Master's Thesis

Two-step OCR Post-correction with BERT and Neural Machine Translation models

Tanyu Tanev

Examiner: Prof. Dr. Hannah Bast
Advisers: Matthias Hertel

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

June 14th, 2022

Writing Period

14. 12. 2021 – 14. 6. 2022

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Prof. Dr. Frank Hutter

Advisers

Matthias Hertel

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

Optical Character Recognition (OCR) is a process, which transforms images (scans) of PDF documents into actual text. This allows, among other things, for historical documents to be digitalized, archived, and made available for researchers to use for further processing. Modern OCR systems, however, produce errors when reading the texts of historical documents due to a number of reasons — degraded quality of the document, archaic forms of words, unknown fonts and so on. In order to make up for this loss of information, Post-OCR correction can be employed to transform the erroneous text back into its original form.

This paper evaluates the performance of a "two-step" approach on the task of Post-OCR correction — using BERT for error detection and marking OCR errors — optionally with context tokens around them — and sequence-to-sequence models for correcting them. In particular, it researches how viable this approach is when compared to the tradition "one-step" sequence-to-sequence method, which involves just running a sequence-to-sequence model on parts of the whole sequence.

Furthermore, the paper also offers in-depth statistics of popular Post-OCR correction benchmark datasets, alongside introducing some new ones. A procedure is outlined, which can harness the aforementioned statistics, in order to build an arbitrary amount of artificial data.

Although the two-step approach massively improves on the baseline method, it achieves very underwhelming results when compared to the state-of-the-art models from wide-known competitions. This is due to the error detection part becoming the

bottleneck of the approach, essentially putting a cap on how good this approach can function based on how many samples the detection model actually marks right. The paper discusses the shortcomings of this approach and gives ideas on how it can be improved in the future.

Contents

1 Introduction

1.1 Motivation

Digitization of historical documents is an important ongoing task worldwide. According to a 2017 survey ([NvdHT17]), which fielded the responses of around 1000 cultural institutions, 82% of them were in active engagement with digitization. The situation is similar in the United States, where [MP17] shows that 75% of 769 surveyed libraries include digitization as a part of their activities. Furthermore, individual papers make mention of local digitization initiatives: [KD16] works with digitized documents, produced by "Arabic Press Archive"; [ADT19] introduces a massive dataset of digitized Swedish newspapers; [TMR09] evaluates the quality of another dataset, made from digitized 19-th century British newspapers; and [BDD+08] tackles the creation of a digitized collection of academic papers (i.e., journal and conference). This saturation of academic works on the topic speaks for its importance. The benefit of digitization is not only limited to preservation of culture and heritage. Rather, it also allows for these historical documents to be made available online for further research or educational usage. Indeed, [MSA+11] showcases how leveraging digitized books can be used to explore the evolution of the English language.

Despite the acute demand for digitization, the manual process of doing so is expensive. [NJCD21] points out that the estimated cost of manually typing out the contents of *one* page amounts to 1 Euro. Scaling this cost up to the volume of all documents that need digitization would result in a large financial investment. Because of this,

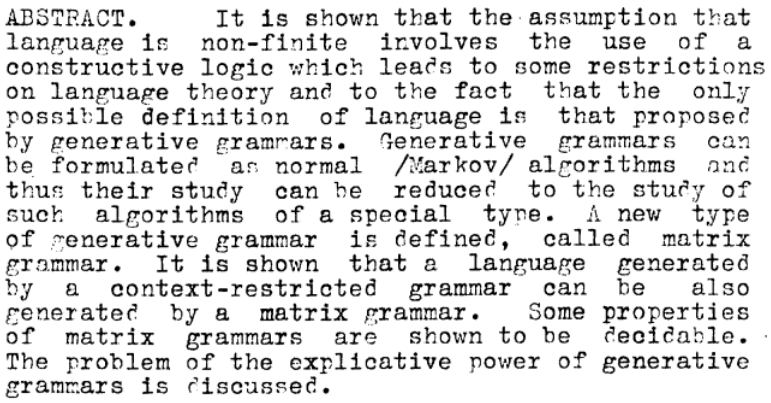
most digitization efforts are done automatically — a process called Optical Character Recognition.

1.2 Optical Character Recognition

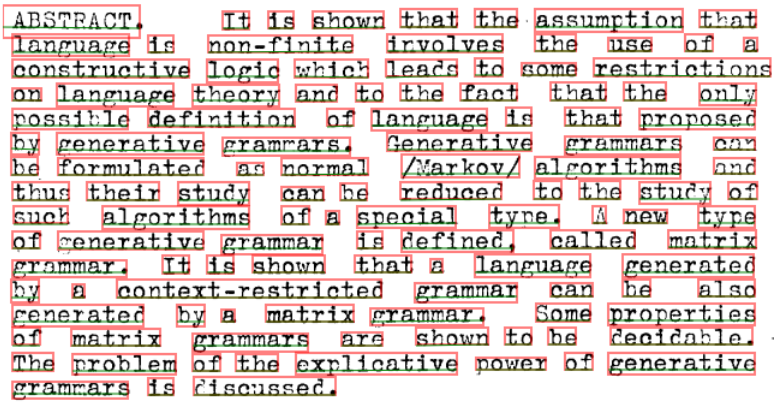
The most commonly-used OCR systems are software-based ([Ass16]). Tesseract OCR[1], for example, is a widespread solution, developed in part by Google[2]. Figure 1 then showcases how Tesseract and similar systems can be used to digitize documents[3]. The user provides an image — usually a scanned version of a document — to the system (as visualized in figure 1a). The system then "breaks down" the image into individual characters and tries to recreate the original text. Brought down to its simplest form, this process starts with the document hierarchically split down into more basic building blocks, e.g., paragraphs into sentences into words into letters. Afterwards, each image of a character (also called: **glyph**) is compared with different representations of letters, coming from various fonts. The most passing match is then chosen to represent the glyph in the resulting text. Finally, the letters are merged back together into words and sentences and returned; the result of this step is showcased in figure 1b. Modern OCR tools naturally include more sophisticated techniques ([Ass16]), including:

- Preprocessing the page to fix skew and/or problems with the aspect ratio
- Using computer vision to get a latent representation of the characters when they are being recognized; this avoids having to compare different letter representation on a pixel-by-pixel basis, which is heavily influenced by font characteristics
- Using a two-pass approach, where recognized characters with high confidence are used as context for the prediction of the rest

[1] `https://github.com/tesseract-ocr/tesseract`
[2] `https://developers.googleblog.com/2006/08/announcing-tesseract-ocr.html`
[3] Interactive tool is available at: `https://tesseract.projectnaptha.com/`



(a) Excerpt from article with its Abstract section



(b) Boundaries of all detected tokens from Tesseract OCR

ABSTRACT. It is shown that the assumption thrat language is non-finite involves the use of a constructive logic which leads to some restrictions on language theory and to the fact that the only rossitle definition of language is that proposec by generative gramrars. fGenerative grammars can be formulated asn normal /M¥arkov/ algorithms and thus their study can be reduced to the stufy of suck algorithms of a special +tyre. 4 new tyrpe of rsenerative grammar is defineé, called matrix grammar. It is shown that 2 languapge generated by a context-restricted grammar can be also generated by a matrix grammar. Some properties of matrix grammars are shown to be deecicable. The problem of the explicative power of generative granrmars is ciscussed.

(c) The resulting text reconstruction; red symbolizes mistakes

Figure 1: Result of applying Tesseract OCR on [Abr65]

Despite the continuous development, [SC19] shows that OCR engines still achieve sub-par results when used on historical documents. Figure 1c highlights all of Tesseract's mistakes when used on an excerpt of an article from 1965 ([Abr65]), provided by the ACL Anthology Reference Corpus ([BDD+08]). As observed, the mistakes are diverse, including **hallucination** of non-existing characters (*that* → *thrat*), **character substitutions** (*proposed* → *proposec*) or *complete mismatch* (*possible* → *rossitle*). There are two reasons, which immediately pop into mind to explain the OCR engines' worse performance on historical documents. For one, historical documents often suffer from degraded paper quality. Secondly, older documents may also contain archaic words, dialect forms of common words no longer in usage, or even words in a non-recognizable font. In the case of academical works, OCR is made even more difficult by the inclusion of technical jargon and mathematical symbols in formulas. All of these factors combined make it, as of current stand, impossible for OCR engines to perfectly reproduce text. But then how is it possible to keep up with the demand for digitization? The answer is **Post-OCR correction**.

1.3 Post-OCR correction

Post-OCR correction is the task of mapping erroneous text, produced by an OCR system, back to its original, error-free form. For example, if an OCR system produces the erroneous sequence *"I l0ve |)esto!"*, Post-OCR correction should map it back to *"I love pesto!"*. As discussed in the previous section, these mistakes include mapping non-letter combinations back to the English alphabet (e.g., *|)* to *p*), merging the blank space between two substrings (e.g., *car pet* → *carpet*), restoring hyphens of compound words (e.g., *viceversa* → *vice-versa*) or all together removing imagined tokens (e.g., *M¥arkov* → *Markov* from figure 1c).

Token is an important term in the field of Post-OCR correction. It is used to represent an **arbitrary sequence of characters**. This means that a token can take the form of a single character, a word, multiple words or even a single blank

space. I will be using token as a general term in this paper to represent sequences of characters, which need to be corrected. Any time I need to specify the exact nature of the token (e.g., word, character, etc.), I will explicitly mention it.

Historically, Post-OCR correction was achieved with the help of rule-based algorithms ([EP14]). These algorithms replaced non-English words with ones that are close to them, e.g., *desto -> pesto*. Rule-based algorithms can work well, but they also carry problems. Indeed, working with a big-enough **vocabulary** (also called: **dictionary**) to cover all words of a language is costly. Additionally, as we will see later in this paper, rule-based algorithms also struggle with **real-word errors**. A real-word error is a correctly-spelled word, which does not fit *contextually*: e.g., *I **dove** pesto!* $\rightarrow$ *I **love** pesto!*. A naive rule-based algorithm would not correct *dove* into *love*, as it is a correct English word, even though it does not make sense grammatically.

