016, gradually outperformed both by classification approach [Dahlmeier et al., 2012, Rozovskaya et al., 2014a], SMT approach [Felice et al., 2014, Grundkiewicz, 2014] and a combination of them [Susanto, 2015].

In 2016, Yuan and Briscoe [2016] achieved new state-of-the-art results using a neural machine machine translation system (NMT) that operated on the word level. Because input texts of GEC consists of relatively many rare (out-of-vocabulary) words, these were handled in a special way. Later, Xie et al. [2016] showed that using NMT system that operates on the character level can both solve the problem with out-of-vocabulary words and achieve new state-of-the-art results. To the best of our knowledge, currently best system is SMT system of Chollampatt et al. [2016] that incorporates two additional neural networks to help SMT system generalize better.

2.5 Diacritization

Antonín Zrůstek in his thesis [Zrůstek, 2000] conducted statistical experiments on two Czech corpora (ESO, DESAM). He found out that approximately 60 percent of words in corpora have a unique diacritics assignment, i.e. when the diacritics is removed, there is only one existing word that can be created by adding diacritics. He also discovered that around 20 percent of occurring words have multiple diacritics variants. The rest 20 percent of words were not recognized by morphological analyzer and were marked as foreign words.

Vrána [2002] points out that when we take into account word frequencies of those corpora, we can conclude that about 70-80 percent of words have unique diacritics assignment, 20-30 percent have multiple assignments and 0.05 percent were not recognized. He also notes to keep this in mind when evaluating diacritization system performance. Accuracy of 96 percent basically means that a system made an error at each sixth word with non-unique diacritics assignment.

Both Zrůstek [2000] and Vrána [2002] developed systems that use very similar approach as Korektor, which (as mentioned in Section 2.3.2) is the state-of-the-art tool for diacritization. I suspect that mainly the used language model is the reason, why their results are not as good as those of Korektor.

2.6 Datasets

To train a natural language corrector, one needs to know typical errors that people make. A language correction dataset should, therefore, consist of text pairs in both corrected and original versions. Depending on the dataset, these pairs are mostly sentence aligned or word aligned. Consequently, the majority of errors are made on the sentence or word level, even if one can in certain scenarios imagine also reordering of whole sentences.

Except for error corpus itself, many natural language correction approaches require also large corpora of clean text to train a language model. Since error datasets are usually of very limited size, other datasets are used in this place. As we describe in Section 4.6, these datasets can be further used to create artificial error data.

2.6.1 Lang-8 Learner Corpora

Lang-8 is social network intended for users trying to learn a new language. To improve foreign language skills, users write texts (e.g. articles, reviews or current news) in the language they are trying to learn and usually also rewrite the same text in their native language. After doing so, other users, who try to learn the opposite language, read the bilingual texts and optionally correct them. In this way, this social network helps people learn a new language and also create new relationships. Currently, there are over 750 000 registered members speaking more than 90 languages.

In December 2010, Lang-8 released *Lang-8 Learner Corpora*.[9] This dataset contains 334 379 multilingual entries written by 59 455 active users in 80 languages. The most common languages were English (237 843 entries), Japanese (185 991) and Mandarin (28 154).

In 2012 and 2013, Lang-8 additionally released *Lang-8 Corpus of Learner English*, *Lang-8 Corpus of Romanized Learner Japanese* and *Lang-8 Corpus of Learner Japanese* with newer data and in the English corpus also with tense/aspect annotation.

This dataset is to the best of our knowledge the biggest English dataset for grammar correction. However, due to the online user-generated entries, this dataset contains multiple non-corrected sentences as well as entries with erroneous corrections.

[9]`http://cl.naist.jp/nldata/lang-8/`

2.6.2 NUS Corpus of Learner English

NUS Corpus of Learner English (NUCLE) is a fully annotated corpus of learner English publicly available for research purposes since June 2011 and described in Dahlmeier et al. [2013]. It consists of 1414 essays written by non-native English speakers at the National University of Singapore. The corpora is much smaller than Lang-8, nevertheless, because the annotators were professional English instructors, it is much cleaner.

Error annotations are made on character level and for each grammatical error instance contain start and end character offsets, error type and correction string. These annotations are then together with original sentences saved in SGML format.

Documents	1 414
Sentences	59 871
Word tokens	1 220 257
Error annotations	46 597
# of word tokens per sentence	20.38
# of error annotations per 100 word tokens	3.82

Table 2.1: NUCLE corpus statistics.

As we can see in Table 2.1, the number of annotated grammatical errors is very sparse. Dahlmeier et al. [2013] analyzed the corpora and revealed that over 57 percent of all sentences have zero errors, almost 21 percent have exactly one error, circa 11 percent have exactly two errors, and only 11 percent of all sentences have more than two errors. Authors also analyzed types of errors and found out that the top five error categories (e.g. wrong collocation/idiom/preposition or local redundancies) cover over 57 percent of all error annotations.

The corpora became widely known when the CoNLL 2013 and 2014 shared tasks were published. NUCLE was stated to be official training data. To make it easier for the participating teams, NUCLE was further preprocessed, which included for example sentence and word segmentation and mapping of character level error annotations to word level. Since NUCLE does does not contain test set, another 50 essays were collected and annotated. These test essays are now also freely available.

2.6.3 CzeSL

CzeSL (Czech as a Second Language) is a first learner corpus of Czech. The building process of the corpus is best described by Hana et al. [2014]. Currently, the first phase of the building process is finished (May 2012). The corpus now consists of transcribed texts of CzeSL (native Czech pupils) and ROMi (pupils with Romani background), which all together make 2 million word tokens.

The texts and error annotations in the corpus are divided into four levels. The first level L-1 are the raw transcripts of essays. Next level L0 is a tokenized text. Level L1 contains text with word forms corrected in isolation. Finally, level L2 handles all other types of errors and therefore, contains whole corrected text. Out of 2 million word tokens that are present in the L-1 layer, 300 000 word tokens were manually annotated on level L1 and L2.

2.6.4 Prague Dependency Treebank

Prague Dependency Treebank (PDT) is a collection of annotated articles from Czech newspapers and journals. Annotations are made on three layers: morphological, analytical and tectogrammatical level. PDT3.0 [Bejček et al., 2013] is an up to date version of this corpus. The full version of this corpus consists of 115 844 sentences with 1 957 247 tokens, which are annotated on the morphological level. From that 87 913 sentences with 1 503 739 tokens are annotated also on the analytical level and 49 431 sentences with 833 195 tokens are annotated on all three levels. All distributions of this corpora contain an official train, dev (d-test) and test (e-test) splits.

2.6.5 SYN2010

SYN2010 [Křen et al., 2010] is a huge collection of Czech texts. The texts come from fiction literature (40 percent), technical literature (27 percent) and journalism (33 percent). There are 152 634 documents in the corpora consisting of 121 667 413 tokens and 8 172 649 sentences. The SYN2010 corpus is also lemmatized and morphologically tagged.

2.7 Analysis summary

In this chapter we discussed approaches to two main tasks in natural language correction: spelling correction and grammatical error correction. The best systems on grammatical error correction task are both statistical machine translation systems and neural machine translation systems. Therefore, one of our proposed model is a neural machine translation system. Besides this model, we propose two more, which may be suitable for another natural language correction tasks such as spelling correction. These models are described in the following chapters.

3. Char2char model

When thinking about tasks such as diacritization, uppercasing text or adding commas to a text, we realized that these tasks have something in common. We can think of all these tasks as of mapping input sequence of symbols to the same number of output labels. This is formally called sequence labeling problem and is discussed in Graves [2012] or Nguyen and Guo [2007] .

It is rather simple to imagine the labels for the diacritization and text uppercasing task, since the labels can be directly the correct symbols. For the adding comma case, it is no longer possible, because when there is a comma, the output sequence has naturally more characters. We can, nevertheless, define the task to generate *true*, if there is a comma right after this character and *false* otherwise.

The model described in this chapter is tailored for these tasks. It benefits from a simple symbol mapping, allowing such tasks to be trained relatively quickly and on big enough corpora also surprisingly accurately.

3.1 Model architecture

The idea behind the model is rather simple. We decided to use recurrent neural network (RNN), which should for each provided sentence symbol output its correct label. This proposal can be seen in Figure 3.1. It contains an already unfolded RNN for input sentence "Cau!" and correctly diacritized output "Čau!".

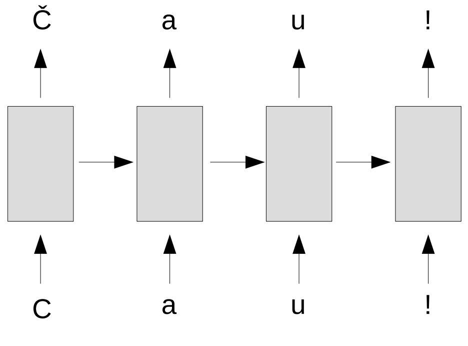


Figure 3.1: Char2char basic model

To formalize and extend the model, let us describe its components:

- Model inputs are individual characters, respectively a sequence of IDs with each ID being an index to an input character vocabulary. The input and output vocabulary contains a special token dedicated for unknown symbols.

- These characters are embedded, i.e. each character in the input sentence represented by a one-hot row vector is multiplied by a trainable matrix $W^{i \times d}$, where i is the size of the input alphabet and d is the embedding dimensionality. The reason for using distributed representation [Bengio et al., 2003] instead of one-hot-representation is that it allows the model to group together characters that are similar for the task. For example in the diacritization task, all non-letter symbols could be in a similar part of space, because diacritics is never added to them.

- These embeddings are fed to RNN, which is in our case bidirectional [Graves and Schmidhuber, 2005]. The bidirectional RNN consists of two unidirectional RNNs, one reading the inputs in standard order (forward RNN) and the other in reverse order (backward RNN). The output of bidirectional RNN is then a sum of forward and backward RNN outputs. In this way, bidirectional RNN is processing information from both preceding and following context.

- The model allows an arbitrary number of stacked bidirectional RNN layers.

- The output of the (possibly multilayer) bidirectional RNN is at each time step reduced by a same fully connected layer to an o-dimensional vector, where o is the size of the output alphabet. A non-linearity is then applied to these reduced vectors.

- Finally, we use softmax layer to produce probability distribution over output alphabet at each time step.

The described model with one RNN layer is illustrated in Figure 3.2.

3.2 Model training and inferring

We train the model by minimizing the negative log likelihood of the training data using stochastic gradient with adaptive learning rates.

Once we train a model, we want to employ it for correcting text. In other words, we are given an uncorrected sentence and we want to use model outputs to find the most probable correction.

The first step in correcting is, obviously, feeding an input sentence $c_1 ... c_N$ into the model. The model then, for each input character c_i, computes probability distribution over all output characters $p_i = (p_{i1} ... p_{iM})$. We can see that p_i depends only on values of $c_1 ... c_N$ and is independent on other p_j. Therefore, to get the most

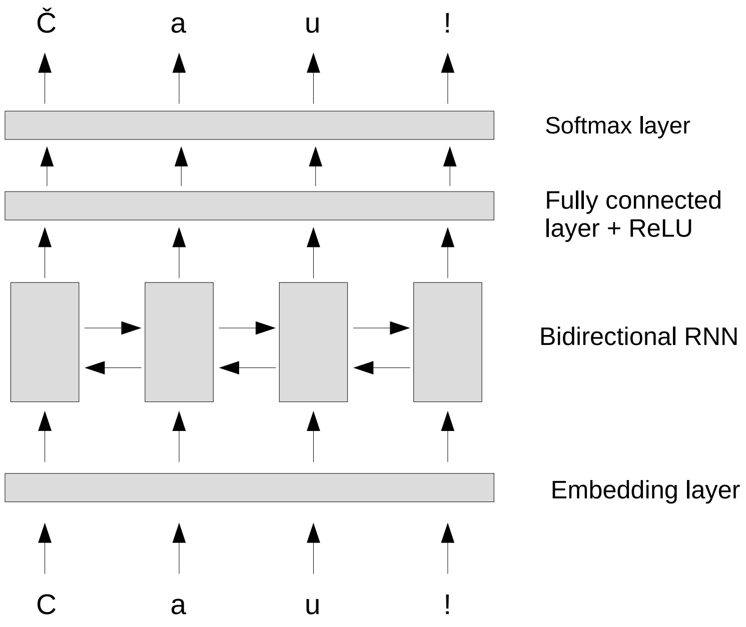


Figure 3.2: Char2char model in detail

probable model correction, we can simply take maximum output probability value for each input character. This process is described in Algorithm 1

```
**Data:** Input sentence consisting of individual characters $c_1, c_2, \dots, c_N$
**Result:** Corrected sentence consisting of individual characters
        $o_1, o_2, \dots, o_N$
$inp_1, inp_2, \dots, inp_N = \text{chars\_to\_input\_indices}(c_1, c_2, \dots, c_N)$
$p_1, p_2, \dots, p_N = \text{run\_model}(inp_1, inp_2, \dots, inp_N)$
$i = 1$
**while** $i \neq N + 1$ **do**
    $out\_index_i = \text{argmax}(p_i)$
    $o_i = \text{output\_index\_to\_char}(out\_index_i)$
    $i = i + 1$
**end**
return $o_1, o_2, \dots, o_N$
```

Algorithm 1: Using char2char model for correcting a sentence

A corrected sentence found by Algorithm 1 is the most probable correction in means of the model. Since the capacity and the learning capability of the model is limited, it may not learn to correct everything perfectly. For this reason, we decided to incorporate language model while inferring. As already mentioned in Section 2.3 and Section 2.4, most of the currently implemented spelling and grammatical error correcting programs use language models to improve the quality of their corrections. There is, therefore, great potential that the use of language model may improve quality of our results as well.

3.3 Language model

A model that computes probabilistic distribution over the set of all possible sentences of the given language is called a language model. The language model assigns low probability to rare occurring sentences and higher probability to more common sentences. From the natural language correction perspective, the sentences with spelling or grammatical errors should have lower probability than their corrected versions. The reason for this is the fact that the language models are trained on clean corpora, which do usually consists of correct text with minimal number of errors. Because the nowadays language models operate on word level, we often refer to them as word level language models.

There are basically two ways how to include these word level language model into char2char model inference. Both options work with a beam of fixed size of hypotheses. The hypothesis is in this context a tuple storing represented sequence of characters and its probability. To understand both approaches, it is neccesary to describe basic beam search first.

Extending notation from Section 3.2, let k denotes the beam size. Then the basic beam search algorithm works as follows. In the beginning, the beam is initialized with k most probable possibilities of the first character. In the next step, all hypotheses are extended with k most probable possibilities of the second character creating k^2 hypotheses. These are than sorted according to their probability and top k of them is kept for further iteration. The process runs till we reach an end of sentence. Finally we have top k most probable corrections. This beam search is more formally described in Algorithm 2.

```
Data: Input sentence consisting of individual characters $c_1, c_2, \dots, c_N$
Result: $k$ most probable corrections
$inp_1, inp_2, \dots, inp_N$ = chars_to_input_indices($c_1, c_2, \dots, c_N$)
$p_1, p_2, \dots, p_N$ = run_model($inp_1, inp_2, ...inp_N$)
$hyps$ = []
$top\_k\_indices$ = argsort($p_1$, highest_first)[:$k$]
for $j$ in $top\_k\_indices$ do
    $hyps$.append(Hypothesis(seq = output_index_to_char($j$),prob = $p_1[j]$))
end
$i = 2$
while $i \neq N + 1$ do
    $top\_k\_indices$ = argsort($p_i$, highest_first)[:$k$]
    $candidates$ = []
    for $hyp$ in $hyps$ do
        for $j$ in $top\_k\_indices$ do
            $candidate\_prob = hyp.prob \cdot p_i[j]$
            $candidate\_seq = hyp.seq + output\_index\_to\_char(j)$
            $candidates$.append(Hypothesis(seq = $candidate\_seq$,prob =
              $candidate\_prob$)
        end
    end
    (2)
    $hyps$ = sorted($candidates$,key = $candidate.prob$, highest_first)[:$k$]
    $i = i + 1$
end
(1)
return $hyps$
```

Algorithm 2: Beam search for finding k best corrections using char2char model

It is obvious that the most probable correction the beam search finds is the same correction as found by Algorithm 1. Nevertheless, there are two places denoted as (1) and (2) in Algorithm 2 dedicated to two mentioned approaches that incorporate language model.

The placeholder denoted as (1) is used by the first approach. It scores all sequences represented by the final hypotheses with the language model. These probabilities are then combined with stored hypotheses probabilities. The hypothesis with highest combined score is then returned.

```
(1):
for $hyp$ in $hyps$ do
    $lm\_score$ = lm.score($hyp.seq$)
    $hyp.prob = hyp.prob + \alpha \cdot lm\_score$
end
```

The second approach (2) incorporates language model each time a space in a candidate is generated.

```
(2):
for $candidate$ in $candidates$ do
    if $candidate.seq$[-1].isspace() then
        $lm\_score$ = lm.score($candidate.seq$)
        $candidate.prob = candidate.prob + \alpha \cdot lm\_score$
    end
end
```

Both approaches have their advantages and disadvantages. Incorporating language model too often with great α may lead to results that are grammatically correct, but do not correspond to input sentences. On the other hand, using language model only on the final hypotheses may help to correct only several last words. It is also not clear, whether using the second approach in cases where different candidate hypotheses contain different number of words does not make the beam search prefer sentences with less words to sentences with more words or vica versa.

We aim to use this model for tasks that do not change the number of words. We also believe that we can select α big enough to take into account the language model probabilities, but simultaneously small enough to reflect char2char model predictions. For this reasons, we decided to use the second approach and, thus, use language model after each space in an output sequence.

So far we supposed that the language model is already constructed. There are currently two approaches to building language models: statistical and neural network based.

The statistical approach uses chain rule of probability and Markov assumption to rewrite the probability of a whole sentence as a product of n-grams probabilities. The maximal value of n utilized by the language model is often called a rank of the language model. For a bigram language model (rank 2), the term expressing the sentence probability can be rewritten as follows:

$$P(s) = P(w_1, w_2, ...w_N) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_2) \cdot ... \cdot P(w_N|w_{N-1})$$

The conditional probabilities present in the above equation are then estimated from data using the maximum likelihood estimation and n-gram counts. This approach would, however, assign zero probability to sentences with at least one n-gram that was not present in the training corpus. For this reason, several smoothing methods (e.g. Good-Turing discounting method or Kneser-Ney discounting method) can be used to add some probability to unseen n-grams and take off some amount of probability from n-grams present in the training corpus.

The neural based approach trains a network to predict next word in a sentence. Mikolov et al. [2011] have shown that neural based language models outperform count based language models. One of the current best models [Kim et al., 2015, Jozefowicz et al., 2016] feeds word embeddings into reccurent neural network, which is trained to predict the next word. The innovative approach of this work is in construction of word embeddings. These are created from separate characters using convolutional neural network in combination with highway networks [Srivastava et al., 2015].

Even though the performance of neural based language models is higher than those of classic statistical language models, we decided to utilize statistical models in our work. The statistical models have been utilized for longer time and, thus, there exist tuned tools that are easy to train and infer. They can be more easily integrated and are much faster to train. Finally, despite the fact that they perform worse than neural based language models, they still achieve satisfactory results.

3.4 Residual connections

The proposed model allows an arbitrary number of stacked RNN layers. The model with multiple layers allows each stacked layer to process more complex representation of current input. This naturally brings potential to improve accuracy of the model.

As stated by Wu et al. [2016], simple stacking of more RNN layers works only to a certain number of layers. Beyond this limit, the model becomes too difficult to train, which is most likely caused by vanishing and exploding gradient problems [Pascanu et al., 2013]. To improve the gradient flow, Wu et al. [2016] incorporate residual connections to the model. To formalize this idea, let RNN_i be the i-th RNN layer in a stack and $x_0 = (inp_1, inp_2, \ldots, inp_N)$ input to the first stacked RNN layer RNN_0. The model we have proposed so far works as follows:

$$o_i, c_i = RNN_i(x_i)$$

$$x_{i+1} = o_i$$

$$o_{i+1}, c_{i+1} = RNN_{i+1}(x_{i+1}),$$

where o_i is the output of i-th stacked RNN layer and c_i is a sequence of its hidden states. The model with residual connections between stacked RNN layers then works as follows:

$$o_i, c_i = RNN_i(x_i)$$

$$x_{i+1} = o_i + x_i$$

$$o_{i+1}, c_{i+1} = RNN_{i+1}(x_{i+1})$$

The comparison of the classic stacked model and the stacked model with residual connections can be seen in Figure 3.3.

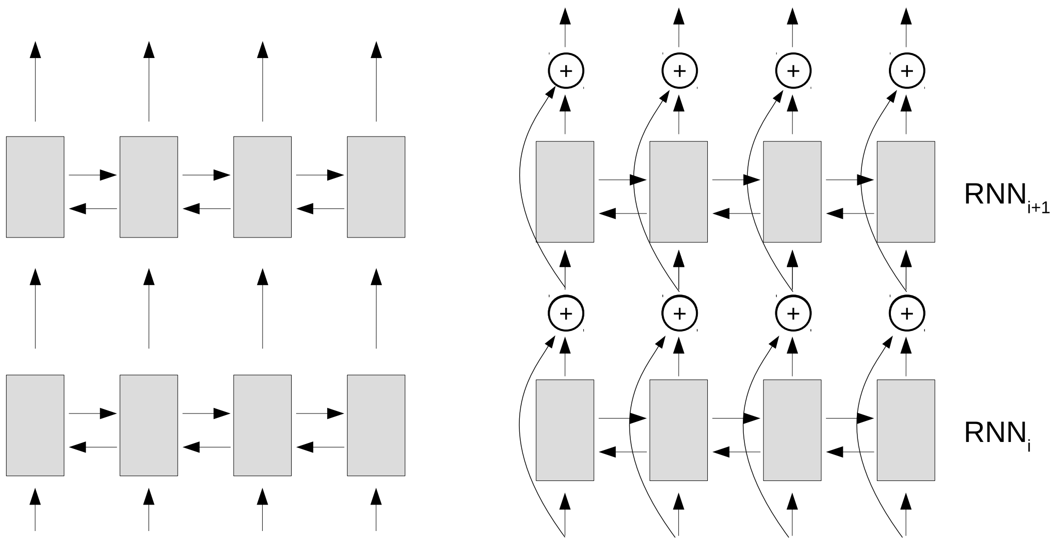


Figure 3.3: The difference between classic stacked RNN and stacked RNN with residual connections.

4. Word2word model

The first model we proposed was generating a label, typically a single character, for each character in the input sequence. This limits the usage of the model to only several tasks such as diacritization. The second model we propose is much more universal. It can be used for all tasks char2char model is applicable to and also for spelling correction and basic grammar correction.

When searching for errors in text, humans often work on a word level. They do not inspect the sentence character by character, but instead take whole word and check if it is correct given its context. This is the core idea behind word2word model.

4.1 Model architecture

We slightly change the architecture of char2char model to fit our current needs. The main computing component remains a multilayer recurrent neural network. Since its main purpose is to correct errors, we call it a correcting module. The inputs of the model are changed from individual characters to whole words. In a similar way, output vectors of the correcting module are projected to probabilistic distribution over all output words instead of all output characters. Although this may seem reasonable, there are two major issues in the proposed model design.

The first one is the representation of whole words that are to be fed to the correcting module. The reason, why the straightforward representation used in char2char model is no longer possible, is that there is basically no upper limit on the number of input words. The input words may contain spelling errors, be multidigit numbers, dates, emoticons or web addresses. Thus, both one-hot representation and distributed representation using a trainable matrix cannot be used. Fortunately, there are several approaches that are capable of creating word embedding even for these cases. We discuss them in Section 4.2.

Also the number of output words is too large; consequently, we cannot use a vector for probabilistic distribution over them. Instead, we incorporate a character decoder. The character decoder is basically a recurrent neural network, whose goal is to map a fixed size vector to a target sequence of characters. The initial state of the decoder is the output vector from the underlying correcting module. The way decoder works depends on whether it is being trained or we infer with it. During training at time t we feed the ground truth character y_{t-1} to the network to predict y_t, whereas at infer time, the most probable output y_{t-1} is usually used. To start decoding, we feed a special start-of-sequence token and similarly decoding is finished when the decoder outputs an end-of-sequence token.

The model visualization correcting the input sentence *Goot luck :)* is shown in Figure 4.1. Note that for simplicity, the correcting module contains only one bidirectional layer and the decoding mechanism is shown only for the first word.

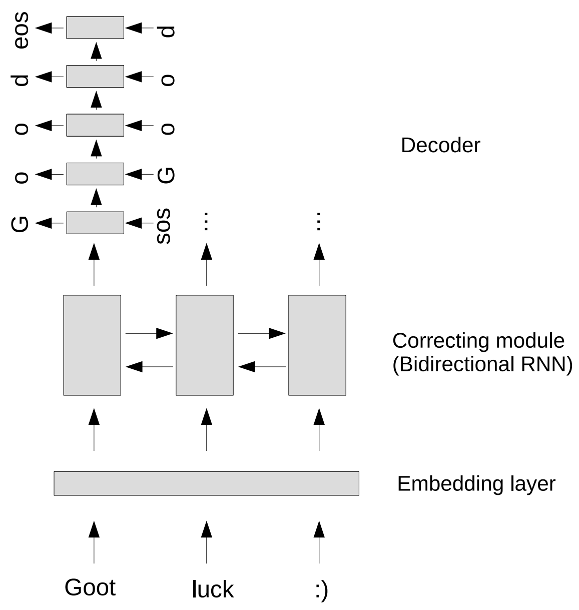


Figure 4.1: Word2word model visualization on an erroneous sentence

4.2 Character level word embeddings

As mentioned in the previous section, we need to create word embeddings for a potentially infinite number of words. Because there is no upper limit on the number of words, we cannot use traditional approaches that create a vocabulary of words and learn an embedding for each word separately. Instead, word embeddings are created from the word parts, mostly its characters. There are currently three main approaches to this issue and we discuss them here. It is worth noting that all mentioned models are composed of trainable parts and are either initialized randomly or pretrained on another task. If the trainable parts are initialized randomly, it may take some time until the word embeddings start helping the model, nevertheless, when trained, words that are similar in the task should be located in the same part of the space.

4.2.1 C2W model

Probably the most straightforward approach that utilizes recurrent neural networks is C2W model proposed by Ling et al. [2015]. It consists of two parts:

character lookup table and a recurrent neural network. The first step in the computation of a word embedding is to decompose the word into individual characters. Each character is then embedded using trainable character lookup table. Such embedded characters are then fed to a bidirectional recurrent neural network composed of LSTM units [Hochreiter and Schmidhuber, 1997]. Finally, the embedding of the word is obtained by combining the last forward and the last backward outputs of the RNN. Whole process is presented in Figure 4.2. Note that for greater clarity, the bidirectional recurrent network is shown in detail, i.e. both forward and backward reccurent neural networks are shown.

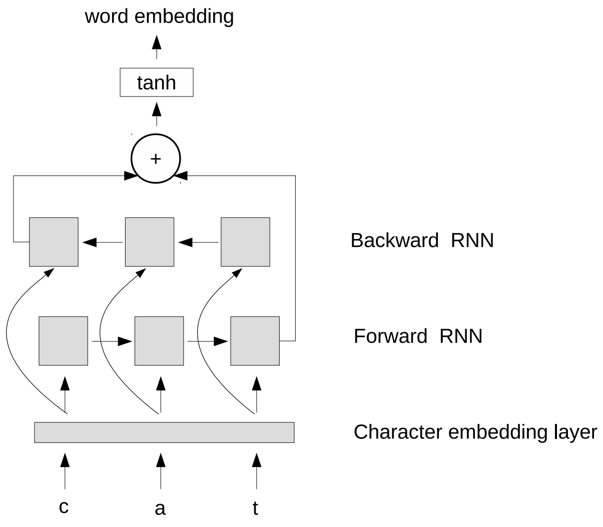


Figure 4.2: C2W model for computing character level word embeddings

4.2.2 CharCNN

The second approach (Kim et al. [2015]) computes word embeddings by applying a fixed set of convolution filters (kernels) to a sequence of embedded characters. Depending on its width, each filter detects presence of some n-gram feature. Since a result of the convolution is a vector whose size depends on the number of input characters, a max pooling layer is used to generate a fixed size embedding. In this way, each item of embedding vector determines the maximum presence of some n-gram feature in the input word.

More formally, let α and β be two characters that are not present in any word and let $H = H_1, H_2, \ldots, H_h$ be a fixed set of filters of possibly varying width. A filter H_i with width w_i is then of dimension $\mathbb{R}^{d \times w_i}$, where d is a dimension of character embeddings. These character embeddings are stored in a character lookup table, which contains embedding for each character in the set of input words as well as for special α and β symbols. The process of embedding computation for a single word w is then described in Algorithm 3.

```
Data: Input word consisting of individual characters $w = c_1, c_2, \dots, c_N$
Result: Embedding of word w: $e_1, e_2, \dots, e_h$
$C = ce_1, ce_2, \dots, ce_{N+2}$ = embed_characters($\alpha, c_1, c_2, \dots, c_N, \beta$)
for $i$ in range(h) + 1 do
    $features$ = tanh(apply_convolution($C, h_i$, stride = 1)) + $b$
    $e_i = \max(features)$
end
return $e_1, e_2, \dots, e_h$
```

Algorithm 3: CharCNN algorithm for computing character level word embeddings

4.2.3 Charagram model

The last approach we describe is Charagram model proposed by Wieting et al. [2016] and independently under different names also by Bojanowski et al. [2016] and Kocmi and Bojar [2016]. In comparison to previous models, all this model needs is one lookup table. This lookup table, however, does not store embeddings only for individual characters, but for the most occurring character n-grams instead. To compute an embedding of a word, we embed those character n-grams of a word that are stored in the lookup table, sum these embeddings and finally use a non-linearity (e.g. tanh or ReLU) on the result of the addition.[1]

The performance of this method clearly depends on the lookup table. Therefore, it is an important question which character n-grams to store in it. In the original paper, authors selected top k most occurring character n-grams for n=1,2,3,4. They experimented with $k \in \{100, 1000, 50000\}$ and claim that the best performance on selected tasks (part-of-speech tagging and semantic similarity) was achieved with the biggest k. Nevertheless, they also mention that for part-of-speech tagging task selecting only 100 n-grams of each order yields surprisingly good results.

The paper also contains comparison of the proposed model with C2W model and CharCNN. The authors found out that Charagram model, which has the simplest architecture, converges fastest to high performance.

[1]Note that some papers implement averaging instead of adding.

4.3 Attention mechanism

In the previous section we discussed multiple ways of computing word embedding on a character level. These embeddings are fed into the correcting module, whose goal is to produce outputs for the decoder. Each output is then processed by the decoder and a correction is generated character by character. Since the generated words may be quite long and the only information the decoder works with is the output vector of the recurrent neural network, the decoder might not learn to generate all words, especially the longer ones, properly. The similar issue had been solved in sequence-to-sequence models for machine translation task.

The goal of the machine translation task is to translate an input sentence in one language to a sentence in another language. The neural based approach utilizes sequence-to-sequence model [Sutskever et al., 2014, Cho et al., 2014], which consists of an encoder and a decoder. The encoder feeds word IDs of a sentence into a C2W model (see Section 4.2.1), which produces the sentence embedding. This is then used by the decoder, which generates translation word by word.[2] Bahdanau et al. [2014] conjectured that the bottleneck of the sequence-to-sequence model is the use of a fixed size encoder output vector and proposed an extension to it. This extension allows the decoder to "attend" to different parts of the input sentence at each step of the output generation and use this additional information while decoding. The decision what part of the input sentence to look at is learnt and is based both on the input sentence and the currently generated output.

Instead of formalizing the attention mechanism for sequence-to-sequence models, let us directly discuss its incorporation in our model. For spelling correction task as well as diacritization and simpler grammar correction, the words generated by the model are quite similar to words that were fed to the model at the corresponding places. While generating an output word, the decoder could, therefore, benefit from knowing what characters were fed to the model. For this reason, we incorporate the attention mechanism in our model.