Nowadays, the field of Post-OCR correction has greatly benefited from advancements in artificial intelligence. Specifically, there is an area called **Natural Language Processing**, or **NLP**. Research in NLP focuses on being able to read, understand and work with *natural*, or human, language. One of the main subfields of NLP is called **Neural Machine Translation** - or **NMT** for short. NMT tackles the problem of translating text from one language to another automatically, like, e.g., Google Translate does. What makes NMT more difficult and interesting than just using a dictionary of words to translate a sentence is text **structure**. Consider the German sentence "*Heute spielen wir Tennis*" and its *literal* English translation "*Today are playing we tennis*". As we can see, the German sentence can not be translated word-for-word: different languages have different rules about sentence **structure** and how the elements in it should be ordered.
Modern approaches to Post-OCR correction re-frame the problem to an NMT problem ([AC18]). In it, the source "language", from which the AI models need to translate, is the language of OCR-ed, erroneous texts, and the target language is proper English. **Pure** NMT approaches — meaning approaches that do not use rule-based approaches

as pre- or post-processing — do not employ vocabularies with correct English words. Instead, they rely on the artificial intelligence to learn about the English language on its own, i.e., to learn what words it makes sense to use in the context of other ones. This allows NMT models to also start correcting real-word mistakes, as shown in [AC18].

Neural Machine Translation has been evolving rapidly in the last couple of years thanks to **LSTM** models ([HS97]) getting bigger and the invention of **attention** ([BCB15]). In a nutshell, this mechanism "forces" the AI to focus on its **context** before making a prediction, which greatly improves results. Its success was so big that it spawned an entirely new AI model - **Transformers**, which use attention as the corner stone of its operations ([VSP+17]). Transformers have been gaining traction since their introduction up to this day, where they are the state-of-the-art in many areas of Natural Language Processing.

Recent research (see [SN20] and [NJN+20]) has also suggested splitting the task of Post-OCR correction into its two individual parts — i.e., error *detection* and *correction* — and using separate NLP models for them. I will refer to this technique as the **two-step approach**. As the name implies, this approach to Post-OCR correction first applies an additional *detection* model to mark the boundaries of erroneous tokens, before passing them on to the *correction* model. In comparison, traditional usage of NMT models for Post-OCR correction has employed them doing *both* tasks at the same time (as shown in [AC18]). The motivation behind the two-step approach can be explained easily: having two *separate* models focus on error detection and correction should lead to a specialization of each model on the task in hand, thus leading to better results. The approach is further made desirable by the opportunity to use a powerful error detection model - **BERT**.
BERT ([DCLT18]) is a **language representation** model, built from elements of the aforementioned Transformer model. It employs *attention* to compute representations of words and sentences, which can then be used for further tasks — e.g., OCR error

detection. BERT is the current state-of-the-art on many Natural Language Processing tasks, and is thus a suitable candidate for an error detection model.

This paper examines the performance of the two-step approach with BERT as an error detection and different NMT models as **character-level** correction models. In particular, it studies the influence of the aforementioned attention mechanism and whether it is applicable on a lower level than words. This is also directly connected to an evaluation of **LSTM** and **Transformer** encoder-decoder (explained in Chapter 3) models. One of the main motivations behind this work was finding out if the Transformer architecture can be more successful than the LSTM one, as is the case with many fields of NLP. Additionally, the paper outlines a procedure for generating artificial Post-OCR correction data, and studies whether it can be used to further increase the performance on community benchmarks. With that being said, the main contributions of this paper are:

- Outlining a procedure for creating artificially many erroneous OCR samples, with the introduced errors mimicking the distribution of the real ones

- Comparing LSTM and Transformer sequence-to-sequence models on character-level Post-OCR correction

- Formalizing a new way to calculate information retrieval metrics on the task of character-level error correction by adapting an algorithm from the field of bioinformatics

- Suggests a new way to evaluate error detection on an **entity-level**, which gives more consideration to properly marking *word boundary* mistakes

2 Related Work

The field of OCR post-correction has been a subject of research for decades. Despite that, there have been, to my knowledge, only two major competitions, which can be used to survey general approaches and set the state-of-the-art. The two competitions were also fairly recent — held in 2017 and 2019 at the respective International Conference on Document Analysis and Recognition (ICDAR). As such, it is easier to track how the field has evolved in the last couple of years.
As for the long period before 2017, there is no official benchmark I can use to compare the most common approaches. The papers I will reference use their own custom benchmarks, specialized in a certain domain. Nonetheless, I will try to present the evolution of OCR post-correction naturally, with each paper reaching a new milestone up to the state-of-the-art. As a convention, I will be referring to all rule-based approaches as **classical** approaches.
In [NJCD21] from last year, Nguyen et al. provided an in-depth survey of the field of Post-OCR correction. The scope of their paper is bigger than this chapter on related work, and I encourage the reader to refer to it for a full picture of the evolution of the field.

Section 2.1 will introduce classical approaches for Post-OCR correction, which were wide-spread before the advent of Neural Machine Translation. Section 2.2 will then go on and look at the NMT approaches, which have reached new milestones and set the new state-of-the-art. The last section 2.3 goes into detail about which approaches I will be using as comparison bases for the work in this paper.

2.1 Classical Approaches

[EP14] uses a **dictionary-based** approach to correct OCR-ed texts of Argentinian army bulletins. For this, they first accumulated a rather big dictionary by mixing different sources — Wikipedia articles, lists with common people or geographical location names, the vocabulary of a pre-existing spell checker and words from the army bulletins themselves. Words from OCR-ed texts are then checked against the aforementioned dictionary, and those that are not found are considered to be mistakes. As mentioned by the authors themselves, this approach bars them from handling *real-word* mistakes, and they thus focus on only on **non-word** mistakes.
Correction candidates for the non-word mistakes are then proposed by collecting valid Spanish words with an edit distance of ≤ 2. The authors additionally employ an external algorithm to determine **equivalence classes** of characters. Every character of an equivalence class is considered to be interchangeable with all other ones. The combination of the edit distance candidate generation and the equivalence class candidate generation is then used to get all words from the dictionary, which can be used as valid corrections. The final correction is the one with the highest **frequency** - meaning the one, which was encountered the most times while building the dictionary. At the end, the authors achieve a 29% error improvement. The authors note that the biggest improvements were made by the inclusion of the lists of names, as well the **domain-specific** words — the ones, extracted from the bulletins themselves. We will see later on that the two conclusions are common notions in the field Post-OCR correction. *Named Entities* like personal or location names cause problem not only for dictionary-based approaches, but also NMT-based ones. This is due to the very nature of named entities — they follow different rules than standard words, but are still *correct*. The second idea is that Post-OCR correction is heavily influenced by the *domain*. While it may seem unnatural at first — many cultures around the world share alphabets — empirical evidence from this paper and ICDAR2019 later on suggests it is true.

Another classical approach is discussed in [TE96]. In it, the authors combine a dictionary $L = \{w_1, w_2, ..., w_m\}$ with **statistical machine translation** (or: **SMT**) for Post-OCR correction. The approach can easily be explained by defining it formally: let $S = s_1 \dots s_n$ be an erroneous OCR string and $W = w_1 \dots w_n$ be a correct word. Then, the most suitable correction — $\hat{W}$ — is:

$$\arg\max_{\hat{W}} P(W|S) = \arg\max_{\hat{W}} \prod_{i=1}^{n} p(w_i|w_{i-1}) * p(s_i|w_i) \tag{1}$$

In equation 1, the left side — $p(w_i|w_{i-1})$ — represents the probability of a correct word w_i appearing after the correct word w_{i-1}. This probability brings context into the Post-OCR correction by forcing the statistical language model to "look one to the left", before making its prediction. In the paper, the authors calculate this probability by building a probability table from the training data. Equation 2 is used for calculation, where $c(w_{i-1}, w_i)$ stands for the number of times the word sequence of w_{i-1}, followed by w_i, was found in the training data; and $c(w_{i-1})$ — the number of times the word w_i was found overall.

$$p(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})} \tag{2}$$

The right side of the equation — $p(s_i|w_i)$ — models the probability of a correct word w_i being recognized by the OCR as the erroneous word s_i. The process of calculating this probability is a bit more involved, and I will thus skip explaining it for the sake of brevity. In short, the authors reduce the problem to independent edit operations on character level, i.e., characters getting deleted, added or substituted. They then initialize the system with a uniform probability for every operation and do *multiple* passes. For each pass, the Post-OCR correction system corrects the text and then

refines its operation probabilities with the equations from 3:

$$\begin{aligned} p(y|x) &= num(sub(x,y))/num(x) \\ p(del(x)) &= num(del(x))/num(x) \\ p(ins(y)) &= num(ins(y))/num(\text{all letters}) \end{aligned} \tag{3}$$

At the end, the authors test out their approach in three different ways: *isolated non-word correction*, *context non-word correction* and *context non- and real-word correction*. Non-word correction first checks words in the vocabulary L and does not correct them, if present. Non- and real-word correction, on the other hand, treats every word as a mistake. The authors did not specify what the difference between *isolated* and *context* correction is. I assume *isolated* correction just sets the left part of equation 1 to a constant 1 — that way, the only probability that matters is the OCR correction itself.

The authors found context correction to work significantly better than isolated correction. Additionally, turning on real-word correction also provided a big boost to the performance of the system. Despite this approach introducing new cases of *overcorrection* — where the system "corrects" a correct word to a wrong one — the fraction of real-word errors that were corrected more than make up for them.

As for limitations, the authors again discuss problems with unrecognizable *named entities*. Moreover, as the system uses white space as a delimiter, it is also unable to handle OCR mistakes related to splitting or merging the characters of the word.

The Post-OCR correction competition at ICDAR2017 gives further insight on classical approaches. It also introduces NMT models into the scene and showcases how they compare with the state-of-the-art in 2017. An overview of the competition and the most significant approaches are described in [CDCM17]. It is worth noting that the authors themselves confirm that the competition dataset is very dirty, which may have hindered the performance of many of the proposed solutions.

Six out of the eleven evaluated submissions in [CDCM17] can be considered classical,

with one (Char-SMT/NMT) being an ensemble approach, which uses both statistical and neural machine translation. Five out of the six submissions use an approach, similar to [TE96], in that they use a vocabulary and statistical machine translation in their workflow. The two best-performing models on *both* tasks — error detection and correction — are based on statistical machine translation, which confirms its effectiveness. **WSFT-PostOCR** is the leader in error detection in the competition. It was created by its authors at the Centro de Estudios de la Real Academia Espanola and features **weighted finite-state transducers**, or WFST. What is interesting about this approach is that WFSTs draw many similarities to character-level NMT models. Even though the organizers of ICDAR2017 did not include a white paper on how the model functioned, we can use a similar paper to get an idea — [LNCPCA10]. In the paper, it is shown how multiple WFSTs can be essentially *combined* to emulate the task of Post-OCR correction. I find that it is worth mentioning, as the weighted finite-state transducers can be seen as a bridge between the areas of classical and NMT approaches, offering both interpretability and comparable performance for error detection. For the sake of brevity, however, and the relative obscurity of using WFSTs for Post-OCR correction, I will refrain from further discussion. The creators of the **Char-SMT/NMT** — Amrhein and Clematide — published [AC18] one year after the competition, in which they go into more details about their model. Their paper features evaluations of different configurations of *character-level* Statistical Machine Translation (**SMT**) and Neural Machine Translation (**NMT**) models.Among all experiments, there are a few significant ones I will point out:

The first experiment the authors carry out is comparing the performance of the models when using *multimodal* datasets. They achieve this by combining datasets across different languages (English and French) and publication types — periodical (i.e., newspapers, magazines) and monographs (i.e., books). The experiment is meant to evaluate if Post-OCR correction is *domain-specific*, or whether it gets better with multimodal data. Against the author's expectations, their evaluations show that mixing the different datasets result in a *drop* in performance. The drop is also shown to

"scale" with the degree of "mixing" — meaning that the drop from mixing documents of different languages is bigger than mixing documents of different publication types. This is the second piece of empirical evidence, which I talked about in the beginning of this section, indicating that Post-OCR correction is dependent on its domain.
The authors also test out the performance of the models when using *isolated* and *context* error correction. As seen before, including the surrounding words around the erroneous tokens improves the performance of *both* models. The intuitive explanation behind this, as explained by the authors in the paper as well, is that the context allows handling **real-word** errors.
Another significant addition to both models, tested out by the authors, is including *metadata* with the predictions. The authors call this **Factored** Neural Machine Translation. While this is shown to help in the case of English monograph documents, this method is not really feasible for usage in this paper, as this information is not always present.
Overall, the authors of Char-SMT/NMT conclude that NMT models perform better on error detection, while SMT remains the best choice for error correction.

2.2 NMT Approaches

ICDAR2017 also features a pure-NMT submission — Character-Level Attention Model, or **CLAM**. The model does not have a white paper attached, but the organizers share it is based on an LSTM sequence-to-sequence model with attention. The noteworthy aspect of the approach is that the authors test out using different sized *context windows*. At the end, they found out that using 4 preceding and 1 succeeding tokens works best for 3 of the ICDAR2017 datasets, while the last one requires 6 preceding and 1 succeeding.
As for how CLAM compares with the leading SMT models, it shows promise. In particular, it achieves the **third** best F1 score on error detection on the English monograph documents, as well as the **second** best % improvement on error correction.

Surprisingly, the performance of the model on the French datasets pales in comparison, with error correction only improving with 1% and 5% on monographs and periodicals respectively. Unfortunately, there is no information provided in the competition report ([CDCM17]) as to why this is. One of the possible explanations may be that the creators of CLAM trained the model on a *mixed* dataset, as it was not specified in the submission description that separate models were used. As seen in the previous section, Post-OCR Correction has empirical evidence to be *domain-specific*, which can hamper the performance of an NMT model.

Another competition on Post-OCR correction was held at ICDAR in 2019. As in the previous event, the organizers provided a summary of the competition, together with information about the data, all proposed submissions, and the evaluation results in [RDCM19]. Compared to the previous iteration, however, ICDAR2019 only features **five** unique approaches, with one submission entering two models (RAE1 and RAE2). Out of the five, two were also submitted to the prior competition in 2017. The team that created *WSFT-PostOCR* in 2017 again proposed a solution based on **weighted finite-state transducers** — RAE 1 and 2 in the paper. As in the previous iteration of the competition, their approach leads to competitive performance in the task of error detection, but they show their best results on error correction. Indeed, RAE1 and 2 achieve the second-best performance on error correction for all datasets they handled, except for one. **CLAM** — or Character-Level Attention Model — was also re-submitted to the competition and achieved the second-best results on error detection on eight out of the ten datasets. The real stand-out of the competition, however, is **CCC**.

CCC is the top-scoring approach on *all* error detection and 8 out of ten error correction datasets. From the provided description, CCC seems to be a **two-step** pipeline. First, a **BERT** model is used for error detection by outputting binary predictions — i.e., a 0 or a 1 — for every erroneous or correct token in a sequence. All tokens, which are determined by BERT to be erroneous, are then passed on to an error correction model — the second step of the pipeline. The correction model itself is also an NMT model

— a character-level sequence-to-sequence model, based on LSTM. The description of the approach also mentions that "context information" is also passed from BERT to the error correction model as input, but there is no further elaboration. What is also worth mentioning is that the BERT model, used for error detection, was a pretrained **multilingual** model. This is significant, as it goes against the previously established empirical evidence of Post-OCR correction being *domain-specific*.

Since ICDAR2019, more papers have been published with an evaluation of a *two-step* approach. In one of them — [SN20] — the authors compare the performance of two types of NMT-based Post-OCR correction approaches on a dataset of pre-19th century German texts. The first approach employs a sequence-to-sequence network to directly detect *and* correct errors in the OCR texts, similar to how it is shown in [AC18]. The second approach, however, splits the pipeline into *two* - error detection, **then** error correction. In particular, the error correction model is only ran on tokens, which the error detection model predicted to be erroneous. The motivation behind this split is two-fold. First, it should reduce the amount of correct characters that end up getting wrongfully changed by the error correction model. Second, by design of the pipeline, the error correction model is exposed to more erroneous tokens than correct ones, which should positively influence its performance. Indeed, the authors show both statements to be true — the "**two-step**" approach achieves a relative improvement of 18.2%, while the "**one-step**" approach actually *increases* the error rate of the documents. Moreover, the percent of wrongly substituted *correct* characters is heavily decreased from6 6 to 0.3%.

The second paper — [NJN+20] — was published by the same author that published the survey paper from the beginning of this chapter. In it, the authors describe a two-step approach, akin to CCC from the ICDAR2019 competition, but with minor improvements for both the detection and correction models. Namely, they propose a simpler BERT model, which utilizes pretrained **word embeddings**. With those adjustments, they reach a better F1 score than all models from ICDAR2019 on its

English dataset, as well as on the English periodical publications from ICDAF2017. The discrepancy in the case of the English monograph dataset is explained by the authors by the larger percentage of *non-word* errors than *real-word* ones. It is argued that BERT can better detect real-word errors, based on the fact that it heavily relies on *context*.

As for the correction part of the pipeline, the authors evaluate three enhancements. The first one is embedding metadata about the dataset, which a particular data sample belongs to. The authors noted that this was inspired by the work of Amrhein et al. in [AC18]. The second suggestion involves creating character-level *aligned embeddings*. And the last — using a *length filter* to remove any correction candidates with an edit distance lower than 3. The authors claim their last proposal was based on previous academical work, which suggests that data exploration by the authors, which suggested that "more than 80% of OCR errors have an edit distance less than 3". The approach achieved moderate results on the English datasets of ICDAR 2017 and 2019, being beaten out by some SMT models.

2.3 A Note on Comparison Bases

As discussed in the beginning of this section, the field of OCR post-correction is still fairly unorganized. There is no universally accepted gold standard, which can be used to measure progress of new approaches, akin to ImageNet in Computer Vision ([DDS+09]). ICDAR2017 and 2019 tried to remedy that by holding their respective competitions on Post-OCR correction, where they provide novel datasets to be used for training and evaluation. As admitted by the organizers of the events themselves, however, the datasets are very dirty, featuring problems with misaligned sequences, erroneous or entire missing ground truths.

Furthermore, I find another problem in the field of Post-OCR correction is the difficult-to-reproduce approaches. For example, I could not find the source code of the two leaders in the ICDAR2017 and 2019 competitions — *Char-SMT/NMT* and

CCC. It is true that the authors of Char-SMT/NMT published a paper about their approach, but it is also a labor-intensive process to recreate.

Despite the aforementioned problems, I will be using the results from the competitions at ICDAR2017 and 2019 as baselines for this paper. As I will be working with the English language exclusively, I will be referring to the evaluation results presented in [NJN+20]. This also allows me to compare my approaches with the ones proposed in the aforementioned paper.
The organizers of both ICDAR competitions provide an evaluation script[1]. I will, however, not be using it when evaluating the performance of the models in this paper, as time did not permit me to adapt my own evaluation technique to work with the provided script. In particular, this was made difficult because of the way error detection samples are created in this paper — explained later on in subsection 7.1.1 and 8.1 — which involves splitting the ICDAR samples into smaller sentences. This, in turn, makes reconstructing the starting offsets of the target tokens — the format which the ICDAR evaluation script expects — difficult to implement. This is a problem, which some participants also had during the ICDAR2017 competition, as revealed in [CDCM17]. That being said, I will try to come as close as I can to the evaluation method, described in both papers, and will thus be skipping any erroneous tokens with *hyphens* in them, as well as any tokens with misaligned ground truths, as per the advice of the authors.

I will also be using two external approaches as additional baselines. The first one is directly inspired by the master's thesis of my supervisor — Matthias Hertel — from his work on using language models and NMT models for spelling correction ([Her19]). In particular, I will use Google as a Post-OCR correction engine by manually picking a subset of all test samples, posting them in a Google Docs document, and correcting them until Google has no more proposals.
The second external approach I will use is taken from [HH19]. It[2] uses a character-level

[1] Available on `https://gitlab.univ-lr.fr/crigau02/icdar2019-post-ocr-text-correction-competition`
[2] The code is offered as a Python package under: `https://github.com/mikahama/natas`

NMT model to propose 10 corrections of erroneous tokens, each one with a specific probability. The most probable correction that is also a proper English word is then chosen to be the final one. For error detection, the model uses a dictionary-based approach.