Let $w_i = (w_{i1}, w_{i2}, \dots, w_{iN})$ be the i-th word of the input sentence and $s_{i,0}$ be the i-th output of the correcting module of our model. The decoder we used so far generated output based on only this vector $s_{i,0}$. The decoder with attention mechanism utilizes at each step of generation also a context vector. To explain, let us futher consider that the decoder just outputted $y_{i,j-1}$ and its hidden state is $s_{i,j-1}$. To generate next character, the decoder combines information both from

[2]Note that recently proposed model [Lee et al., 2016] operates both in the encoder and in the decoder on a character level, but this makes the described problem even more difficult.

$s_{i,j-1}$, $y_{i,j-1}$ and a computed context vector $c_{i,j}$:

$$P(y_j|y_{i,0}, y_{i,1}, ...y_{i,j-1}, w_i) = f(y_{i,j-1}, s_{i,j-1}, c_{i,j})$$

The context vector $c_{i,j}$ basically tells the decoder to which parts of input to "attend". It depends on a sequence of annotations $h_i = (h_{i0}, h_{i1}, ...h_{iN})$ to which an encoder maps the i-th word. To create these annotations, we have one more recurrent neural network to which we feed w_i and obtain h_i as its outputs. Note that when use *C2W model* for word embedding computation, we can utilize it also for creating the annotations. The context vector $c_{i,j}$ is then computed as a weighted sum of h_i:

$$c_{i,j} = \sum_{k=1}^{N} \alpha_{ijk} \cdot h_{i,k}$$

The weight α_{ijk} of each annotation $c_{i,k}$ is computed as follows:

$$\alpha_{ijk} = \text{softmax}(e_{ijk}) = \frac{\exp(e_{ijk})}{\sum\limits_{o=1}^{N} \exp(e_{ijo})},$$

where e_{ijk} basically tells how relevant is the annotation h_{ik} for current state $s_{i,j-1}$. It is computed using a feed forward neural network followed by a non-linearity and linear projection to map it to a single number:

$$e_{ijk} = g(s_{i,j-1}, h_{ik}) = u^T \cdot \tanh(V s_{i,j-1} + W h_{ik})$$

4.4 Error classifier

The proposed model is using decoder for each output of the correcting module. This approach might have two disadvantages. Firstly, the decoder is learnt both to correct words with errors and to copy words that do not contain errors. Doing both simultaneously may be too difficult for the model. Secondly, the attention mechanism proposed in the previous section slows down both training and inference. This also means that if we could sometimes omit the decoding process, we could make the model faster.

To face these challenges, we come up with one more extension to the model. On top of the correcting module, we stack an error classifier. Its goal is to determine, whether the corresponding word contains a mistake and, thus, we should use the decoder to correct it, or if there is no mistake and we can just copy the input word to the output. The error classifier is a simple fully connected neural network with two outputs and a softmax layer. It produces a probabilistic distribution over an error existence in the input word.

With this extension, the decoder is learnt to only correct erroneous words, which may result in more precise corrections. Also, if the input text contains only a few errors, the model may run substantially faster. Finally, since the error classifier outputs probabilities, we could provide the user with this additional information, i.e. the user would know, how probable is that the specified word contains error.

The usage of the error classifier has also its disadvantages. The major one is that it makes the error classifier the performance bottleneck of the model. If the error classifier misclassified word as correct, the decoder never corrects it. Similarly, if the decoder misclassified word to contain error, the decoder will most probably generate a wrong output, because it was trained to only correct errors.

Because it is unclear, whether the usage of the error classifier may actually help the model perform better, we leave this extension optional and evaluate it in the experiments.

4.5 Model training and decoding

We train the model by minimizing the negative log likelihood of the training data using stochastic gradient with adaptive learning rates.

To infer with the model, we use a modified version of a beam search decoder. The classical beam search decoder as described in Algorithm 4 works with one encoder output vector from which it decodes whole correction (sentence). While decoding, it may also use the attention mechanism. The beam search decoding algorithm utilizes a set of hypotheses. A hypothesis in this context is a tuple storing a represented character sequence, its probability and the hidden state of the decoder after outputting last character from the represented sequence.

The encoder in word2word model is the correcting module. However, it does not output a single vector, but multiple vectors. To solve this issue, we run the beam search decoder multiple times. Firstly, we run the beam search decoder with the first output vector of the correcting module and an empty set of initial hypotheses. This gives us k most probable corrections *hyps* of the first word. After this, the language model may be incorporated to modify the probability of each correction. Afterwards, we run the beam search decoder with the second output vector of the correcting module and provide also a set of initial hypotheses *hyps*. This process is then iteratively applied on all correcting module output vectors together with optional language model incorporation. In the end, we have k most probable corrections of input sentence, from which we take the most probable one. This process is more formally described in Algorithm 5.

```
Data: encoder output $o = (o_0, \dots, o_d)$, input annotations
      $h = (h_1, \dots, h_N)$, initial hypotheses $hyps$
Result: $k$ most probable hypotheses
if $hyps$ == None then
    $hyps$ = [Hypothesis(seq=['GO'], prob = 1, state = $o$)]
end
$results = []$
$i = 0$
while $i \neq decoder\_max\_step$ and len($results$) $< k$ do
    $candidates = []$
    for $hyp$ in $hyps$ do
        $new\_state, output\_probs$ = decoder_step($hyp.state, h, hyp.seq[-1]$)
        $top\_k\_indices$ = argsort($output\_probs$, highest_first)[:$k$]
        for $j$ in $top\_k\_indices$ do
            $candidate\_prob = hyp.prob \cdot output\_probs[j]$
            $candidate\_seq = hyp.seq + output\_index\_to\_char(j)$
            $candidates$.append(Hypothesis(seq = $candidate\_seq$,prob =
              $candidate\_prob$,state=$new\_state$)
        end
    end
    $hyps = []$
    for $cand$ in sorted($candidates$, key = $candidate.prob$, highest_first) do
        if $cand.seq[-1]$ == 'EOS' then
            $results$.append($cand$)
        else
            $hyps$.append($cand$)
        end
        if len($hyps$) == $k$ or len($results$) == k then
            break;
        end
    end
    $i = i + 1$
end
if $i$ == $decoder\_max\_step$ then
    $results$.extend($hyps$)
end
return sorted($results$, key = $result.prob$, highest_first)[:$k$]
```

Algorithm 4: Beam search decoding algorithm

```
Data: Input sequence of words $w = (w_1, w_2, \dots, w_n)$
Result: Most probable hypothesis
$o_1, o_2, \dots, o_n$ = compute_correcting_module_outputs($w$)
$hyps$ = beam_search_decoder($o_1$, annotations($w$), None)
$hyps$ =apply_language_model($hyps$)
for $i$ in range($n + 1$, start = 1) do
    $hyps$ = beam_search_decoder($o_i$, annotations($w$), $hyps$)
    $hyps$ = apply_language_model($hyps$)
end
return sorted($hyps$, key=$hyp.prob$, highest_first)[0]
```

Algorithm 5: Using word2word model for correcting a sentence

4.6 Error generation

The obvious usage of the proposed model is to correct errors. Nevertheless, in some cases when there is for example not enough data to train the model properly, we could benefit from being able to generate errors instead. The model that is able to generate artificial errors may be used for instance to create bigger training dataset from clean data. Such dataset can be then used either by the same or more complex model to learn to correct errors better.

To train the model to generate errors, we first need to change the data. The original input and gold data are swapped so that the input data are the corrected sentences and the gold data contain errors. The model for the error generation task also contains an error classifier. The training then works in a similar way as described before.

The usage of the error classifier has two purposes. Firstly, because we assume there is a few data, we want the decoder to learn only to produce errors. Secondly, we want to be able to regulate the number of words with errors together with their quality in the output. To clarify, think of what the error classifier is learnt in the error generation task. It is learnt to tell the likelihood that the person would type the particular word incorrectly. At infer time, we could use this information and run the decoder only on words with high mistyping probability. It is also worth mentioning that in comparison to correcting task, the fact that the decoder did not learn to produce the gold data properly may not be that crucial.

4.7 Model summary

To summarize, the second proposed model is trying to mimic the way the people correct errors. It looks on whole words and generates a correction for each input word. To be able to feed whole words to the recurrent neural network, the character level word embeddings are created. We described three main approaches to this task, however, we decided to implement only C2W model (see Section 4.2.1) and Charagram model (see Section 4.2.3). The implementation of C2W model was straightforward, because we could reuse the main part used by the attention mechanism (see Section 4.3). The reason for preferring Charagram model over CharCNN was that its authors claim that it is much faster to train and also its results are comparable. To infer with the model, the beam search decoder (see Algorithm 4 and Algorithm 5) is used.

Apart from the model standard usage, we also proposed a way to train the model to produce errors in Section 4.6. This may have various purposes, but we conjecture that it could be used to create bigger dataset for more complex models. Even though all proposed models may be used for error generation, we suppose that due to the existence of an error classifier (see Section 4.4), this model is the right one if we want to generate the mistakes in words themselves.

The last thing to discuss is what tasks is the model capable of solving. It is clear that all tasks performed by char2char model such as text diacritization or text uppercasing can be done by this model as well. The model can also handle spelling correction tasks, which have the same number of words both in the input and the output sentence. This brings up the question whether the model can perform tasks that either swap, delete or create new words. It is clear that the deletion operation is possible, because the decoder can simply generate an empty word. The swap operation is also possible, but it seems intuitive to turn off the attention mechanism if there are many of these operations. Finally, to generate more words than there are in an input sentence, we can use a simple trick. Its main idea is to concatenate some adjacent words with a special symbol (e.g. tabulator). The decoder then for some input word outputs correction that is a string with possibly multiple words separated with the special symbol. With this trick, the model may theoretically learn to correct even the difficult grammar correction errors. Practically, we suspect the main area of tasks the word2word model will be used for contains spelling correction and simple grammar correction tasks.

5. Translation model

The last model we implemented is the most universal one. It does not require the data to have any special properties such as the same number of characters or words in both the input and the gold sentences. The model is capable of solving all possible natural language correction tasks.

As described in Section 2.4, the current state-of-the-art results in grammatical error correction are achieved both by statistical machine translation systems, neural machine translation systems or with their combination. For this reason, our third model is a neural based translation system. It adopts several features from the former state-of-the-art model of Xie et al. [2016].

5.1 Neural machine translation

Since 2014, the neural based approach to machine translation have been utilizing sequence-to-sequence model [Sutskever et al., 2014, Cho et al., 2014], which consists of an encoder and a decoder. The encoder is a bidirectional recurrent network, whose goal is to map input sentence to a fixed d-dimensional vector. This is then passed to the decoder, which is a recurrent network that generates a translation. The obvious bottleneck of the model is the fact that the decoder utilizes only one fixed size vector. Bahdanau et al. [2014] proposed an attention mechanism (see Section 4.3), which allows the decoder to use additional information from the input.

The encoder and the decoder originally operated on the word level, i.e. the inputs and the outputs to the model were individual words, respectively their indices. The indices are pointers to a fixed size vocabulary that contains certain number (e.g. 100k) of most common words. The straightforward usage of such model on natural language correction tasks may not work due to this fixed vocabulary. The input words for natural language correction task usually contain both spelling errors, dates, email addresses or different types of emoticons. Most of these words is not saved in the vocabulary and, thus, are fed to the encoder as out-of-vocabulary words. It is clear that the encoder may not perform satisfactory when fed with many out-of-vocabulary words.

In 2016, Lee et al. [2016] used both character level encoder and decoder for machine translation and Xie et al. [2016] for natural language correction. The issue they both had to cope with was that the input is much longer when measured in characters than when measured in words. It is worth noting that the sequence-to-sequence model is quite deep and, thus, slow to train, even if fed with words.

The attention mechanism overhead is also not so small and its computational cost grows quadratically with respect to the number of input annotations. Therefore, when one suddenly starts operating on a character level, both the training process and inferring process become slower. For this reason, authors incorporated a mechanism that reduces the number of input annotations.

An encoder of Lee et al. [2016] applies a combination of convolutional layers, max-pooling layers and highway networks on the input sentence, which results in a shorter sequence. This sequence is then fed to a standard bidirectional recurrent network which produces input annotations and a decoder initial state. An encoder of Xie et al. [2016] is a pyramidal encoder proposed by Chan et al. [2015]. It is composed of a stack of bidirectional layers, where each layer processes input of half a size of the previous layer output. More formally, let $x = (x_1, x_2, \ldots, x_n)$ be input sequence of character IDs. The forward, backward and combined (output) i-th layer encoder activations are computed as follows:

$$f_t^i = \mathrm{GRU}(f_{t-1}^i, h_t^{i-1})$$

$$b_t^i = \mathrm{GRU}(b_{t+1}^i, h_t^{i-1})$$

$$o_t^i = f_t^i + b_t^i,$$

where GRU is the gated recurrent unit [Cho et al., 2014], h denotes input from the previous layer, $h_t^0 = x_t$ and for $i > 0$ we have

$$h_t^i = \tanh(W^i[o_{2t}^{i-1}, o_{2t+1}^{i-1}] + b)$$

with matrix W^i reducing number of previous layer outputs by half.

5.2 Model architecture

Our implementation of neural machine translation system utilizes sequence-to-sequence model with the attention mechanism. As discussed in the previous section, because inputs may contain misspellings and other types of rare words, we decided to operate on the character level instead of the word level. With this decision, we had to also think over, whether to keep the standard encoder or implement one of two improvements described in the previous section. Since Xie et al. [2016] claim that their implementation is two times faster than the standard encoder when using three pyramidal layers in the encoder and also achieves good results on grammatical error correction tasks, we decided to implement their encoder. To evaluate the performance boost, we also implemented a standard multilayer encoder. Despite our decision, we think that the implementation of the encoder of Lee et al. [2016] may be a good option to try in the future work.

To sum up, the model consists of the pyramidal encoder, which produces the initial hidden state for the decoder and the input annotations (vector h). The decoder is a multilayer recurrent neural network that at each time step generates next character based on both its hidden state and the computed attention vector. As illustrated in Figure 5.1, the attention vector computed from the previous hidden state and the input annotations h is used twice at each time step, which may help the decoder when copying characters.

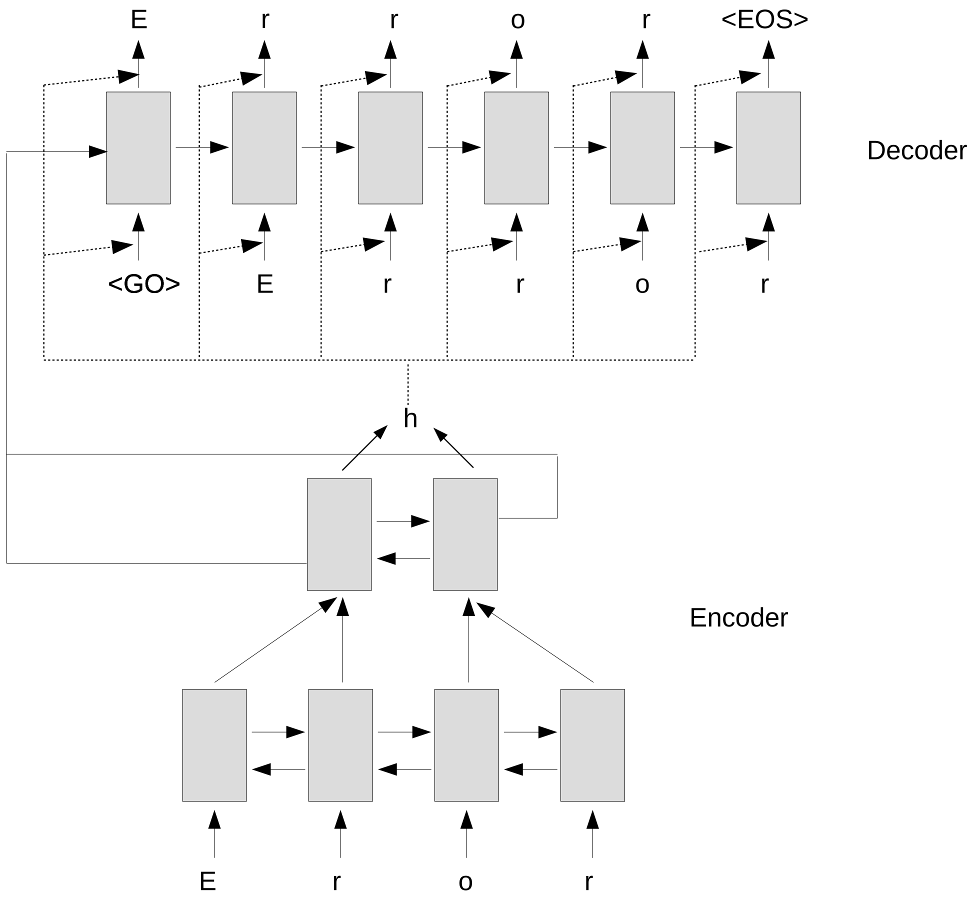


Figure 5.1: Translation model visualization with two encoder stacked layers – note that the attention is shown with a dashed line

We train the model by minimizing the negative log likelihood of the training data using stochastic gradient with adaptive learning rates.