3 Background

This chapter will introduce the technical concepts, which are required to be able to follow the paper's approach. Section 3.1 explains the basics of machine learning. I assume the reader has a basic understanding of machine learning and will therefore keep the section brief. Section 3.2 will focus on deep learning and delve into the mechanics of neural networks. Finally, section 3.3 introduces the concept of recurrent neural networks, as well as Long Short-Term Memory (LSTM) cells.

3.1 Machine Learning

Machine learning is a subfield of artificial intelligence. It focuses on algorithms, which "learn" to solve problems on their own by building *mapping functions* from inputs to expected outputs. Formally, if we let X be a collection of input samples and y a collection of expected outputs, then a machine learning algorithms learns f, such that:

$$y = f(x) \tag{4}$$

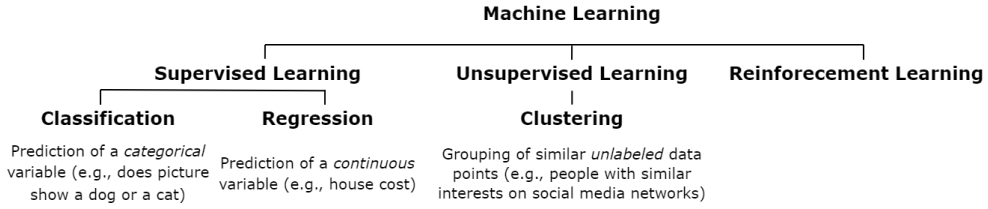


Figure 2: Branches and tasks of machine learning

The mapping function f from equation 4 is, in most cases, *not analytically computable*. Therefore, the **parameters** of the algorithm (also called: **weights**) θ are *iteratively* improved. For a more in-depth explanation of how the weights are iteratively made better — the same process, which is called *learning* — I refer the reader to [GBC16]. Depending on the format and availability of the data, machine learning can be split into supervised, unsupervised and reinforcement learning (as shown in fig. 2). Unsupervised and reinforcement learning are not used in the boundaries of this work, and will thus not be discussed further.

Supervised learning is observed when the dataset, which is given to a machine learning algorithm to learn from (also called **training** dataset), contains the *expected output value* for each input sample. A trivial example of this is a dataset, which maps integers to their successor: $(1 \to 2)$, $(2 \to 3)$, ... In practice, supervised datasets are more complex, e.g., the widely-known "Boston Housing" dataset, first published in [HR78].
The presence of expected outputs allows the *model* (another term for machine learning algorithm) to compare its predictions with the targets and, based on how wrong it was, update its parameters to do better in the next iteration. Different algorithms exist to handle data according to its complexity. There exist simple models like **linear regression**, which assume that the underlying mapping f is **linear**. Linear mappings (or **functions**) are all mappings, which can be expressed by equation 5. For reference, the trivial dataset from above with the successive integers *is* an example of such a linear function.

$$y = f(x) = mx + b \tag{5}$$

Linear models (meaning models, which assume a linear dependency) can work well for simple problems, but become unsuitable when it comes to complex, higher-dimensional data. Non-linear models like support vector machines [Hea98], random forests [Bre01] or artificial neural networks [MP43] can be applied instead to get better mapping approximations, albeit at the cost of interpretability. Neural networks have become the

most popular approach in the last decade because of their robust nature, empirically good results and property of being universal function approximators under certain conditions [Cyb89]. Their study is also known as deep learning and will be reviewed in the following subsection.

Supervised learning itself can also be classified into further subgroups, depending on the task. The two most common ones are **classification** and **regression**. Regression tasks are characterized by the prediction of a *continuous* numerical value. Both datasets from the previous paragraph — the successive integer example dataset, and the Boston Housing dataset — can be used for regression tasks. In the case of the Boston Housing dataset, a machine learning model can learn to predict the *price* of a house, based on its features.
Classification, on the other hand, handles prediction of *categorical* variables. Categorical variables only allow a certain set of values, e.g., Yes/No (**binary** classification) or Cat/Lion/Tiger (**multi-class** classification). Linear regression can be adapted to handle classification — the algorithm is called **logistic** regression ([Cox58]). As with regression, however, complex datasets need non-linear models. More importantly, multi-class classification can also be used to generate text. This will be shown in the upcoming chapter 4 on Neural Machine Translation.

3.2 Deep Learning

Deep learning is a subfield of machine learning, which focuses on "deep" neural networks. Figure 3 shows what a neural network consists of in its simplest form — layers of *neurons*, which do non-linear computations to map a set of inputs to expected outputs. Every layer, except for the input and output layers, is called *hidden*, as the user does not have information on what data exactly goes in and out of them. Deep learning is a loose term and encapsulates *every* neural network, which has *more than one* hidden layer.

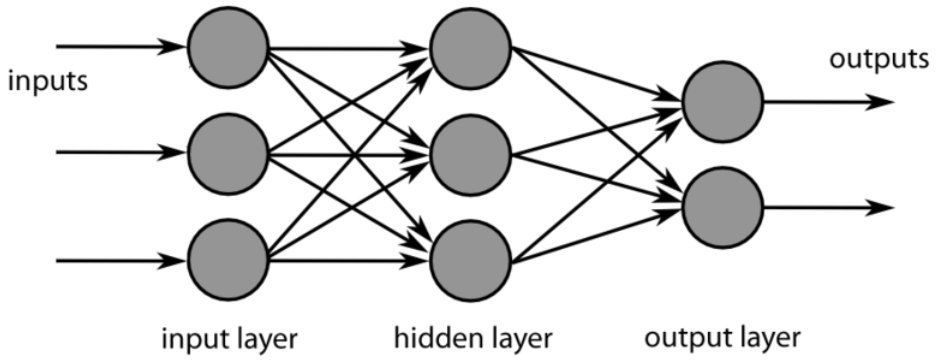


Figure 3: Feed-forward neural network [Com10]

Formally, the computations within a neural network are described as such:
Let there be K hidden layers in the neural network with M neurons each. Every layer i is then *fully-connected* with the previous layer $i-1$ by a **weight matrix** W^{i-1}, where $w_{x,y}^{i-1}$ connects the *x-th* neuron from layer $i-1$ to the *y-th* neuron of layer i:

$$W^{i-1} = \begin{pmatrix} w_{0,0}^{i-1} & \dots & w_{0,M}^{i-1} \\ \vdots & \ddots & \vdots \\ w_{M,0}^{i-1} & \dots & w_{M,M}^{i-1} \end{pmatrix}$$

Additionally, every neuron (except the ones in the input layer) has a **bias** b. Collectively, all biases in hidden layer i are expressed with the vector $\boldsymbol{b}^i$, and the bias of a neuron j in layer i is b_j^i. Bold variables are often used in machine learning literature to indicate **vectors**.
As a convention, the *input* to neuron j in layer i (except input layer) is notated with z_j^i. The output, on the other hand, as a_j^i.

Now, let $\boldsymbol{X} = \{x_1, x_2, \dots, x_n\}$ be a vector of N input values, and $\boldsymbol{y} = \{y_1, y_2, \dots, y_n$ be a vector of N corresponding expected outputs. The nodes in the input layer of the neural networks, as shown in Figure 3, are not neurons like the others. They only hold the input samples $\{x_1, x_2, \dots, x_n\}$. Then, the vector of input values $\boldsymbol{z}_i$ to all

neurons in the *i-th* hidden layer is:

$$
\begin{aligned}
\boldsymbol{z}^i &= W^{i-1} * \boldsymbol{a}^{i-1} + \boldsymbol{b}^i \\
&= \begin{pmatrix} w_{0,0}^{i-1} & \dots & w_{0,M}^{i-1} \\ \vdots & \ddots & \vdots \\ w_{M,0}^{i-1} & \dots & w_{M,M}^{i-1} \end{pmatrix} * \begin{bmatrix} a_0^{i-1} \\ a_1^{i-1} \\ \vdots \\ a_M^{i-1} \end{bmatrix} + \begin{bmatrix} b_0^{i-1} \\ b_1^{i-1} \\ \vdots \\ b_M^{i-1} \end{bmatrix} \\
&= \begin{bmatrix} w_{0,0}^{i-1} * a_0^{i-1} + \dots + w_{0,M}^{i-1} * a_M^{i-1} \\ w_{1,0}^{i-1} * a_0^{i-1} + \dots + w_{1,M}^{i-1} * a_M^{i-1} \\ \vdots \\ w_{M,0}^{i-1} * a_0^{i-1} + \dots + w_{M,M}^{i-1} * a_M^{i-1} \end{bmatrix} + \begin{bmatrix} b_0^{i-1} \\ b_1^{i-1} \\ \vdots \\ b_M^{i-1} \end{bmatrix}
\end{aligned}
\tag{6}
$$

As seen in equation 6, the input to each neuron is nothing more than a weighted sum of all outputs from the previous layer, plus a bias value. In order to get the output a_j^i of neuron j in layer i, the input of the neuron is additionally put through an **activation** function g, as show in equation 7.

$$
a^i = g(z^i) \tag{7}
$$

This is repeated for every layer (hidden and output), and in the end the neural network returns a set of outputs $\{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n\}$. This output set is then fed into a **loss function**, which calculates the mean loss value of the model's outputs. Afterwards, the loss value is *propagated* through the network in a process called **backpropagation**, which determines the influence of every parameter (weights and biases) on the model output. The parameters of the models are then adjusted, so that the model can make better predictions in the next iteration (same **learning** process as in machine learning). All of these concepts, together with activation functions, will be discussed in the upcoming subsections.