To infer with the model, we first run the encoder to compute the initial decoder hidden state and input annotations. To decode, we utilize a beam search decoder, which we provide with the decoder initial state, input annotations and an empty set of initial hypothesis. The beam search algorithm works exactly as described in Algorithm 4.

5.3 Models summary

So far, we proposed three models. They differ in their complexity and range of tasks they are applicable to. The first model is designed for tasks that map each input character into a label. An example of such task may be diacritization or uppercasing task. The supposed usage of the second model is spelling and basic grammar correction area. The model is trying to mimic people, who when solving these tasks, usually look on whole words and check, whether they are correct in their context. Finally the third model is the most general model that is applicable to any task, for which we have a corpus of parallel data. One might now wonder why should we have three models when the translation model is capable of solving any task. This is theoretically true, nevertheless, the translation system is the most complex. It contains the most parameters and also its attention mechanism is trained to align to specific segments of a whole sentence. On the other hand, char2char model has no explicit attention mechanism and word2words model learns to align only to characters in given words. For this reason, we suppose the smaller models may be trained better to their tasks, which we evaluate in the next chapter.

When we compare approach used by our models with those used by current spelling correctors (see Section 2.3), we can notice one important difference. The statistical spelling correctors usually consist of a mechanism that generates corrections to consider and of an error model that assigns probabilities to them. Both these modules are created independently. A candidate generation mechanism proposes candidates, which are not beyond some edit distance, and an error model is somehow computed from available data. However, even if the task is as simple as diacritization, there may be too many edits that are needed to transform the input word to a correct one; therefore, the correct word is not even proposed by candidate generator. Also the way to estimate the error model is anything, but easy. On the other hand, our models learn all mentioned components from training data when trying to maximize negative log likelihood of the training data. Moreover, our models are language independent in comparison to standard spelling correctors. In other words, when we want to make a diacritization system for instance for Arabic, all we need to do is to train the model with new data. In traditional diacritization systems, both the candidate generation and error model must be modified, which requires certain amount of non-trivial work.

On a theoretical level, all the advantages of neural based approach seem great. Now it is time to find out, whether the proposed models can achieve comparable performance with the traditional statistical systems.

6. Experiments

In this chapter we perform experiments and present their results. We start by detailing chosen tasks and describing their data. Then we define evaluation metrics, perform experiments and discuss their results. Finally, for each task, we compare achieved results with other tools.

6.1 Tasks and data

6.1.1 Diacritization

As already mentioned in Section 2.1, many Czech texts partially or even completely lack diacritics. This is a very motivating and, due to existence of large and clean Czech corpora, also solvable task. The creation of a dataset naturally brings up the question, whether input data for the diacritization should be completely without diacritics or whether diacritics should be occasionally preserved. The diacritization of the text with occasionally preserved diacritics seems to be an easier task, since we can always remove all diacritics and convert it to the text without diacritics. This was together with the fact that even if a text has some diacritics, it does not need to be correct, the reason, why we decided to keep all training data completely without diacritics.

The corpora we utilized for this task are PDT3.0 (see Section 2.6.4) and SYN2010 (see Section 2.6.5). PDT3.0 already contains train/dev/test sentence split, which we preserved in our task, therefore, we evaluate all our experiments on PDT3.0 test set with removed diacritics. For training purposes only, we utilized also SYN2010 corpora, which is significantly bigger. Basic characteristics of both datasets can be found in Table 6.1.

Name	# sentences	# words	% word errors
PDT3.0 train set	90 828	1 535 826	42.5
PDT3.0 dev set	11 880	201 651	42.8
PDT3.0 test set	13 136	219 765	42.4
SYN2010	8 172 649	129 847 673	40.0

Table 6.1: Basic statistics of diacritization dataset

6.1.2 I vs Y

One of the most common errors in Czech texts is interchanging *i* and *y*. Even though it is not very probable that a text would contain only these errors in separate, i.e. no other error types would be present in the text, it could still be useful to be able to correct such texts. To create dataset for this task, we can use process similar to the creation of the diacritization dataset. Instead of replacing all diacritized characters by their ASCII counterparts, all occurrences of letters *yYýÝ* are replaced by letters *iIíÍ*. The described process is applied to sentences from PDT3.0 train/dev/test sets. The official PDT3.0 train/dev/test sentences are then used as the gold data.

6.1.3 Czech spelling and basic grammar correction

The third group of experiments we conducted is devoted to the spelling and basic grammar correction for Czech. The reason for choosing Czech is that there exist two datasets we can directly work with: CzeSL (see Section 2.6.3) and Švejk, which is a manual transcript of an audiobook performed by authors of Korektor (see Section 2.3.2).

CzeSL

CzeSL contains 20 752, respectively 13 267 parallel pair of original and corrected sentences annotated in two annotator rounds (A1 and A2). All sentences are word tokenized. The variety of error types in the dataset is large. We want, however, to handle only spelling and basic grammar error types in this task. For this reason we postprocessed original CzeSL corpus and created new four datasets with varying difficulty.

- czesl-sent2sent – Original data (sentence aligned).
- czesl-word2words – Subset of czesl-sent2sent. Each sentence is word aligned. A gold word may be empty or contain tabulators to separate multiple logical words.
- czesl-word2word – Subset of czesl-sent2sent. Each sentence is word aligned. A gold word contains exactly one non-empty word.
- czesl-word2simword – Subset of czesl-sent2sent. Each sentence is word aligned. A gold word contains exactly one non-empty word, which differs from the input word by no more than 50 percent in means of edit distance.

All four datasets are in two versions preserving the annotator rounds. To enable comparison of results, we also split all four datasets into training, development and testing sets. The development and testing input sentences were chosen to be same for both annotators. Consequently, the evaluation script may choose the gold correction which more resembles the system output. Note that for simplicity reasons, we evaluate our models only on the first annotator round testing set.

The basic statistics of the created datasets are presented in Table 6.2. To summarize them, each dataset has more training sentences annotated in the first annotator round (A1) than in the second annotator round (A2). In means of sentence counts, czesl-word2simword is similar to czesl-word2word and czesl-word2words is similar to czesl-sent2sent. Finally, all datasets contain quite a lot of word errors and the highest error proportion is at czesl-word2words dataset.

Dataset	train	dev	test	% word errors
czesl-sent2sent (A1)	18 233	1 191	1 198	–
czesl-sent2sent (A2)	10 748	1 191	1 198	–
czesl-word2words (A1)	18 264	1 199	1 169	[20.2, 17.4, 16.9]
czesl-word2words (A2)	10 804	1 199	1 169	[19.4, 17.2, 17.1]
czesl-word2word (A1)	11 493	760	700	[12.4, 10.0, 11.5]
czesl-word2word (A2)	6 859	760	700	[12.0, 10.3, 11.6]
czesl-word2simword (A1)	10 390	688	622	[10.2, 9.9, 9.8]
czesl-word2simword (A2)	6 230	688	622	[9.9, 9.8, 9.5]

Table 6.2: Basic statistics of four new datasets derived from CzeSL – the last column contains percentage of word errors in training, development and testing set of particular dataset does not have values for czesl-sent2sent, because this dataset is only sentence aligned.

Out of all four datasets, czesl-word2simword is the only dataset that may contain spelling and basic grammatical errors and, therefore, we decided to utilize it in our second experiments. The rest three datasets are devoted to grammatical error correction and we leave them for future research.

Švejk

Švejk is a manual transcript of an audio book. It consists of approximately 1 000 sentences of both original and corrected texts. Because of its short length, we decided to use sentences in Švejk only as a testing set. We preserved the way we handled CzeSL and created four testing sets of varying difficulty: svejk-sent2sent

(982 sentences), svejk-word2words (982 sentences), svejk-word2word (964 sentences), svejk-word2simword (964 sentences). Out of these four datasets, we used svejk-word2simword in our second set of experiments. Nevertheless, the sentence counts of the created datasets indicate that even the original text contains mostly spelling errors but not grammatical errors. It is also worth mentioning that the percentage of words with error is around 5 percent, which is significantly less than in CzeSL.

Automatically generated spelling correction corpus for Czech (Czech-SEC-AG)

Besides the texts themselves, Švejk authors also created a character error model. This model contains probabilities of producing an error with respect to four character operations:

- substitution[x][y] – probability of typing x when y was intended
- swap[xy] – probability of typing yx when xy was intended
- insertion[x][p][s] – probability of erroneous insertion of x between p and s
- deletion[x][p] – probability of erroneous deletion of x after p

We utilized the error model to create new dataset with artificial errors. Despite the fact that the described error model does not contain probabilities of changing character casing, we estimated these probabilities and incorporated them while generating the artificial errors. The new dataset is derived from PDT3.0 corpus and we call it Czech-SEC-AG. While generating new dataset, we tried to keep the percentage of words with errors similar to Švejk dataset. For training purposes only, we also created errors on SYN2010 corpus with substantially higher error percentage (approximately 20 percent).

Let us now summarize datasets that we use in the spelling and basic grammar correction task. Firstly, we processed CzeSL corpus and created new four datasets. These are available at `http://hdl.handle.net/11234/1-2143`. From these datasets, we use czesl-word2simword in this thesis. Secondly, Švejk dataset is used as a testing set. Thirdly, we created new dataset Czech-SEC-AG based on the statistical error model. This dataset is together with Švejk available at `http://hdl.handle.net/11234/1-2144`. Czech-SEC-AG is in the thesis utilized for testing and SYN2010 with errors for training.

6.1.4 English grammar correction

The last set of experiments we conducted is devoted to grammatical error correction. Probably the most known task in this area is CoNLL 2014 shared task (see Section 2.4), which we utilize. CoNLL 2014 shared task comes with preprocessed NUCLE corpus (see Section 2.6.2), but since it is relatively small, most teams train their models also on Lang8 corpus (see Section 2.6.1). To evaluate system performance, CoNLL 2014 test set is used.

6.2 Evaluation metrics

To compare system performance, one needs to define evaluation metrics to use. Let us first concentrate on tasks having the same number of spaces both in input and gold sentences. This condition is satisfied in the diacritization, I vs Y and the spelling correction tasks, since they correct errors in words only. The reason why we require the same number of spaces instead of the same number of words is that we would like to use the same metrics also for the grammar correction task. The grammar correction system may naturally produce more (or less) words than was in the input. Nevertheless, we can use the trick mentioned in word2word model (see Section 4.7) and separate new words with tabulator instead of space.

Having this limitation in mind, we can split both input, output and gold sentences on spaces and obtain input, output and gold lists of chunks. The evaluation of the system is then performed on these chunks.

The most common metric is *accuracy*. Accuracy is defined as a number of corresponding chunks that are same both in system and gold sentences, divided by total number of chunks:

$$accuracy = \sum_{s=1}^{\#(sentences)} \sum_{i=1}^{\#(sentences[s])} \frac{system[s][i] == gold[s][i]}{number\ of\ all\ chunks}$$

Note that for some task (e.g. diacritization), we could also measure accuracy on single characters. However, we did not find this metric intuitive.

Accuracy is the sufficient metric as long as the number of chunks that should be corrected is not much smaller than the number of chunks that should stay unchanged. This can be clearly illustrated on the spelling correction task. Let's suppose that every 10-th word is misspelled and should be corrected. We train a system with accuracy of 87 percent, which may seem quite good. Now consider a completely useless system that just copies its input to the output. Such system has accuracy of 90 percent. This is weird as the useless system has higher accuracy than the system that tries to correct misspelled words. For this reason, it is better to use other metrics for such unbalanced cases.

One of the most popular metrics capable of handling this issue is F_1*-score*. Let an edit be chunk with corresponding input chunk being different from it. Then we have a set of system edits (*fsystem*) and a set of gold edits (*fgold*), i.e. the set of chunks that the system changed and the set of chunks that should be changed. Let us further denote *fboth* a set of edits that the system made correctly. Then F_1-score is defined as follows:

$$P = \frac{|fboth|}{|fsystem|} \qquad (precision)$$

$$R = \frac{|fboth|}{|fgold|} \qquad (recall)$$

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

F_1-score is the harmonic mean of precision and recall. However, in the natural language correction scenario, precision is often emphasized more than recall. In other words, when for example a spelling corrector runs in production mode, it is very important that the proposed corrections are accurate. Omitting some corrections is usually not as bad as proposing a bad one. For this reason, $F_{0.5}$-score is often used:

$$F_{0.5} = \frac{(1 + 0.5^2) \cdot P \cdot R}{0.5^2 \cdot P + R}$$

As we can see, F-score requires a set of system and gold edits. To make the evaluation simpler, we required that the number of spaces in input and gold sentences are the same. This is naturally often impractical; therefore, there are more sophisticated methods that can extract these edit sets directly from sentence pairs. Out of all these methods, we would like to mention MaxMatch algorithm [Dahlmeier and Ng, 2012], which is used in CoNLL 2013 and 2014 shared tasks.

To sum up, we report $F_{0.5}$-score for all evaluated tasks and for the diacritization and the I vs Y task, we report accuracy as well.

6.3 Language model

While inferring, all three models have an option to incorporate a language model (see Section 3.3, Section 4.5 and Section 5.2). To train the language model as well as to run it, we used open-source KenLM toolkit.[1] The language model was trained on a concatenation of the following corpora:

- Czech part of W2C (`http://hdl.handle.net/11858/00-097C-0000-0022-6133-9`)

[1]`https://github.com/kpu/kenlm`

- CZES corpus (`http://hdl.handle.net/11858/00-097C-0000-0001-CCCF-C`)
- articles from CWC2011 (`http://hdl.handle.net/11858/00-097C-0000-0006-B847-6`)
- SYN 2005 (`http://hdl.handle.net/11858/00-097C-0000-0023-119E-8`)
- SYN 2006 PUB (`http://hdl.handle.net/11858/00-097C-0000-0023-1358-3`)
- SYN 2009 PUB (`http://hdl.handle.net/11858/00-097C-0000-0023-1359-1`)
- SYN 2010 (`http://hdl.handle.net/11858/00-097C-0000-0023-119F-6`)
- SYN 2013 PUB (`http://hdl.handle.net/11858/00-097C-0000-0023-3B09-4`)

We trained two language models of rank 3 and 5. For training, only those {2,3,4,5}-grams that occurred at least twice were considered.

It is worth noting that we also experimented with own implementation of neural based language model operating on individual characters, but its performance was rather poor. Therefore, this is the only mention of it in the thesis.

6.4 Diacritization

The first solved task is the diacritization. For this task, we trained all three models on SYN2010 corpus with removed diacritics. We discuss each model settings and results in separate and in the end compare their results with currently existing tools.

6.4.1 Char2char model

To train the char2char model, several hyperparameters must be set. For the initial set of experiments, we decided to choose hyperparameters described in Table 6.3. This table also contains domains of the chosen hyperpameters and the values we experimented with.

Parameter name	Domain	Searched domain
RNN cell type	choice	LSTM, GRU
RNN cell dimension	integer	25, 50, 75, 100, 150, 200, 300, 400
Character embedding dimension	integer	25, 50, 100, 150, 200
Number of RNN layers	integer	1, 2, 3
Learning rate	float	$10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$
Dropout keep probability	float	0.3, 0.5, 0.8, 1.0

Table 6.3: Char2char model hyperparameters searched domains

Out of all training hyperparameters, we need to explain how the dropout technique [Srivastava et al., 2014] is used at the model. Dropout in general helps to reduce overfitting of the model. While training, it zeros some outputs of the particular layer. This should force the next layer not to depend only on certain features of the current layer and, thus, create more complex representation. At infer time, the network is used standardly. For this reason, all weights are corrected when the training is finished. In our experiments, we use dropout on both character embeddings and each output of RNN layers. The values mentioned in Table 6.3 then tell the probability that the particular field of a character embedding or RNN layer output is kept.

We would ideally try each possible combination of the described hyperparameters to find the best model. However, because there are 4 800 possible combinations and it takes approximately two days to train even the smallest model on CPU, we experimented with each hyperparameter separately, i.e. we fixed all hyperparameters, but the selected one. With this approach, our experiments show effects of the specified hyperparameters rather than the best possible hyperparameters.

All models were trained with Adam optimizer [Kingma and Ba, 2014] with a batch size of 200 for approximately 3 days on a single CPU. Results of the experiments are presented in Figure 6.1. We discuss each experiment now.

The first hyperparameter we experimented with is a dimension of RNN unit. The basic intuition tells that a unit of greater dimensionality may capture more features than a unit with lower dimensionality and, thus, reach better performance. Because the amount of training data is relatively large, we could hope that the size of RNN unit may be quite large even if it processes single character, which itself may not provide much information. This hypothesis turned out to be true, the bigger the RNN cell dimension, the higher the accuracy. The obvious disadvantage of using model with bigger RNN dimension is the model memory size and substantially longer training time (the training time is almost quadratic with respect to RNN cell dimension).

The second hyperparameter is a number of RNN layers that are used in stack. Similarly to the RNN unit dimension, the more layers the model has, the more computing power it theoretically possess. It turned out that, in practice, this is not that simple. We can see that the model with 3 layers has lower accuracy than the model with 2 layers. We discuss the possible reason why this happens later.

The third hyperparameter is RNN cell type. The graph shows that LSTM cell performs slightly better than GRU cell. Despite this fact, we used GRU cell in all further experiments mainly because it is simpler and, thus, runs faster.

The fourth hyperparameter is learning rate. The experiment shows that

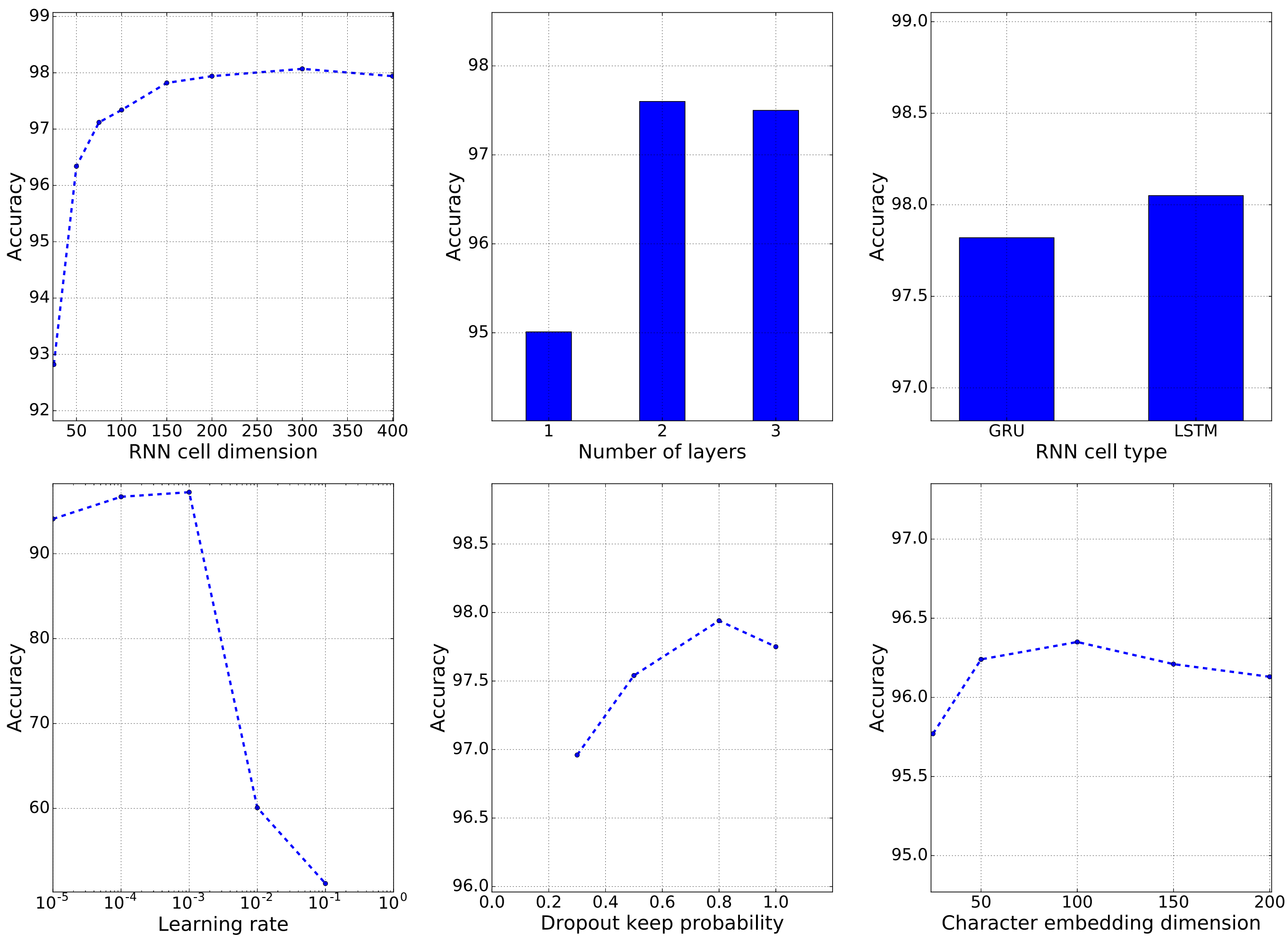


Figure 6.1: Results of basic experiments with char2char model on the diacritization task

the bad choice of this hyperparameter may lead to catastrophic results. The learning rate between 10^{-4} and 10^{-3} seems to be a good choice for the task.

The fifth hyperparameter is a keep probability of dropout. The choice of this hyperparameters does not seem to be that crucial as for example the learning rate. Nevertheless, the good choice of this hyperparameter definitely improves overall model performance.

The last hyperparameter we experimented with is a dimension of the character embedding. It is important to note that all models used in this last experiment had RNN units of size 100. Possibly due to this reason, the best accuracy is achieved with character embedding of size 100. Nevertheless, the model exhibits good performance even if the relatively small character embedding of size 50 is used.

Incorporating language model

As described in Section 3.3, incorporation of a language model to decoding may improve the diacritization accuracy. Therefore, we trained three char2char models with 1,2 and 3 stacked layers and also trained the language model of rank 5 as described in Section 6.3. The language model was then used when inferring with char2char models. We examined the impact of a language model weight α and a beam size on the diacritization accuracy. The results of the experiment are visualized in Figure 6.2.

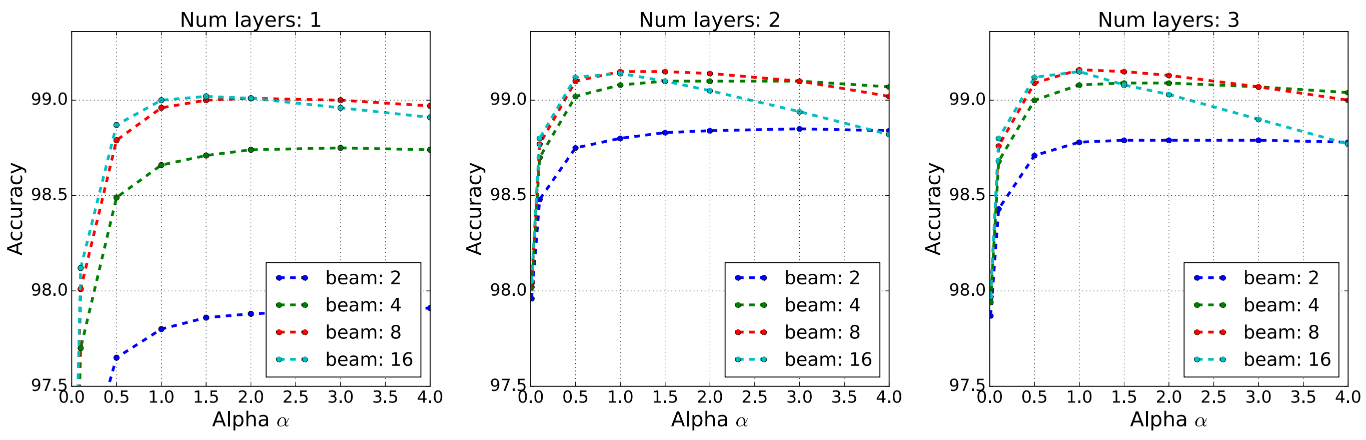


Figure 6.2: Char2char model diacritization – incorporating language model

The most important thing to notice is that the accuracy of all models greatly improves when the language model is utilized. In comparison to version without the language model, the best accuracy achieved by each model is higher by more than 1 percent.

Secondly, there is a great difference in the chosen beam size. The beam size of 2 achieves the worst results, the beam size of 4 achieves significantly better results and both the beam size of 8 and the beam size of 16 lead to the highest achieved accuracy.

Thirdly, the language model weight α is an important constant that needs to be estimated as well. The value of this constant that seems reasonable for the diacritization task is somewhere between 0.5 and 1.5.

Finally, even if the language model is incorporated, the model with 1 layer still performs significantly worse than two other models.

Language model rank

The language model used in the previous section has rank 5. This means that the language model was built from {1,2,3,4,5}-grams that occurred in the training corpus. In other words, the language model was trained to consider context of maximally 5 words. We were wondering, whether the usage of a less powerful language model would lead to worse results and if so, how much worse would they be.

To find out, we trained a language model with rank 3 and performed the similar experiment as described in the previous section. The comparison of results of both experiment is visualized in Figure 6.3. Note that the first row shows the experiment with the language model of rank 5 and the second row shows the results of the language model of rank 3.

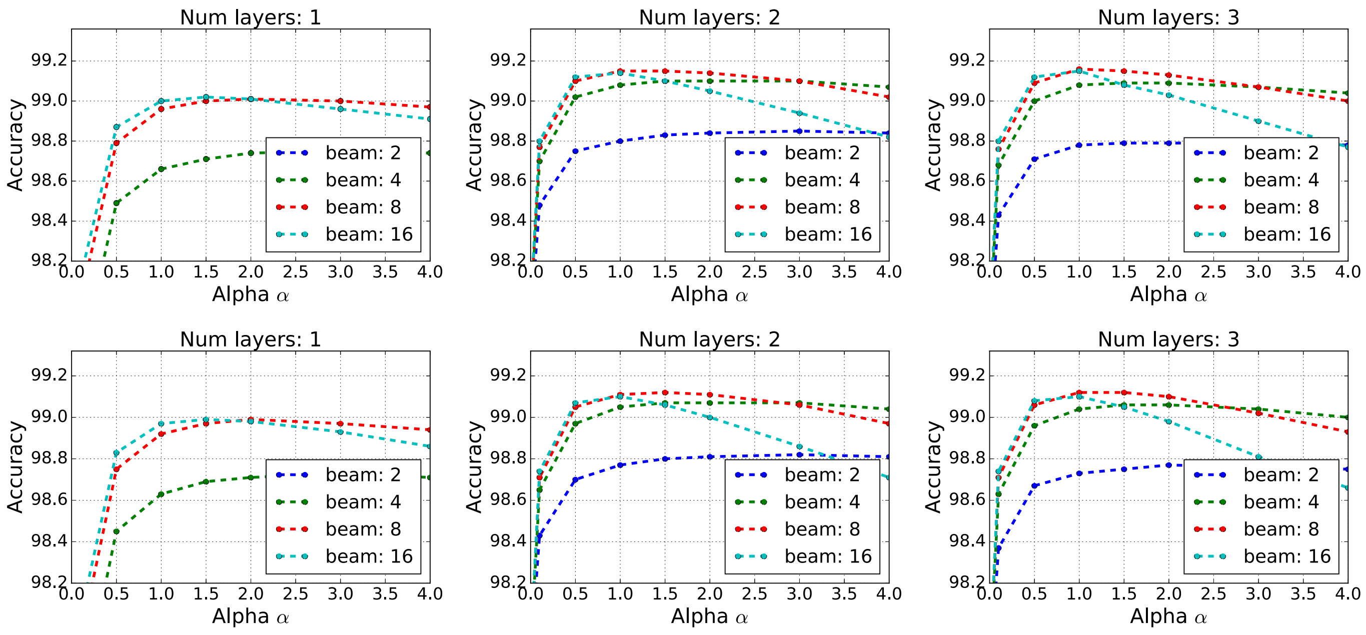


Figure 6.3: Char2char model diacritization – comparison of two language models with different rank.

The results of both experiments are visually almost identical. When inspecting the results more carefully, we found out that the experiments with the weaker language model achieved a slightly worse accuracy. Nevertheless, the accuracy was for the most of measured points not worse than 0.05 percent. From this we conclude that the language model rank does not have that important effect on the accuracy, which may be still great even for a smaller language model.

Residual connections

In all previous experiments, the model with 2 layers always performed the best. The reason why the model with more layers performed worse may be the vanishing or the exploding gradient problem. As mentioned in Section 3.4, after a certain number of stacked layers, the model becomes too deep and the backward gradient becomes either too small or too large, which leads to poorly trained models. To solve this issue, we suggested to add the residual connections to the model. In this experiment, we evaluate, whether the model with residual connections performs better even if it has more layers.

Because there is no sense in adding residual connections to the model with one layer, we added them to models with 2, 3 and 4 layers. The training is then performed as previously.

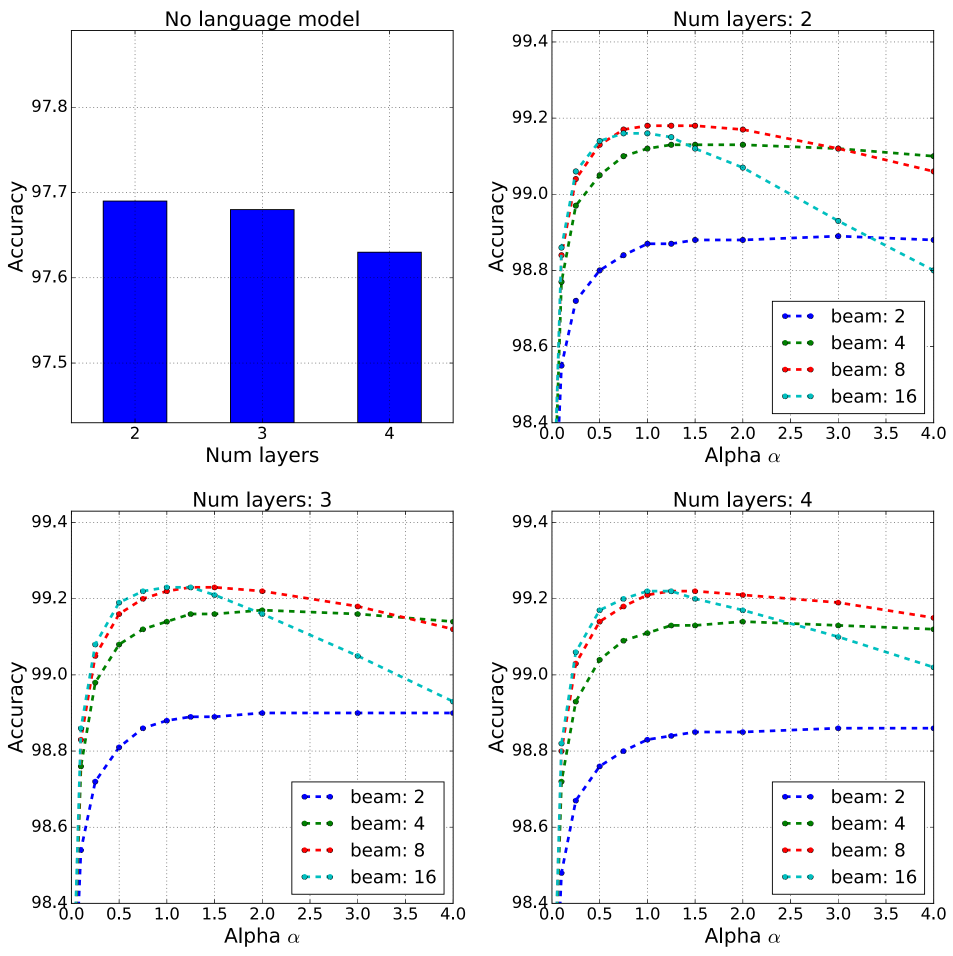


Figure 6.4: Char2char model diacritization – adding residual connections

The results of the experiment are presented in Figure 6.4. The first subplot shows accuracy of the models when no language model is incorporated. The model with 2 layers still performs best, but the differences between models are now smaller than if no residual connections were used. The other three subplots then show the effect of incorporating the language model. These graphs are much more interesting. We can see that the accuracy increased at both model with 2 and 3 layers. Moreover, the achieved accuracy of the model with 3 layers and the model with 4 layers is greater than the accuracy achieved by the model with 2 layers. This confirms the hypotheses that the residual connections allow training of deeper models and, moreover, also slightly increase performance of current models.