Activation Functions

Activation functions are the source of *non-linearity* in neural networks. They take some numerical input x and map it to a new range, which represents its perceived "importance". All hidden layers of a neural network use activation functions, as well as the output layer. The three most commonly used activation functions for *hidden* layers are:

$$\text{Sigmoid: } g(x) = \frac{1}{1+e^{-x}}; \quad g'(x) = g(x)(1-g(x)) \tag{8}$$

$$\text{Hyperbolic Tangent: } g(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}; \quad g'(x) = 1-g(x)^2 \tag{9}$$

$$\text{ReLU: } g(x) = max(0,x); \quad g'(x) = \begin{cases} 1 & \text{if x} > 0 \\ 0 & \text{if x} < 0 \\ \text{undefined} & \text{if x} == 0 \end{cases} \tag{10}$$

It is important to the training process of a neural network that the chosen activation function(s) (in practice: same one is used in every neuron) is **differentiable**. In the case of the Sigmoid (eq. (8)) and Hyperbolic Tangent (eq. (9)) functions, the derivatives are well-defined. ReLU, however, is not defined at $x = 0$ (eq. (10)). This limitation is usually circumvented by deep learning libraries by setting it to 0.

The activation function of the output layer is dependent on the machine learning task. In *regression*, for example, the activation function can be skipped (more specifically — set to the identity function $g(x) = x$), as the expected result is a continuous numerical value. In the case of *classification*, **sigmoid** is used for binary classification, and **softmax** - for multiclass. The formula for softmax ([Bri90]) with m different

classification classes and an output vector $\hat{\boldsymbol{y}} = \{\hat{y}_1, \ldots, \hat{y}_m\}$ is shown in equation 11. In general, both the sigmoid and the softmax functions map the output values of the network to a probability distribution in the range (0, 1).

$$softmax(\hat{\boldsymbol{y}})_i = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{m} e^{\hat{y}_j}} \tag{11}$$

For an example of what softmax does to a network's output, consider the example output vector $\hat{\boldsymbol{y}}$ = {-0.27, 1.54, 0.68}. When through softmax, the values of the three elements become the following:

$$\begin{aligned}
softmax(\hat{\boldsymbol{y}})_1 &= \frac{e^{-0.27}}{(e^{-0.27} + e^{1.54} + e^{0.68})} \approx \frac{0.763379}{7.401847} \approx 0.103133 \\
softmax(\hat{\boldsymbol{y}})_2 &= \frac{e^{1.54}}{(e^{-0.27} + e^{1.54} + e^{0.68})} \approx \frac{4.664590}{7.401847} \approx 0.630193 \\
softmax(\hat{\boldsymbol{y}})_3 &= \frac{e^{0.68}}{(e^{-0.27} + e^{1.54} + e^{0.68})} \approx \frac{1.973878}{7.401847} \approx 0.266674
\end{aligned}$$

Loss Functions

Loss functions calculate how close a model's predictions are to the expected targets. They are heavily-connected to the machine learning task and can heavily influence if the learning progression of the model. Given a prediction vector $\hat{\boldsymbol{y}}$ and its corresponding target vector $\boldsymbol{y}$, n amount of samples and m classification classes, two examples of loss functions are shown immediately below.

Equation 12 showcases **Mean Squared Error** (also: **MSE**) — an error function, which can be used with regression tasks. It computes the difference between each pair of outputs and model prediction, and squares them to *exaggerate* big errors. At the end, all the differences are summed up, and their mean value is returned.

$$\text{Mean Squared Error: } \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{12}$$

Equation 13 visualizes **multiclass Cross-Entropy** loss ([Goo52]). Cross-Entropy is used with *categorical* variables. It assigns high loss values to wrong predictions with high probabilities, and vice-versa. This can easily be seen in equation 13 itself:
y_{ij} is an **indicator variable**. It is set to 1 whenever the excepted output class of training sample i is class j, and 0 in all other cases. The right part of the equation under the sums - $log(\hat{y}_{ij})$ - assigns high loss values to low probabilities (e.g., $log(0.0001) = -4$) and low loss values to high probabilities (e.g., $log(0.9999) \approx -0.00004$).

$$\text{Cross Entropy: } \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} log(\hat{y}_{ij}) \tag{13}$$

Loss functions, similar to activation functions, have to be **differentiable**, in order for neural networks to be able to optimize with them. The derivative of MSE with respect to the network output $\hat{\boldsymbol{y}}$, for example, is shown in equation 14:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} &= \frac{\partial}{\partial \hat{\boldsymbol{y}}} \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial \hat{y}_i} (y_i - \hat{y}_i)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} 2(y_i - \hat{y}_i) \\
&= \frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)
\end{aligned} \tag{14}$$

More relevant to this paper is the derivation of *cross-entropy*. In particular, the derivative of cross-entropy with respect to a **softmax** output vector $\hat{\boldsymbol{y}}$. The operations for computing that specific derivative is harder due to the nature of the *softmax* operation, and thus I refer the reader to a wonderful guide, which covers that in more depth — [Ben16]. The final derivative is presented in equation 15:

$$\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} = \hat{\boldsymbol{y}} - \boldsymbol{y} \tag{15}$$

Backpropagation

Backpropagation is an algorithm for training neural networks, first proposed by Rumelhart et al. in [RHW86]. Intuitively explained, the algorithm calculates how big of an *impact* a specific parameter had on the resulting output loss. If, for example, the output loss is big, and the impact of a weight between two neurons "influenced" the result, then backpropagation adjusts the weight to do better the next time the neural network makes a prediction.
Formally, backpropagation works on the basis of the **chain rule**. I will first show an example with the chain rule to further build intuition and then continue with formally defining it. Let y and z be two example functions, as shown in equation 16:

$$\begin{aligned} y &= 6x + 9 \\ z &= 2^y \end{aligned} \tag{16}$$

Then, the derivative of z with respect to x can be calculated as follows:

$$\begin{aligned} \frac{\partial z}{\partial x} &= \frac{\partial z}{\partial y} * \frac{\partial y}{\partial x} \\ &= (ln(2) * 2^y) * 6 \\ &= 3 * ln(2) * 2^{6x+10} \end{aligned}$$

Backpropagation uses the chain rule to compute the derivative of the loss value with respect to all the neural network's parameters — weights and biases. Formally, the chain rule "reduces" the computation of the derivatives of different elements of the neural network to equations 17 through 20. The extended computations of the derivatives are skipped for the sake of brevity and can be seen in Chapter 6 of

[GBC16].

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^i} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^i} * \frac{\partial g(\boldsymbol{z}^i)}{\partial \boldsymbol{z}^i} \tag{17}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^i} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{i-1}} * W^{i-1} \tag{18}$$

$$\frac{\partial \mathcal{L}}{\partial W^i} = \boldsymbol{a^{i-1}} * \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^i} \tag{19}$$

$$\frac{\partial \mathcal{L}}{\partial b^i} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^i} \tag{20}$$

Optimizers

Optimizers leverage the calculated gradients (synonym for **derivatives**), calculated in the last subsection, to adjust the parameters of the neural network. Due to the nature of derivations, the calculated gradients are *guaranteed* to be pointing in the direction of minimal loss (also discussed in [GBC16]). Optimizers use this fact and employ different strategies, in order to adjust the network's parameters.

The simplest optimizer is called **Gradient descent** ([Rud16]), and it optimizes a given weight $w^i_{k,l}$ by using a user-defined (also called: **hyperparameter**) *learning rate* η, as shown in equation 21:

$$w^i_{k,l} = w^i_{k,l} - \eta * \frac{\partial \mathcal{L}}{\partial w^i_{k,l}} \tag{21}$$

Gradient descent has weaknesses, which are discussed in-depth in [Rud16]. In particular, the success of the optimization is very dependent on the learning rate η — too low and the process might *never* **converge** to a minimum, too high and the process might **diverge**. [Rud16] also lists improved optimizers, which try to solve the aforementioned problem *automatically*. In particular, the improved algorithms try to keep track of the gradient values from previous iterations and integrate them as **momentum**.

One of the most widely-used optimizers is **Adam**, first published in [KB15]. It is an *adaptive* learning rate method, which uses a running estimate for a gradient's first

and second derivatives to adjust the learning rate. Formally, if we set the estimate of the first derivative to be m and the estimate of the second derivative to be v, then Adam uses equation 22 to update a network's parameters:

$$w_{k,l}^{i} = w_{k,l}^{i} - \eta * \frac{m}{\sqrt{v} + \epsilon}, \text{ with } \epsilon = 10^{-8} \tag{22}$$

Schedulers

In addition to adaptive learning rate methods, one can also control the training process by using **learning-rate schedulers**. A *learning-rate scheduler* adjusts the learning rate η during the training process according to a pre-established formula. As a convention, learning-rate schedulers which *reduce* the learning rate are also called **decays**. For example, a really common scheduler is the so-called **linear decay**, shown in equation 23. Linear decay takes two parameters: η_{init}, which is the initial learning rate value, and a decay value γ. Like linear decay, other learning-rate schedulers commonly add additional *hyperparameters* for optimization.

$$\eta_t = \eta_{init} - (t * \gamma) \tag{23}$$

3.3 Recurrent Neural Networks

Recurrent neural networks (or **RNN**) are special types of neural networks, specialized in working with *sequence* data. [She20] provides a very thorough look into the computations that go on in recurrent neural networks, in addition to one of its most widely-spread varieties — Long Short-Term Memory (or **LSTM**) networks. In this subsection, I will also provide a brief overview of the *base* RNN and LSTM, as well as go over how those networks train with **Backpropagation Through Time**.

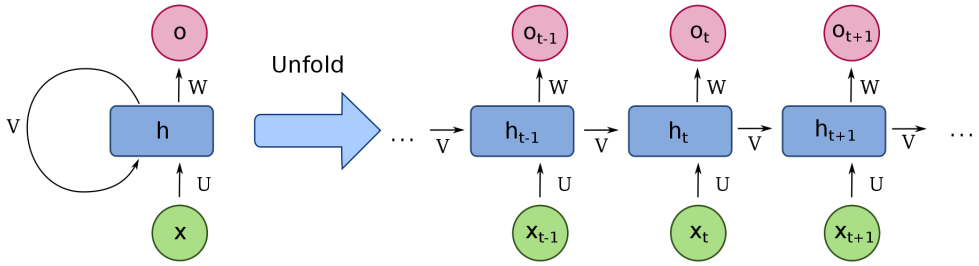


Figure 4: Architecture of a Recurrent Neural Network [Com17]

Figure 4 visualizes the architecture of a base RNN. $\boldsymbol{X}$ represents the input sequence, which can be segmented into individual **time steps** $\{x_1, \dots x_t \dots x_n\}$. Time step is an umbrella term when working with sequence data, and just represent the value of a certain element in the sequence. If we use *text* as sequence data, for example, time step i would be the same thing as *character* $\#i$ in the text. Similarly to $\boldsymbol{X}$, $\boldsymbol{o}$ also represent a vector, made up from the model predictions for the individual time steps $\{o_1, \dots o_t \dots o_n\}$.