Intrinsic evaluation and other notes

When training models for longer time (approximately 7 days on CPU, 400k batches), the accuracy sometimes suddenly dropped. Since we did not observe this behaviour when the L2-loss was used as a regularization term in the cost function, we suspect that the models may sometime learn too big weights that may indicate model overfitting.

The beam search algorithm (see Algorithm 2) that is used for finding the most probable diacritization is designed generally, so that it may be applied to all tasks that char2char model is applicable to. For each input character, it takes k-most probable model outputs and utilizes them to extend current beam of hypothesis. It is clear that the algorithm does not have to use the model outputs when the particular input character cannot be diacritized. After implementing this modification to the beam search decoder, the achieved accuracy of all models improved slightly by approximately 0.05 percent when the language model was utilized. We inspected sentences, on which the new algorithm performs better, and found out that the original beam search algorithm with the language model occasionally rewrites the original character *'\'* to *'-'*, *'-'* to *','* or whole word *1992* to *1997*. The reason why this happens is the language model. Even though char2char model outputs these wrong characters with low probability, the language model assigns them later high probability, convincing the algorithm to use the wrong ones.

The summarized results of all experiments are presented in Table 6.4. Note that all hyperparameters including the language model weight α and the beam size were chosen to maximize system performance on the development set.

System	Precision	Recall	$F_{0.5}$ score	Accuracy
No language model	96.02	95.76	95.97	97.60
With language model of rank 3	98.57	98.46	98.55	99.11
With language model of rank 5	98.62	98.55	98.61	99.15
With residual connections	98.69	98.60	98.67	99.18
Modified beam search	98.87	98.60	98.82	99.25

Table 6.4: Comparison of different modifications to char2char model on the diacritization task.

When describing the architecture of char2char model (see Section 3.1), we mentioned that the similar characters may be grouped in one part of the space when embedded. To check whether this is actually true, we embedded all characters from the training corpus using the trained model. To visualize these d-dimensional vectors, we used t-distributed stochastic neighbor embedding [Maaten and Hinton, 2008] to embed these vectors to 2D-space. To make the visualization more illustrative, we further divided the characters into four groups: letters to which the diacritics cannot be generated (e.g. b, H or W), letters to which diacritics can be added (e.g. a, o, Y), digits and the last group (other) containing the rest of symbols. The visualization of the described embeddings split in the groups is presented in Figure 6.5.

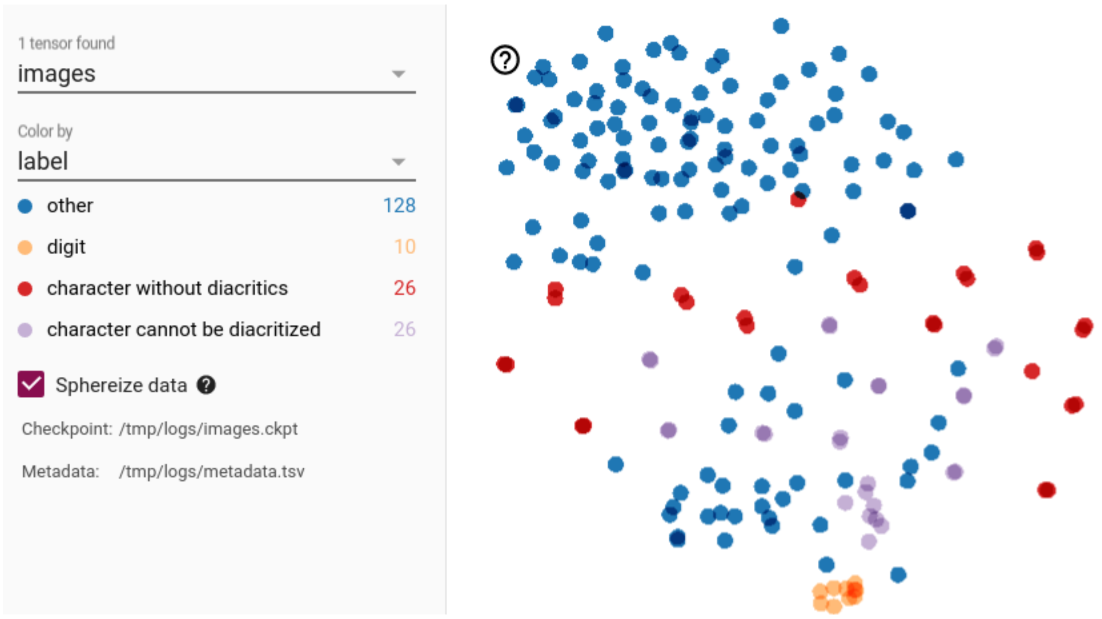


Figure 6.5: Char2char model embeddings for individual characters

As we can see in the visualization, the model learnt that all digits have roughly the same meaning for the diacritization task and placed them to the similar part of the space. It also seems important that letters that cannot be diacritized are separated from letters to which the diacritization can be generated. The intuition behind the placement of rest characters is unclear, which could be caused by the too raw group division. Nevertheless, we can state that the initial hypothesis was correct and the embeddings of the trained model are meaningful with respect to given task.

6.4.2 Word2word model

The experiments conducted on char2char model gave us basic intuition on how to set model parameters. Therefore, the first experiment we performed with word2word model tries to adopt settings from char2char model. The model has 3 stacked layers, the dimension of word embeddings and also the size of RNN unit in correcting module was set to 300. To compute the embeddings, we utilize C2W model (see Section 4.2.1), which operates on embedded characters of dimension 150. The size of RNN unit in the decoder was also 300. All RNN units are GRU. Dropout is used both on embedded characters and between correcting module layers. Finally, Adam optimizer with a learning rate of 10^{-4} and a batch size of 50 was used to train the network.

We trained the model for 22 days (191k batches) on a single CPU. The positive fact is that possibly due to the large training corpora, the model seemed to be improving all the time. The accuracy of the model without the language model incorporation is 97.08 percent. When the language model is incorporated, the accuracy increases almost to 99 percent. The illustration of how a selected beam size and a language model weight α affect the accuracy is presented in Figure 6.6. Similarly to char2char model, the best accuracy is achieved with the beam size of 8, the beam size of 16 and 4 shows satisfactory results and the beam size of 2 performs the worst. The optimal α seems to be somewhere between 0.5 a 1.0.

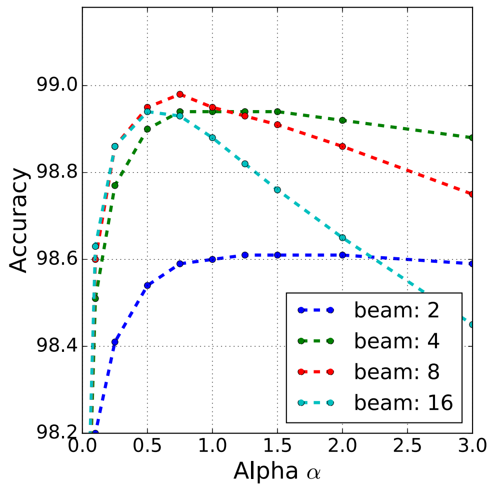


Figure 6.6: Word2word model diacritization – incorporating language model

Using error classifier

In Section 4.4 we proposed the extension to word2word model. Word2word model with this extension corrects only those words that are flagged as wrong by the error classifier. To check, whether this extension may help the model perform better in the diacritization task, we performed three experiments. The experiments have two differences compared to the experiment conducted in the previous section. Firstly, they utilize the error classifier. Secondly, each of them uses different size of correcting module RNN size and C2W RNN size. The first experiment uses correcting module with RNN size of 200 and C2W model with RNN size of 100, the second experiment 300 and 150 and the third model 400 and 200, respectively.

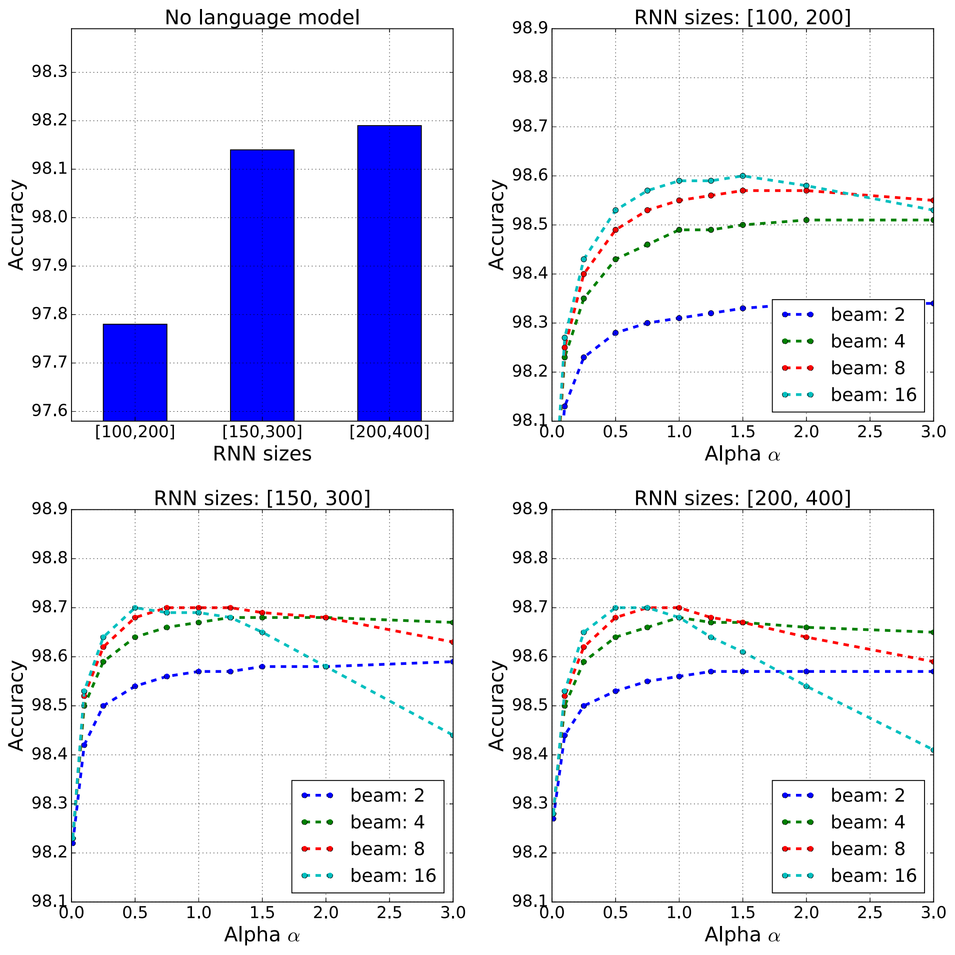


Figure 6.7: Word2word model diacritization – using error classifier

Results of the experiments are presented in Figure 6.7. The first fact to note is that similarly to char2char model, it is important to choose hyperparameters (e.g. RNN size) carefully, because they substantially influence the performance. In comparison to the version without the error classifier, the accuracy of the models without a language model increased. On the other hand, when the language model is utilized, the accuracy increase is much smaller than we could have seen in all previous experiments. To understand why this happens, it is useful to inspect Table 6.5, which besides accuracy also shows precision, recall and $F_{0.5}$ score of the models. Note that to compare similar experiments, the table shows the results of the second experiment with the error classifier, i.e. both experiments in the table utilize RNN cells of the same size.

System	Precision	Recall	$F_{0.5}$ score	Accuracy
Standard word2word without LM	95.12	94.88	95.08	97.08
Standard word2word with LM	98.31	98.43	98.33	98.98
Word2word with error cls. without LM	97.43	96.50	97.24	98.19
Word2word with error cls. with LM	98.52	97.33	98.28	98.70

Table 6.5: Test set performance comparison of different modifications to word2word model on the diacritization task.

In Table 6.5 we can see that the error classifier on one hand increased the precision of the model above the precision of the standard model, however on the other hand the recall remained smaller. The low recall indicates that the error classifier failed to flag some words as wrong giving the language model no chance to increase performance. Therefore, word2word model with the error classifier should be preferred over the standard word2word model only in cases, when precision is emphasized substantially more than recall. In all other cases, we suggest to use standard word2word model instead.

6.4.3 Translation model

The translation model is the most complex model and, consequently, it takes the longest time to train. Therefore, only one experiment with translation model was conducted on the diacritization tasks. The experiment utilized pyramidal encoder and multilayer decoder with 3 layers and GRU cells with size 200. Dropout was used on the embedded inputs as well as between encoder layers and between decoder layers for regularization purposes. Finally, Adam optimizer with a learning rate $3 \cdot 10^{-4}$ and a batch size of 100 was used to train the network.

The model was trained for approximately 2 weeks on a single GPU (NVIDIA GeForce GTX 1080). When evaluating performance of the model, we found out that the model sometimes does not output the same number of words as was in the input. In most cases, it failed to generate the final dot. For this reason, all model outputs were postprocessed and when there was a missing word, a random non-sense word was generated.

The accuracy of translation model when used without a language model and with the beam size of 16 is 93.51 percent. The graph showing the effect of a selected beam size and a language model weight α on the accuracy of the model with the language model is presented in Figure 6.8. We can see that the best accuracy is achieved with the beam size of 16 and the language model weight 0.5.

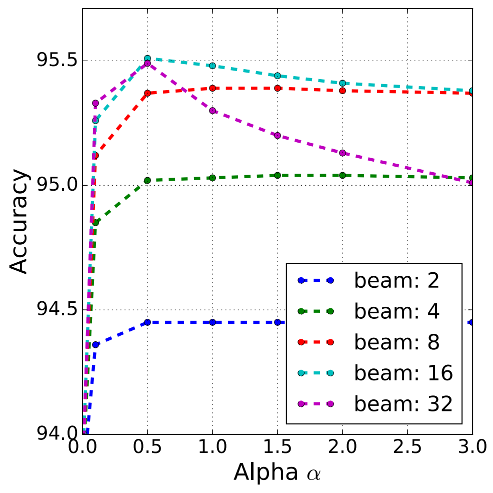


Figure 6.8: Translation model diacritization – incorporating language model

6.4.4 Results comparison

We compare performance of our models with Korektor (see Section 2.3.2), CZACCENT[2], which is the diacritization tool developed at the Faculty of Informatics at Masaryk University, Microsoft Office Word 2010 and ASpell.

The evaluation with Korektor and CZACCENT was simple, because both these tools have command-line, respectively web interface, capable of generating diacritics for the whole text at once. Therefore, these evaluations were performed on the whole test set.

Microsoft Office Word 2010 correcting module and ASpell are primarily spell checking tools that for each detected misspelling offer a list of possible corrections. From these suggestions, we always chose the first one, which is supposed to be the most probable. Since this process requires a non trivial amount of manual effort, we corrected only the first 1000 sentences from PDT3.0 test set. Finally, from the corrected sentences, we chose only those sentences that were created by adding diacritics (746 for Microsoft Office Word and 636 for ASpell).

As we can see in Table 6.6, standard spelling correctors that are not tuned on the diacritization task perform badly. The achieved accuracy of 89.10 percent for Microsoft Office 2010, respectively 88.39 percent for ASpell is in comparison with other tools simply poor. The diacritization tool CZACCENT achieves much better results, but its performance is still rather weak when compared to Korektor. Finally, two of our proposed models outperform all current diacritization tools.

The performance of the translation model is low when compared to other

[2]`https://nlp.fi.muni.cz/cz_accent/`

Tool	Accuracy	$F_{0.5}$ score
Microsoft Office Word 2010 (*)	89.10	86.20
ASpell (*)	88.39	83.11
CZACCENT	96.07	94.33
Korektor	98.61	98.02
Char2char model	**99.25**	**98.82**
Word2word model	**98.98**	**98.33**
Translation model	95.51	91.47

Table 6.6: Diacritization result. Note that all measurements, but Microsoft Office Word 2010 and Aspell, were measured on whole testing set.

tools and models. We suspect that the main reason for this is its complexness and the design that allows it to be used for any grammatical error correction task.

Word2word model, despite being designed for more general tasks, achieves 98.98 percent in accuracy and 98.33 percent in $F_{0.5}$ score beating the Korektor performance by more than 0.3 percent.

The best performance is achieved by char2char model that surpasses Korektor results by more than 0.6 percent in accuracy and by more than 0.8 percent in $F_{0.5}$ score. Out of all our three models, char2char model trains the fastest and also its memory footprint is lowest. Consequently, using char2char model for the diacritization task seems to be a an obvious choice.

6.4.5 Sample corrections

Several sentences corrected by our system and Korektor are shown in Table 6.7. The sentences were chosen to show two main disadvantages of Korektor: a fixed vocabulary and a fixed word context. On the other hand, our system is capable of adding diacritics to rare words and considering whole left and right contexts when generating diacritics.

Table 6.8 contains several sentences that were generated by our system and considered wrong, even if they are valid Czech sentences. Note that the last pair of sentences in the table shows an example of wrong gold sentence and correct system output.

6.5 I vs Y

Similarly to the diacritization problem, all three models could be trained to perform this task. However, because we found char2char model performing best in the diacritization task, we decided to train only the char2char model. The utilized

	Correction
K	V počátku rekonstrukce bydleli na tuchomericke faře .
O	V počátku rekonstrukce bydleli na tuchoměřické faře .
K	Ceny benzinu jsou v naší zemi snad tím nejcitlivejsim indikátorem nálady lidí .
O	Ceny benzinu jsou v naší zemi snad tím nejcitlivějším indikátorem nálady lidí .
K	Ponechali jsme personálně zredukované ředitelství , vypustili střední řidiči článek - závody - a mírně jsme posílili technicko - hospodářský úsek středisek .
O	Ponechali jsme personálně zredukované ředitelství , vypustili střední řídící článek - závody - a mírně jsme posílili technicko - hospodářský úsek středisek .
K	Počet nových registrací automobilu dosáhl v lednu 1.085 milionů .
O	Počet nových registrací automobilů dosáhl v lednu 1.085 milionu .

Table 6.7: Sample diacritization outputs proposed by our system (O) and Korektor (K).

	Correction
G	(Vyšší hmotnosti výrobků na závadu nejsou .)
O	(Vyšší hmotnosti výrobku na závadu nejsou .)
G	Sehnat pracovníky , kteří by se k tomuto majetku dokázali chovat jako ke svému , je velmi obtížné , prohlašují .
O	Sehnat pracovníky , kteří by se k tomuto majetku dokázali chovat jako ke svému , je velmi obtížné , prohlašuji .
G	Potvrdila to skutečnost , že hned při zahájení výroby byly kvalitativní parametry automobilů minimálně na úrovni japonských závodů .
O	Potvrdila to skutečnost , že hned při zahájení výroby byly kvalitativní parametry automobilu minimálně na úrovni japonských závodů .
G	A protože ji vznikla konkurence i v dalších obcích , musela zkoušet nové cesty jak si zákazníky udržet .
O	A protože jí vznikla konkurence i v dalších obcích , musela zkoušet nové cesty jak si zákazníky udržet .

Table 6.8: Sample diacritization outputs that were consider wrong despite being valid Czech sentences – Note that G stands for gold sentence and O for correction produced by our system.

model has 2 stacked layers, the dimension of character embedding as well as the dimension of GRU cells is 200. Dropout of keep rate 0.8 is used on the embedded characters as well as between stacked layers. To train the model, Adam optimizer with a learning rate 10^{-4} and a batch size 200 is used.

The trained model achieves accuracy of 99.58 percent. When the language model of rank 5 is incorporated, this value increases up to 99.82. As we can see in Table 6.9, our model surpasses Korektor performance by more than 0.2 percent in accuracy and more than 0.6 percent in $F_{0.5}$ score.

System	Precision	Recall	$F_{0.5}$ score	Accuracy
Korektor	99.00	96.91	98.57	99.56
char2char	99.28	99.03	99.23	99.82

Table 6.9: I vs Y system comparison.

6.6 Czech spelling and basic grammar correction

Both word2word model and translation model may be trained to correct spelling and basic grammatical errors. However, because translation model is computationally demanding and also did not reach comparable results in the diacritization task, we decided to experiment only with word2word model in this task.

As described in Section 6.1.3, we acquired three datasets. The first (czesl-word2simword) contains training, development and testing sets. Despite the fact that we tried to make the training set as big as possible, it still contains substantially less sentences than the training set for the diacritization task. Therefore, we generated one more training corpus (czesl-word2simword-big) containing the similar errors to those occurring in the czesl-word2simword training set. The construction of this dataset is described in Section 6.6.1. The second utilized dataset is Czech-SEC-AG, which also contains training, development and testing sets. Since we implemented a script that is capable to generate spelling errors from a character error model and a corpus of clean text, we are not limited by the size of the training corpus. Therefore, for training purposes, we utilized SYN2010 with substantially more errors (approximately 20 percent) than in the testing set. The last dataset is Švejk, which contains only test sentences.

Ideally, we would like to have one model that would perform satisfactory on all test sets. However, each dataset has its own specific types of errors. Therefore, we created one more training set (spelling-combined). This training set is a mixture of all described training corpora. Namely, it contains the original czesl-word2simword training set, 500 000 sentences from Czech-SEC-AG training dataset and 500 000 sentences of newly created czesl-word2simword-big dataset.

In the next sections we discuss our experiment with respect to the given test sets and in the end compare achieved results with Korektor and Microsoft Office Word 2010.

6.6.1 Generating new training set

As described in Section 4.6, we can utilize word2word model to generate artificial errors. Inspired by this idea, we trained word2word model on inverse dataset, i.e. the input training sentences were gold training sentences of czesl-word2simword and the gold training sentences were the input sentences of czesl-word2simword. The trained model used the same hyperparameters as the standard models described in the next section. The only difference to standard correcting models is that this model uses error classifier to determine which words to decode. To create errors with this model, the process described in the next paragraph is applied on each input sentence.

First of all, certain number of input words to be damaged is chosen randomly with fixed probability (30 percent). After this, we run the model on input sentence and sort the error classifier word error probabilities. Finally, the decoder is executed on the k word with highest error probability, where k is the precomputed number of words to be damaged.

The described process was then applied on 1 000 000 sentences from SYN2010 corpus, which resulted in new dataset czesl-word2simword-big.

6.6.2 Basic experiments

All in all, we have four training and three testing sets. On each training set, we trained one model. All models have the same architecture – 3 stacked layers, the dimension of word embeddings and also the size of RNN unit in correcting module was set to 300. To construct these embeddings, we utilized C2W model (see Section 4.2.1), which operates on embedded characters of dimension 150. The size of RNN unit in the decoder is also 300. All RNN units are GRU. Dropout of keep rate 0.8 is used both on embedded characters and between correcting module layers. To train the network, Adam optimizer with learning rate 10^{-4} and batch size of 50 is used.

Apart from these four models, we trained two more models that utilize Charagram model to create word embeddings (see Section 4.2.3). These models were trained on Czech-SEC-AG and spelling-combined datasets with the same hyperparameters as the previous models. The n-gram dictionary of Charagram model contained top 100 000 occurring {1,2,3,4}-grams.

Table 6.10 shows $F_{0.5}$ score of all models without a language model incorporation with respect to three test sets. There are several things to mention:

- The model trained on czesl-word2simword-big training set achieves the highest $F_{0.5}$ score on czesl-word2simword test set and similarly the models

Train \ Test	czesl-word2simword	Czech-SEC-AG	Švejk
czesl-word2simword (C2W)	47.03	17.92	16.15
czesl-word2simword-big (C2W)	65.82	14.83	16.36
Czech-SEC-AG (Charagram)	42.42	82.30	54.28
Czech-SEC-AG (C2W)	40.11	78.72	54.14
spelling-combined (Charagram)	65.36	62.62	31.66
spelling-combined (C2W)	65.79	66.75	35.87

Table 6.10: Results of the spelling and basic grammar correction

trained on Czech-SEC-AG training sets performs the best on the Czech-SEC-AG test set. The performance of these models rapidly decrease when they are applied on two other datasets. We can see that both models that were not trained on any part of Czech-SEC-AG dataset have very poor results on this dataset. This confirms our claim that there are differences in error types in the datasets.

- The model trained on spelling-combined dataset achieves the second highest score on czesl-word2simword and Czech-SEC-AG datasets. This indicates that despite there are differences in the datasets, there is a chance that they could be captured by a single model.

- Švejk dataset is most challenging for our models. The main reason is possibly the fact that none of our datasets is capable of fully capturing the errors present in the dataset.

- The model trained on czesl-word2simword-big performs substantially better than the model which was trained on czesl-word2simword. In Section 4.6, we stated that the bigger dataset should help mainly more complex models requiring more data. Nevertheless, the results indicate that the bigger dataset, despite being less human-like, may help even the equally complex model.

- From the presented results, it is difficult to say whether it is better to use Charagram model or C2W model. Charagram model performs slightly better for Czech-SEC-AG model, while C2W model works slightly better for the model trained on spelling-combined dataset. Nevertheless, when inspecting the training logs, we found out that Charagram model converges much faster. On the other hand, because Charagram model contains relatively large embedding matrix for most common n-grams, it requires more memory.

6.6.3 Incorporating language model

We trained several models in the previous section. To fully reveal their potential, we incorporate a language model while inferring. The language model is of rank 5 and was trained as described in Section 6.3. To incorporate the language model, a language model weight α and a beam size has to be set. Similarly to the diacritization task, we chose the values of language model weight α and beam size that maximize the system performance with respect to one of two development sets. In other words, we have two pairs of these constants for each model. One pair maximizes its performance on the czesl-word2simword development set and the second pair maximizes its performance on the Czech-SEC-AG development set. The results of this process are presented in Table 6.11. Since the process requires a non-trivial gridsearch, we did not run it for three models. We did not optimize the models trained on czesl-word2simword and czesl-word2simword-big on Czech-SEC-AG dataset (the version with no language model performed poorly) and we also did not optimize the model with Charagram embeddings trained on the spelling-combined training set set for any test sets (the version with no language model was worse than the same model with C2W embeddings).

Train \ Test	czesl-word2simword	Czech-SEC-AG	Švejk	Constants
czesl-word2simword (C2W)	<u>66.05</u>	25.75	17.70	[0.75, 16]
czesl-word2simword-big (C2W)	<u>72.16</u>	34.03	26.52	[0.1, 16]
Czech-SEC-AG (Charagram)	<u>50.16</u>	77.94	47.34	[0.75, 16]
Czech-SEC-AG (Charagram)	47.99	<u>89.20</u>	58.65	[0.07, 16]
Czech-SEC-AG (C2W)	<u>46.85</u>	75.48	48.50	[0.75, 8]
Czech-SEC-AG (C2W)	45.10	<u>86.86</u>	58.61	[0.07, 16]
spelling-combined (C2W)	<u>72.36</u>	75.49	40.22	[0.25, 8]
spelling-combined (C2W)	71.02	<u>78.37</u>	42.75	[0.1, 16]

Table 6.11: Results ($F_{0.5}$ score) of the spelling and basic grammar correction with language model incorporation – Note that the last column shows the optimal constants that maximize the system performance on the underlined dataset.

The results presented in Table 6.11 show several notable points:

- Incorporating the language model substantially increases performance of all models.
- The model trained on the spelling-combined training set with Charagram embeddings surpasses all other models on the czesl-word2simword testing set. Namely, it outperforms both models that were trained solely on czesl-word2simword or its derivative.

- The highest $F_{0.5}$ score on the Czech-SEC-AG testing set was achieved by the models that were trained on its training set. Nevertheless, the performance of the model trained on the spelling-combined training set seems not to be bad as well.

- When focusing on the Švejk testing set, the models for which the constants were chosen to maximize their score on the Czech-SEC-AG testing set outperform the models for which the constants were chosen to maximize their score on the czesl-word2simword. The performance gap is most evident on models that were trained on the Czech-SEC-AG training set. This fact indicates that the error types in the Švejk dataset substantially more resemble the error types present in the Czech-SEC-AG dataset. When inspecting Švejk dataset in detail, we found out that its gold sentences contain slang terms and colloquialisms. This is in contrast to czesl-word2simword dataset, which strictly corrects all these words. Therefore, the model trained on the spelling-combined training set performs significantly worse on the Švejk testing set than the model trained on Czech-SEC-AG training set.

Out of 5 models presented in Table 6.11, we further consider only two models - spelling-combined (C2W) and Czech-SEC-AG (Charagram), which performed best.

So far we were maximizing system performance with respect to one dataset. However, when running a model in production mode, it would make sense to maximize system performance with respect to all three testing sets. Supposing the results on all three datasets are equally important, the constants should maximize the sum of achieved scores on all three datasets.

When inspecting the performed measurements, we found out that maximizing the system performance on czesl-word2simword leads to higher score on this dataset, however, the achieved score on two other datasets decreases. Similarly, maximizing the model performance on Czech-SEC-AG leads to increase in performance on this and also Švejk datasets, but to decrease in performance on czesl-word2simword. Therefore, the constants selected for the final comparison are the constants that maximize the system performance on Czech-SEC-AG dataset. The comparison of our models and currently existing spelling models is presented in Table 6.12.

As we can see in Table 6.12, no system achieves highest score on all three datasets. Nevertheless, we can see that if our model is trained on particular training set, it is capable of outperforming the statistical tools. In other words, the model trained on the spelling-combined training set achieves the highest score on both czesl-word2simword and Czech-SEC-AG datasets and the model trained

System \ Test	czesl-word2simword	Czech-SEC-AG	Švejk
Microsoft Office Word 2010	60.68	–	49.05
Korektor	62.08	64.47	58.05
Czech-SEC-AG (C2W)	45.10	**86.86**	**58.61**
spelling-combined (C2W)	**71.02**	**78.37**	42.75

Table 6.12: Comparison of our model with currently existing spelling correctors

on the Czech-SEC-AG training set outperforms all other model on the Czech-SEC-AG testing set. Moreover, this model shows slightly better performance also on Švejk dataset, which is the training corpus for Korektor. We suspect that to achieve higher score on Švejk dataset, we would have to tune our training sets to contain more error types occurring in Švejk dataset.

Finally, we present several corrections of our system spelling-combined (C2W) and compare them to corrections produced by Korektor. The first 5 sentences in Table 6.13 show that our system is capable of correcting even quite complex sentences, while last 2 sentences show that sometimes our system either does not correct erroneous sentence or provides wrong correction. We suspect that the main reason, why our system may sometimes struggle to recognize whether an input word is wrong, is that in comparison to Korektor it does not posses a dictionary. Therefore, extending character level word embeddings with at least information, whether current word is in the dictionary, could substantially help the model recognize misspelled words. Such extension is a proposal for future work.

6.7 English grammar correction

The last set of experiments we conducted is devoted to grammatical error correction. Generally, we could use both word2word and translation model for these experiments. However, since word2word model is designed for rather smaller edits, we decided to use only translation model.