The main difference between recurrent and feed-forward neural networks is that recurrence provides a better way to include **context** in a model's predictions. To explain this with an example, consider the task of completing the token *carpe*. By using information about all processed characters until the last *time step*, it becomes much easier to predict the letter *t*, as this completes the word *carpet*.

The blue box, marked with h on the left side of figure 4 is called a **recurrent**, or a **hidden** cell. The blue boxes on the right, marked with h_{t-1}, h_t and h_{t+1}, represent the cell being used for the different time steps (but in all time steps it is the **same** recurrent cell). In reality, the hidden *cell* is an abstraction for a set of linear algebra operations, as was the case with neurons in feed-forward neural networks. Figure 5 inspects what is "inside" a regular RNN cell. Two inputs go in — the input at time step t x_t, and the **hidden state** from the previous time step h_{t-1}. x_t is then multiplied with the weight matrix U, and h_{t-1} - with the weight matrix V, as shown

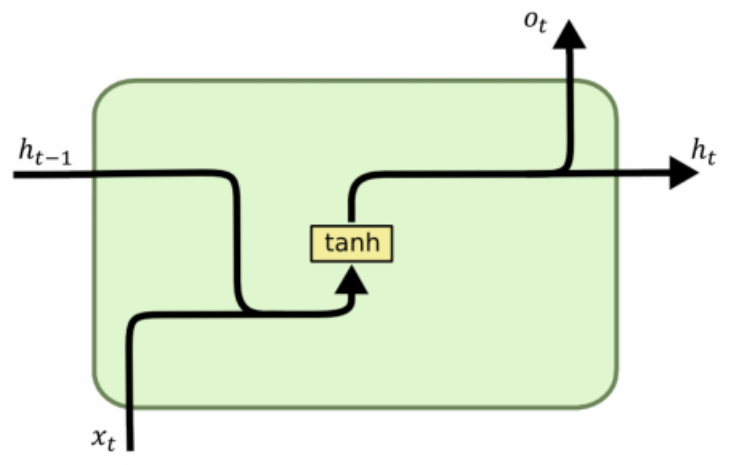


Figure 5: Visualization of vanilla RNN cell; design taken from [Ola15]

in figure 4. The result z_t is put through an activation function (in the case of figure 5 - hyperbolic tangent). The result of the activation function is then returned **twice** - once as o_t to be multiplied by the weight matrix W (see figure 4) and produce the prediction for time step t, and once as h_t to be used in the next time step.
Formally, recurrent neural networks use equation 24 to pass information through. I use a similar notation as with neural networks: $\boldsymbol{X}$ is the input vector, $\boldsymbol{z}$ is the vector of input values to the hidden cells $\boldsymbol{h}$, g is the *activation function* for producing the output of the hidden cells, and $\boldsymbol{o}$ is the vector of output values from the hidden cells, which also double-up as the final model predictions. The hidden state h_0 is usually initialized randomly or set to contain all zeros.

$$\boldsymbol{o}^t = W * g(U\boldsymbol{x}^t + Vh^{t-1}) \tag{24}$$

Backpropagation Through Time

Backpropagation Through Time (or **BPTT**), explained in [Wer90], is an adapted version of the backpropagation algorithm for use with recurrent neural networks. BPTT is harder to write out analytically, but shares the same intuition as normal

backpropagation. Namely, the gradient calculations follow the same rules as regular backpropagation (shown in rules 17 through 20), but they are **averaged** over all T time steps. As an example, differentiating the loss with respect to the output at time step t of the recurrent neural network is shown in equation 25:

$$\frac{\partial \mathcal{L}}{\partial o_t} = \frac{1}{t} * \frac{\partial \mathcal{L}}{\partial o_t} \tag{25}$$

The only "new" weights, which the recurrent neural network introduces are V for the hidden states $\boldsymbol{h}$ between time steps. The unfolded version of their gradient computation is introduced in equation 26. Readers can refer to [Wer90] for the full explanation of the gradient equations.

$$\frac{\partial \mathcal{L}}{\partial V} = \frac{1}{T} * \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial a_t} * \frac{\partial a_t}{\partial z_t} * \frac{\partial z_t}{\partial V} \tag{26}$$

Executing BPTT can be a very expensive process with higher values of T, which makes it hard to use. In practice, a modified version of BPTT is used, aptly called **Truncated** Backpropagation Through Time. This algorithm was first proposed in [WP98] and modifies BPTT to only us τ time steps, before averaging the gradients and optimizing them. Truncated BPTT works well in practice, but it comes with the caveat that the RNN puts more focus on *short-term* dependencies.

3.3.1 Long Short-Term Memory

Hochreiter et al. show in [HBFS01] that base recurrent neural networks do not work well in practice. They explain that the learning algorithms (including BPTT) result in **unstable** gradients. The process of multiplying gradients over T time steps can lead to both extremes of their values getting either too big — so-called **exploding** gradients, or too small — so-called **exploding** gradients. Hochreiter argues that a novel proposal, called **Long Short-Term Memory** (or **LSTM**), achieves better empirical results and solves the problem of *vanishing* gradients.

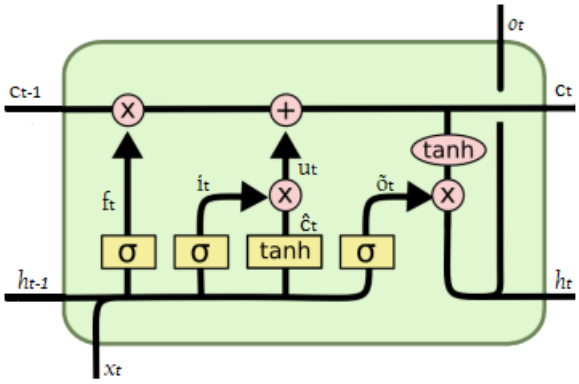


Figure 6: Visualization of LSTM cell; design taken from [Ola15]
red circles indicate *point-wise operations*
yellow blocks indicate *layers*

It is important to understand that the LSTM "only" changes the architecture of the blue *cells* from figure 4. In order to keep up with the notation, I will henceforth be referring to hidden cells that use the LSTM design as **LSTM cells**, and to recurrent neural networks, which use LSTM *cells* in them, as **LSTM networks.**
Figure 6 visualizes the architecture of an LSTM cell. Unlike a vanilla RNN cell, the LSTM cell accepts *three* inputs: the input at time step t x_t, the **hidden state** from last time step $t-1$ h_{t-1}, and the **cell state** from last time step $t-1$ c_{t-1}. Similar to the RNN cell, the output at time step t o_t is a *copy* of the hidden state h_t. In addition to the two regular outputs, the LSTM cells also outputs c_t, which is the new *cell state* of the recurrent network.

[Ola15] provides a wonderful intuitive explanation of the mechanics of an LSTM cell. Essentially, LSTM cells are capable of *remembering* more long-term dependencies because of the *cell state*. The cell state, in comparison to the hidden state, is only modified by *linear*, point-wise operations — marked by the red circles in figure 6. A big part of the rest of the LSTM structure is dedicated *layers*, which influence the modification of the cell state. In the literature, they are referred to as the **forget gate**

(marked with f_t on figure 6), the **input gate** (marked with i_t), and the **candidate gate** (marked with $\tilde{c}_t$).

The function for calculating the value of the **forget gate** is shown in equation 27. The $[x_t, h_{t-1}]$ operator in the function signalizes **concatenation** of the two vectors. It is called the "forget gate", because it inspects the input at time step t and *decides* what part of the context is no longer needed. As an example, if we are using LSTM cells to copy sentences character by character, a *white space* might be a good indicator that a new word is about to begin, and the last one should be partially ignored. Formally, this is done by outputting a normalized probability in the range of (0, 1) for every element of the cell state. This way, the influence of elements, which encode "old" information, can be mildened.

The **input** and **candidate** gates are combined to *update* the cell state. For this, the "input gate" first determines the positions of the elements in the cell state, which have to be updated (equation 28). The "candidate gate", on the other hand, calculates new values for the elements in the hidden state (equation 29). The elements of the two outputs — i_t and $\tilde{c}_t$ respectively — are multiplied point-wise to create the final *update vector* u_t (equation 30).

Finally, the cell state vector c_t is calculated with equation 31. The old cell state c_{t-1} is first multiplied point-wise with the output of the forget gate f_t to erase unneeded context information. Then its elements are added point-wise with the update vector u_t to carry in the new, relevant context information.

$$f_t = \sigma(W_f * [x_t, h_{t-1}]) \tag{27}$$

$$i_t = \sigma(W_i * [x_t, h_{t-1}]) \tag{28}$$

$$\tilde{c}_t = tanh(W_{\tilde{c}} * [x_t, h_{t-1}]) \tag{29}$$

$$u_t = i_t \odot \tilde{c}_t \tag{30}$$

$$c_t = (c_{t-1} \odot f_t) + u_t \tag{31}$$

The output of the LSTM cell at time step t o_t (which is a copy of the new hidden state h_t) is described in [Ola15] as a "filtered version of the cell state". The *filtering* process is done by the **output gate**, which uses the same mechanism as the forget gate. The "output gate" inspects the concatenated version of the input and previous hidden state and decides what part of the new *cell state* to return (marked as $\tilde{o}_t$ on figure 6). Formally, the output gate does this by computing (equation 32):

$$\tilde{o}_t = \sigma(W_{\tilde{o}} * [x_t, h_{t-1}]) \tag{32}$$

The "filter" $\tilde{o}_t$ is then point-wise multiplied with the new cell state c_t, normalized in the range of (-1; 1) by the hyperbolic tangent function (equation 33). This is then used as an output of the cell, as well as the hidden state for the next time step.

$$o_t = h_t = \tilde{o}_t \odot tanh(c_t) \tag{33}$$

Exploding Gradients

At the start of this subsection, I mentioned that gradients can also suffer from massive values - **exploding** gradients. While this is still the case even with LSTM cells, there is a simple and wide-spread solution - **gradient clipping** ([PMB12]). With gradient clipping, one sets a maximum *threshold* on a gradient's **norm**. The threshold is also sometimes called a **clip value**.
For example, setting the clip value to 1 and the gradient norm to 2 will force all n points of gradient $\boldsymbol{g}$ to be scaled down to satisfy equation 34:

$$\begin{aligned} \|\boldsymbol{g}\| &= 1 \\ \sqrt{g_1^2 + g_2^2 + \cdots + g_n^2} &= 1 \end{aligned} \tag{34}$$

4 Neural Machine Translation

Natural Language Processing is a field of research, connected to understanding, being able to work with and generating "natural", human language on computers. The first work in the field was done in the middle of the 20th century and featured translating 60 Russian sentences to English ([NOMC11]). Nowadays, NLP has rapidly evolved and is almost always connected with deep learning. This is due to the great success neural networks have had when applied to problems in the field, particularly recurrent ones (see section 3.3).