The design of translation model is very similar to those proposed by Xie et al. [2016], who also experimented with CoNLL 2014 shared task. Therefore, we decided to keep their hyperparameters for all training. The model uses pyramidal encoder and multilayer decoder with 3 layers and GRU cells with size 400. Dropout is used on the embedded inputs as well as in the encoder and in the decoder for regularization purposes. Finally, Adam optimizer with a learning rate $3 \cdot 10^{-4}$ and a batch size of 100 is used to train the network.

To train models, we experimented with three training sets:

I	Máma má heský oči .
K	Máma má hezký oči .
O	Máma má hezké oči .
I	Ale v okamžiku jsem se rozhodnula , že musím změnit svůj život .
K	Ale v okamžiku jsem se rozhodnula , že musím změnit svůj život .
O	Ale v okamžiku jsem se rozhodla , že musím změnit svůj život .
I	Moje dětství bylo nejobyčejnejši : plenky , dřevený a gumový hračky .
K	Moje dětství bylo nejobyčejnejši : plenky , dřevěný a gumový hračky .
O	Moje dětství bylo nejobyčejnější : plenky , dřevěné a gumové hračky .
I	kuře na kary je nejoblíbenejší a nejznamnejší jídlo v Indiji .
K	kuře na kari je nejoblíbenější a nejznamnejší jídlo v Indii .
O	Kuře na kari je nejoblíbenější a nejznámnější jídlo v Indii .
I	V sále šuměly hlasi , parketi šustily od dívčíh střevíčku , lustry se houpali , jak horký vzduch stoupal vzhůru ke stropu , avšak ubívajíci svíčky byli jako uslzené .
K	V sále šuměly hlasy , parkety šustily od dívčího střevíčku , lustry se houpali , jak horký vzduch stoupal vzhůru ke stropu , avšak ubývající svíčky byli jako uslzené .
O	V sále šuměly hlasy , parkety šustily od dívčích střevíčků , lustry se houpaly , jak horký vzduch stoupal vzhůru ke stropu , avšak ubývající svíčky byly jako uslzené .
I	Které přeměty tě baví ?
K	Které přemety tě baví ?
O	Které přeměty tě baví ?
I	A taky možná bude potřebovat mamka opčas s něčím pomoct , protože je těhotná a opčas jí není dobře .
K	A taky možná bude potřebovat mamka občas s něčím pomoct , protože je těhotná a občas jí není dobře .
O	A taky možná bude potřebovat mamka počas s něčím pomoct , protože je těhotná a opčas jí není dobře .

Table 6.13: Sample corrections produced by Korektor (K) and our system (O) on several input sentences (I)

- NUCLE corpus
- NUCLE corpus + Lang8 corpus
- NUCLE corpus + subset of Lang8 corpus sentences, where each input sentence contains at least one error (approx. 430 000 sentences)

On each training set, we trained the model for circa 3 weeks (approx. 30 epochs for the model operating on the biggest training set) on NVIDIA GeForce GTX 1080. After each epoch, all models (their respective weights) were saved.

Once the training was finished, out of all saved checkpoints of the particular model, we chose the one with the highest $F_{0.5}$ score on the development set. The results of the trained models on the CoNLL 2014 testing set are presented in Table 6.14. Note that a beam size of 64 was used when decoding.

Training set	$F_{0.5}$ score
NUCLE corpus	0.089
NUCLE + Lang8	0.126
NUCLE + Lang8 erroneous	0.185

Table 6.14: Translation model performance on CoNLL 2014 test set

Table 6.14 shows that using only original NUCLE corpus for training leads to the worst results. When whole Lang8 corpus is added to NUCLE for training, the overall $F_{0.5}$ score increases, but not as much as if only erroneous sentences from Lang8 are added. The model trained on NUCLE and subset of erroneous sentences from Lang8 achieves highest performance. Unfortunately, the achieved $F_{0.5}$ score is lower than those achieved by Xie et al. [2016], who claim to achieve 19.81 $F_{0.5}$ score with their model. The reason why our model achieved slightly worse results may be either a little different attention mechanism used by Xie et al. [2016] or different initial values used for the weights. We also think that decaying the learning rate may improve results.

We also examined the corrections performed by our system. We found out that the system prefers making rather smaller edits. The most common errors our system correctly handles are: correcting word misspellings (indivdial $\rightarrow$ individual), adding (removing) "s" to the verb ending (he read $\rightarrow$ he reads), changing verb into past tense, changing singular to plural form (structure $\rightarrow$ structures) and article generation or deletion. The system occasionally performs bigger changes like reorganization of multiple words, but these changes are quite rare and often wrong.

When the language model of rank 5 trained on English GigaWord [Parker et al., 2011] is incorporated, $F_{0.5}$ score of 26.5 is achieved.

7. Our implementation

This chapter is both a user documentation and a description of attached file contents. The attached file contains the text of this thesis in PDF format (`thesis.pdf`), description of the attachment (`README.md`) and the folder (`code`) with the source code for replicating and running our experiments. All scripts are written in Python 2.7. To train and run models, TensorFlow[1] 0.12.1 is required.

Folder `code` consists of five subfolders: `data`, `common`, `char2char`, `word2word` and `translation`. The subfolder `data` is intended for storing corpora. The subfolder `common` contains several scripts that are shared by all other folders. The rest three subfolders store the source code that is needed to train and run the particular model.

7.1 Training models

Before training a model, a dataset must be prepared. A common dataset for any natural correction task consists of six files containing: train set inputs, train set targets, development set inputs, development set targets, test set inputs and test set targets. Paths to these files are stored in a single configuration file, which on each line contains the type of the set (e.g. *train_inputs*) and its path. The official diacritization dataset derived from PDT3.0 corpus is prepared in folder `data/diacritization`. The configuration file storing the required information is `data/diacritization/configuration.txt`.

After preparing the dataset configuration file, there are three available models to train. Note that each model requires the data to satisfy specific restrictions. For instance char2char model requires that both the input and target sentence have the same length. To train the selected model, each model folder contains the script `train.py`. Both char2char model and word2word model train scripts contain an explicit argument (`--dataset`) to set the configuration file with the dataset to be used. The script for translation model allows specifying only train sets (as two first arguments) and the development and the testing set performance must be controlled from outside. The train script of all three models then contains another set of parameters that are used to specify the training settings. These are for example the experiment name, the directory to save the experiment checkpoints to or the model hyperparameters. To view the full list of these parameters together with the default values, run the train script with option `-h`.

Let us give a practical example on how to train a simple diacritization sys-

[1]`https://www.tensorflow.org/`

tem. We train char2char model with 2 layers for 5 epochs and keep the default value of the rest of model hyperparameters. Because we train the diacritization system, the name of the experiment is set to `diacritization`. The training script automatically creates two new folders containing the current timestamp in `save/diacritization` and `logs/diacritization`. The first folder stores the model checkpoints together with some configuration files. The second folder then contains logs generated for Tensorboard.[2] The command to perform described training is:

```
python char2char/train.py --dataset data/diacritization/configuration.txt
  --num_layers 2 --epochs 5 --exp_name diacritization
```

7.2 Running models

Once a model is trained, it can be used for correcting text. For this purpose, each model folder contains script `infer.py`. This script has three mandatory positional arguments: path to file with sentences to be corrected, path to file to store corrected sentences and path to folder with the model checkpoints. The script can also utilize a trained language model (see Section 6.3) using the following arguments: `--lm`, which points to the language model binary file, `--beam_size` to specify the size of the used beam and `--alpha` to specify the language model weight.

Each model folder includes a directory `save`, which contains the best performing models from Chapter 6. Namely, `char2char/save` contains the diacritization and the I vs Y models, `word2word/save` contains two models for the spelling and basic grammar correction and `translation/save` contains a model for the English grammatical correction task.

To give a practical example of how to run a trained model, we show a command for generating diacritics using char2char model. Let us suppose that `uncorrected.txt` stores sentences without diacritics. The command that runs the diacritization model to create diacritized sentences stored in `corrected.txt` looks as follows:

```
python char2char/infer.py uncorrected.txt corrected.txt
char2char/save/c2c_diacritization/2017-02-28_194903/
```

Note that because the size of the language models used in Chapter 6 is 5GB, respectively 14GB, we do not include them in the attachment.

[2]https://www.tensorflow.org/get_started/summaries_and_tensorboard

8. Conclusion

This thesis examined the specific tasks of and current approaches to the natural language correction. Inspired by the current accomplishments of deep neural networks in areas such as machine translation or named entity recognition, we proposed and implemented three models based on neural networks for various natural language correction tasks ranging from general grammar correction to the specific task of diacritization. Specifically, the trained models were evaluated on the four chosen tasks: diacritization, I vs Y, spelling and basic grammar correction, and general grammar correction. To evaluate the models in regards to all tasks except for the general grammar correction, we acquired and created several datasets. As even the current best systems do not achieve satisfactory results in the general grammar correction task, we focused mainly on the first three chosen tasks, for which a practical system may be used in practice.

The accuracy achieved in the diacritization and I vs Y task seems to be significantly higher than the accuracy of currently used statistical tools. The trained models could therefore be used in production mode when most precise results are desired. The models trained on the spelling and basic grammar correction tasks outperform current spelling correctors if a dedicated model is used for each dataset, with the best single model outperforming existing systems on two of the three datasets. This indicates that these models could replace current systems, but may not always achieve the best performance. Finally, the results achieved in the general grammar correction are slightly worse than those achieved by the reference paper. This is likely caused by the minor differences in the model architecture.

Our future work includes enhancing our models with several other features. In order to make the current model for spelling and basic grammar correction more accurate, the character level word embedding needs to be complemented by a standard word embedding. Furthermore, residual connections could be implemented in both word2word and translation model to allow a training of deeper models. Finally, to allow usage of our models in production mode, inferring process could be parallelized to make the decoding faster.

Bibliography

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Eduard Bejček, Eva Hajičová, Jan Hajič, Pavlína Jínová, Václava Kettnerová, Veronika Kolářová, Marie Mikulová, Jiří Mírovský, Anna Nedoluzhko, Jarmila Panevová, Lucie Poláková, Magda Ševčíková, Jan Štěpánek, and Šárka Zikánová. Prague dependency treebank 3.0, 2013. URL `http://hdl.handle.net/11858/00-097C-0000-0023-1AAF-3`. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3 (Feb):1137–1155, 2003.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

Chris Brockett, William B Dolan, and Michael Gamon. Correcting esl errors using phrasal smt techniques. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 249–256. Association for Computational Linguistics, 2006.

William Chan, Navdeep Jaitly, Quoc V Le, and Oriol Vinyals. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.

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

Martin Chodorow, Joel R Tetreault, and Na-Rae Han. Detection of grammatical errors involving prepositions. In *Proceedings of the fourth ACL-SIGSEM workshop on prepositions*, pages 25–30. Association for Computational Linguistics, 2007.

Shamil Chollampatt, Kaveh Taghipour, and Hwee Tou Ng. Neural network translation models for grammatical error correction. *arXiv preprint arXiv:1606.00189*, 2016.

Daniel Dahlmeier and Hwee Tou Ng. Better evaluation for grammatical error correction. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 568–572. Association for Computational Linguistics, 2012.

Daniel Dahlmeier, Hwee Tou Ng, and Eric Jun Feng Ng. Nus at the hoo 2012 shared task. In *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pages 216–224. Association for Computational Linguistics, 2012.

Daniel Dahlmeier, Hwee Tou Ng, and Siew Mei Wu. Building a large annotated corpus of learner english: The nus corpus of learner english. In *Proceedings of the Eighth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 22–31, 2013.

Robert Dale and Adam Kilgarriff. Helping our own: Text massaging for computational linguistics as a new shared task. In *Proceedings of the 6th International Natural Language Generation Conference*, pages 263–267. Association for Computational Linguistics, 2010.

Robert Dale, Ilya Anisimoff, and George Narroway. Hoo 2012: A report on the preposition and determiner error correction shared task. In *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pages 54–62. Association for Computational Linguistics, 2012.

Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

Mariano Felice, Zheng Yuan, Øistein E Andersen, Helen Yannakoudakis, and Ekaterina Kochmar. Grammatical error correction using hybrid systems and type filtering. In *CoNLL Shared Task*, pages 15–24, 2014.

G David Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

Alex Graves. Supervised sequence labelling. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 5–13. Springer, 2012.

Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18 (5):602–610, 2005.

Marcin Junczys-Dowmunt Roman Grundkiewicz. The amu system in the conll-2014 shared task: Grammatical error correction by data-intensive and feature-rich statistical machine translation. *CoNLL-2014*, page 25, 2014.

Na-Rae Han, Martin Chodorow, and Claudia Leacock. Detecting errors in english article usage by non-native speakers. *Natural Language Engineering*, 12(02): 115–129, 2006.

Jirka Hana, Alexandr Rosen, Barbora Štindlová, and Jan Štěpánek. Building a learner corpus. *Language resources and evaluation*, 48(4):741–752, 2014.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. *arXiv preprint arXiv:1508.06615*, 2015.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kevin Knight and Ishwar Chander. Automated postediting of documents. In *AAAI*, volume 94, pages 779–784, 1994.

Tom Kocmi and Ondřej Bojar. Subgram: Extending skip-gram word representation with substrings. In *International Conference on Text, Speech, and Dialogue*, pages 182–189. Springer, 2016.

M Křen, T Bartoň, V Cvrček, M Hnátková, T Jelínek, J Kocek, R Novotná, V Petkevič, P Procházka, V Schmiedtová, et al. Syn2010: žánrově vyvážený korpus psané češtiny. *Ústav Českého národního korpusu FF UK, Praha*, 2010.

Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *arXiv preprint arXiv:1610.03017*, 2016.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso. Finding function in form: Compositional character models for open vocabulary word representation. *arXiv preprint arXiv:1508.02096*, 2015.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

Tomáš Mikolov, Anoop Deoras, Stefan Kombrink, Lukáš Burget, and Jan Černockỳ. Empirical evaluation and combination of advanced language modeling techniques. In *Twelfth Annual Conference of the International Speech Communication Association*, 2011.

Hwee Tou Ng, Siew Mei Wu, Ted Briscoe, Christian Hadiwinoto, Raymond Hendy Susanto, and Christopher Bryant. The conll-2014 shared task on grammatical error correction. In *CoNLL Shared Task*, pages 1–14, 2014.

Nam Nguyen and Yunsong Guo. Comparisons of sequence labeling algorithms and extensions. In *Proceedings of the 24th international conference on Machine learning*, pages 681–688. ACM, 2007.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English gigaword. *Linguistic Data Consortium*, 2011.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

Michal Richter. Pokročilý korektor češtiny. Master's thesis, Charles University, the Czech Republic, 2010.

Michal Richter, Pavel Straňák, and Alexandr Rosen. Korektor-a system for contextual spell-checking and diacritics completion. In *COLING (Posters)*, pages 1019–1028, 2012.

Alla Rozovskaya, Kai-Wei Chang, Mark Sammons, Dan Roth, and Nizar Habash. The illinois-columbia system in the conll-2014 shared task. In *CoNLL Shared Task*, pages 34–42, 2014a.

Alla Rozovskaya, Dan Roth, and Vivek Srikumar. Correcting grammatical verb errors. In *EACL*, pages 358–367, 2014b.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

Raymond Hendy Susanto. *Systems Combination for Grammatical Error Correction*. PhD thesis, 2015.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Jakub Vrána. Obnovení diakritiky v českém textu. Master's thesis, Charles University, the Czech Republic, 2002.

John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Charagram: Embedding words and sentences via character n-grams. *arXiv preprint arXiv:1607.02789*, 2016.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Xiaoming Xi, Martin Chodorow, Michael Gamon, and Joel Tetreault. The utility of article and preposition error correction systems for english language learners: Feedback and assessment. *Language Testing*, 27(3):419–436, 2010.

Ziang Xie, Anand Avati, Naveen Arivazhagan, Dan Jurafsky, and Andrew Y Ng. Neural language correction with character-based attention. *arXiv preprint arXiv:1603.09727*, 2016.

Zheng Yuan and Ted Briscoe. Grammatical error correction using neural machine translation. In *Proceedings of NAACL-HLT*, pages 380–386, 2016.

Antonín Zrůstek. Doplňování diakritiky do českých textů s chybějící diakritikou. Master's thesis, Masaryk University, the Czech Republic, 2000.

List of Figures

List of Tables