Neural machine translation is a task in the field of NLP, whose focus lies on using *recurrent* neural networks to translate sequences from one language, to another. This section will first cover how neural networks can be made to work with strings, by introducing the concepts of a **dictionary**, along with **integer** and **one-hot** encodings (section 4.1). Next, section 4.2 will introduce the family of **sequence-to-sequence**, or **encoder-decoder** models, which leverage recurrent neural networks to carry out NMT. Section 4.3 will focus on **attention** - a mechanism, which marked a new milestone in the performance of encoder-decoder networks. Further, section 4.4 will cover **Transformers** — a new type of feed-forward models that use attention to process sequence data. Finally, section 4.5 will introduce **BERT** - a state-of-the-art *language representation* model, built on the base of the Transformer model.

4.1 Strings in Machine Learning

Machine learning algorithms (this includes all deep learning models as well) require *numerical* inputs. This means that native text, in the form of strings and individual characters, is not directly supported. To remedy that, there exist two ways, with which to cast *tokens* (be it character, words, or sequences larger than that) to numerical representations - **integer** and **one-hot** encodings.
Both encodings require building a **vocabulary**. A vocabulary is nothing else than a **hash map** (also referred to as a **dictionary**), which maps tokens into integers. In fact, that is all there is to integer encoding tokens — one just needs to build a big dictionary and assign unique IDs to each one of them.
There is one problem with using integer encodings - namely, **ordinality**. Assume, for example, that we have the following dictionary:

$$\{\text{"cat"}: 1, \text{"dog"}: 2, \text{"sloth"}: 3\}$$

The integer encodings of the strings suggest an **order** between them, when there is not any. There is no objective basis, on which to compare cats with dogs and dogs with sloths, and assign them an order. A machine learning algorithm, however, does *not* know that — it only sees that some input neurons have bigger numerical values. The bigger numerical values can also influence the performance of the model in a bad way. Consider the simple feed-forward neural network from figure 3. Having a disproportionately high numerical value in one input neuron will blur out the effects of all other ones, except if the weights that are connected to it are miniscule.

One-hot encodings, or **embeddings**, fix the problem with ordinality by building *flat vector* representations for each token. As an example, if we take the animal vocabulary from above again, the one-hot embeddings of the three strings are shown in equation 35. One-hot embeddings have dimensionality N, equivalent to the number

of tokens in the dictionary. Additionally, the sum of the integers in every one-hot embedding is always 1, and all positions, except one, contain a 0.

$$\begin{aligned} \text{"cat"}&: [1, 0, 0] \\ \text{"dog"}&: [0, 1, 0] \\ \text{"sloth"}&: [0, 0, 1] \end{aligned} \tag{35}$$

4.2 Encoder-Decoder Models

Encoder-Decoder models (also called: **sequence-to-sequence**) are a specific family of deep learning approaches, which can be used to map variable-length sequence inputs to variable-length sequence outputs. They were first introduced in [SVL14], published in late 2014. Encoder-decoder models feature two "sub-modules" in their architecture: an **encoder** and a **decoder**. Figure 7 visualizes this concept, showing an LSTM encoder-decoder network in *training mode*.

As seen in the figure, the encoder reads input sequences one time step (i.e., token) at a time and accumulates *context* information in the **hidden** and cell states, passed between time steps. Before being passed to the LSTM cells, every input is also one-hot encoded in the **Embedding** layer. The *Embedding* layer is essentially a densely-connected *feed-forward* linear layer. It accepts a vector $X = \{x_1 \dots x_n\}$, with length n, consisting of **integer encodings**, as discussed in section 4.1. The Embedding layer then gradually creates and *optimizes* (via backpropagation while training) **latent representations** for every word from the vocabulary. At the end, each word from the vocabulary has a *fixed* integer encoding and a *learned* latent representation. The reasoning for doing this is to force the neural network to create *enhanced* one-hot embeddings for every word, which contains more information than just the order in the vocabulary. Rather, these *learned* embeddings are expected to

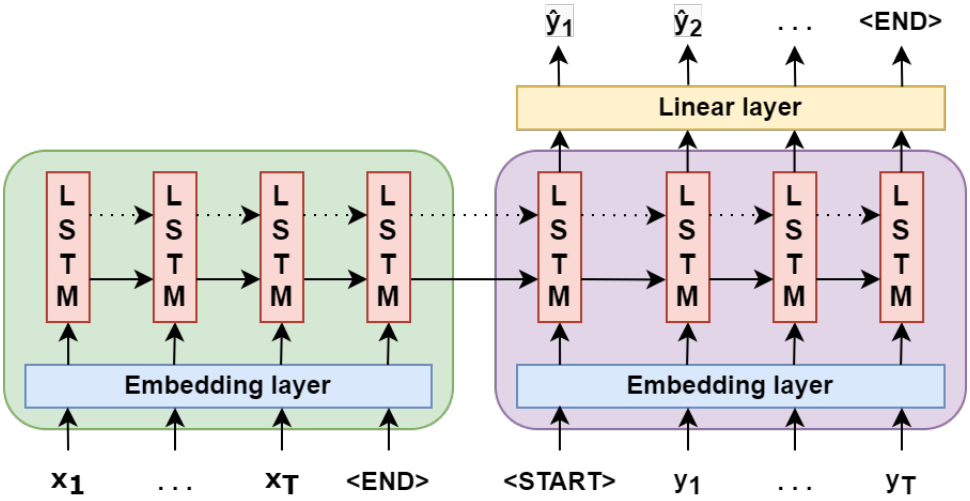


Figure 7: Training an LSTM encoder-decoder network with **teacher forcing**; the green block is the encoder and the purple block is the decoder solid arrows between LSTMs represent **hidden** and dotted - **cell** state

map words, which are often used in the same context (e.g., *king* and *queen*), **close** to one another.

The outputs of the encoder are generally discarded in sequence-to-sequence models and are therefore not visualized in the figure. At the end, the last states c_{T+1} and h_{T+1}, which are returned by the LSTM cell that processes the meta <**END**> token, are passed to the LSTM cell in the **decoder** block. As a convention, the hidden state passed from the encoder to the decoder is also called the **context vector** c. Formally, the equation of how the context vector is computed is shown in equation 36.

$$c = h_{T+1} = LSMT_T(<\text{END}>, h_T, c_T) \tag{36}$$

The intuition behind the process of the encoder is to generate *context* for the translation step in the decoder. The <END> token is a so called **meta token**, which is added extra to the vocabulary and is used to mark the end of a sequence. Usage of the <END> token allows **variable-length** input to the encoder-decoder network, even though there is a maximum sequence length (in figure 7, it is T+1). Indeed,

by introducing another meta token - <**PAD**>, one can also input sequences with length $< T$, by placing <END> at the end of the sequence and then filling the rest of the sequence with <PAD> meta tokens.

The decoder module of the encoder-decoder model is the one carrying out the *translation* itself. As shown in 7, the decoder model also accepts inputs one time step at a time and can produce *variable-length* output sequences (same mechanism as with encoder inputs). Additionally, the decoder also has an embedding layer, which maps inputs to one-hot encodings. What is immediately noticeable, however, is that the input for the *decoder* itself is **shifted** one time step to the right. This is because the decoder module is **autoregressive**. Autoregressive is a broad term outside the scope of deep learning and means a process, which uses information from *past* time steps to decide the output of the *current* time step. In order for this to hold true for the *first* character of the model prediction, a helper meta token <**START**> is used. Formally. the *training* prediction $\hat{y}_t$ of the encoder decoder model at time step t is calculated as shown in equation 37:

$$\hat{y}_t = Linear(LSTM_t(emb(y_{t-1}), h_{t-1}, c_{t-1})) \tag{37}$$

The linear layer at the top of the decoder module is often notated in the literature as a **classifier** layer. Indeed, it is a normal densely-connection *feed-forward* layer, which maps the *latent* output of the LSTM cell to the **task vocabulary**. As such, its weight matrix has dimensions $d_{hidden} \times N$, where d_{hidden} is the size of the hidden state in the LSTM cell, and N is the size of the task vocabulary. The outputs of the classifier layer are often additionally put through the **softmax** activation function, in order to create a normalized probability distribution in the range (0; 1). Then, if element i has the highest-assigned probability for time step t, it means that the encoder-decoder model predicts the token with *integer encoding* i has to be the output. Coming back to the example from the previous section 4.1 with the vocabulary of {"*cat*", "*dog*", "*sloth*"}, let the encoder-decoder output for time step t to be [0.2, 0.75, 0.05]. As

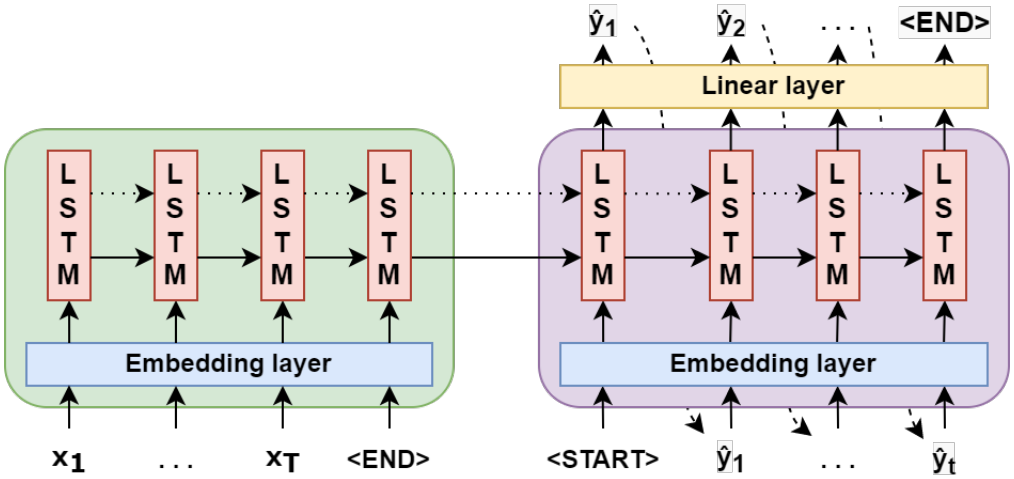


Figure 8: Inference with LSTM encoder-decoder network;
the green block is the encoder and the purple block is the decoder
solid arrows between LSTMs represent **hidden** and dotted - **cell** state

dog is the second token in the vocabulary, and thus has the integer encoding **1**, the output shows that the model predicts *dog* to be the expected token at time step t.

4.2.1 Training Encoder-Decoder Models

As pointed out in the beginning of the last subsection, figure 7 visualizes the encoder-decoder model while *training*. Indeed, the figure shows the inputs of the decoder model to be a shifter version of the *expected* output sequence y. This method of learning is called **teacher forcing**. Whether it is good to use teacher forcing is still debated in the literature — see [HZZG19]. On one hand, it is claimed to speed up weight *convergence*, meaning it takes less time for the network to be fully trained. On the other hand, teacher forcing conditions the model on *always* seeing the perfect sequence of inputs. This might lead to worse performance when running inference with the model, as one wrongly predicted character might ruin the whole decoding process.

The other type of optimization algorithm that may be used to train encoder-decoder

networks is "non-teacher forcing". The process is the same as running inference with the network and is visualized in 8. With non-teacher forcing, the output of the **previous** time step is used as input for the current prediction. This ensures that the network learns from its mistakes, which is supposed to make it generalize better. In the same time, however, non-teacher forcing has a *slower* and more *unstable* convergence.

In practice, teacher-forcing and non-teacher forcing are *mixed* during training. This is done by assigning a probability for executing teacher forcing (usually: 0.5) and executing non-teacher forcing if the randomly generated number is higher. This mix of optimization algorithms combines the faster convergence of teacher forcing with the better generalization ability of non-teacher forcing. I refer the reader to [BVJS15] for more information.

4.2.2 Inference with Encoder-Decoder Models

Figure 8 showcases how inference is done with an encoder-decoder model. As mentioned in the beginning of this section, the decoder is an **autoregressive** model. It takes the output from the last time step $\hat{y}_{t-1}$ and uses it to calculate the current prediction $\hat{y}_t$. Formally, this is expressed in equation 38:

$$\hat{y}_t = LSTM_t(emb(\hat{y}_{t-1}), h_{t-1}, c_{t-1}) \tag{38}$$

4.2.3 Bidirectional LSTM

Bidirectional LSTM networks feature an additional *layer* — i.e., an additional LSTM cell on top — which processes the input sequence **in reverse**. Figure 9 visualizes this concept with an encoder module, which uses a bidirectional LSTM network. The rationale behind bidirectional LSTM networks is that knowing what is *coming* in the

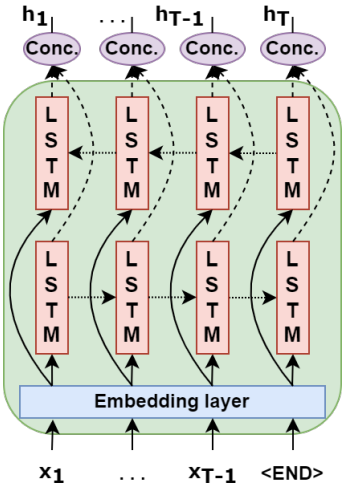


Figure 9: Encoder module with **bidirectional** LSTM network; arrows between LSTMs represent **both** hidden and cell state

sequence might provide additional context for the sequence at time step t. If we take text generation as an example, the word **cat** would be harder to predict in "*There's a _*" than in "*There's a _ stuck in the tree*". Formally, this means that the hidden state h_t (also applies to the cell state c_t), generated at time step t, can be calculated by using equation 39

$$h_t = concat(LSTM_{ft}(x_t, h^f_{t-1}, c^f_{t-1}), LSTM_{bt}(x_t, h^b_{T-t}, c^b_{T-t})) \tag{39}$$

It is worth mentioning that different implementations of bidirectional LSTM networks use different strategies for *combining* the states from both directions. Figure 9 shows the option to **concatenate** both states, but it is also possible to **sum** them up, or **average** them.

4.3 Attention

It was first proposed in [BCB15] to include an additional **attention layer** to encoder-decoder models, which consists of two additional densely-connected feed-forward

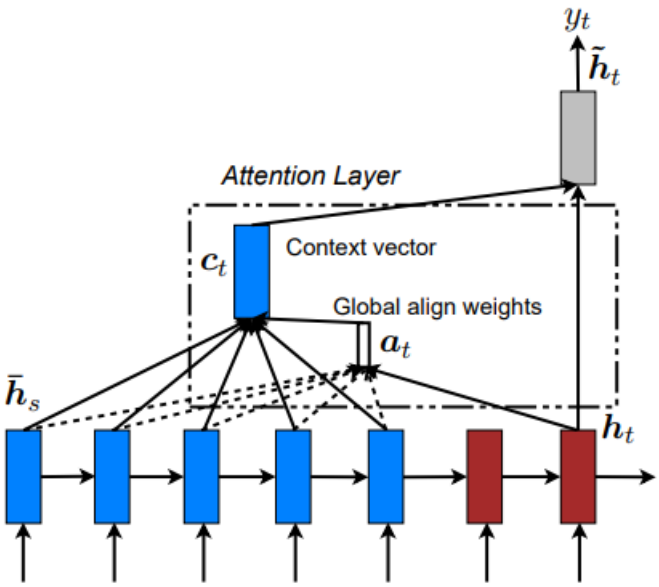


Figure 10: Visualization of **attention** in an encoder-decoder network; $\boldsymbol{h}_t$ is the output (hidden state) of decoder RNN cell at time step t; $\bar{\boldsymbol{h}}_s$ is a vector with encoder outputs; $\boldsymbol{a}_t$ represents a vector of **alignment weights**; $\boldsymbol{c}_t$ is the resulting **context vector**; taken from [LPM15]

layers. Intuitively explained, the attention layer calculates which words from the **input sequence** have the biggest *importance* with the network prediction at time step t. In order to build more intuition behind this, consider the German sentence "*Das ist eine schwarze Katze*" and its English translation "*That is a black cat*". When making a prediction for the last time step (the 't' character in 'cat'), it is natural to give the most **attention** to its German equivalent **Katze**. One may also wish to put **schwarz** under attention, as it might further describe the word that needs prediction.

The concept of attention in encoder-decoder models was refined by Luong et al. in [LPM15]. In particular, they showed that the attention layer can be *simplified* down to a **dot product**, and still provide good results, if not better for certain tasks. Figure 10 visualizes the attention mechanism, as proposed by Luong et al.

The **attention layer**, as shown in the figure, takes in two inputs — the output h_t of

the decoder RNN cell at time step t, and a **vector of vectors** (also called: **a tensor**) $\bar{\boldsymbol{h}}_s$, which contains the outputs of the encoder RNN cells at **every time step**. It is important to keep in mind that what I mean by *outputs* in the previous sentence are just the **hidden states**, which are returned by the RNN cells (refer to section 3.3). The first step in calculating the attention is determining the **alignment weights** (also called: **attention weights**). Intuitively, the alignment weights determine for *every* encoder output $\{\bar{h}_1, \ldots, \bar{h}_T\}$ how *important* its information is for the current prediction. If we return to the previous example of predicting the last word *cat* in the sentence pair "*Das ist eine schwarze Katze*" and "*That is a black cat*", one might expect a high attention weight for the encoder output of last time step — *Katze*. These alignment weights are usually put through the *softmax* activation function, in order to transform them into a probability distribution in the range (0; 1). The equation for computing the alignment weights with a dot product **scoring function** is shown in equation 40.

$$a_t = softmax(h_t \cdot \bar{\boldsymbol{h}}_s^\top) \tag{40}$$

The calculated attention weights a_t at decoder time step t are then used to determine the **context vector** c_t. The *context vector* is nothing more than a further dot product between the vector of attention weights a_t and the matrix of encoder outputs $\bar{\boldsymbol{h}}_s$. By doing the operation, encoder outputs with a low attention *weight* get blended out, while the ones with a high attention weight stand out. Formally, the computation of the attention scores is shown in equation 41. In order to avoid confusion with the other notation for context vector, which is the final hidden state of the encoded that is passed to the decoder, I will be referring to c_t from the attention layer as **attention vector**.

$$c_t = a_t \cdot \bar{\boldsymbol{h}}_s \tag{41}$$

Additive vs. Multiplicative Attention

Except the complexity of the *scoring function* — dot product versus two densely-connected feed-forward layers — there is one more difference between the attention mechanisms between [BCB15] and [LPM15]. Namely, they propose different points, at which to calculate the **attention vector**.

In [LPM15], Luong et al. suggest using the decoder RNN cell output h_t at time step t and then concatenating the attention vector with the output, before passing the result further down (usually — to a densely-connected linear **output** layer). This type of attention is called **multiplicative** attention in the literature.

On the other hand, Bahdanau et al. propose in their original paper [BCB15] to use the hidden state h_{t-1} from the **last** time step $t-1$ to calculate the attention vector. For time step $t = 1$, the context vector from the encoder is considered. Moreover, the alignment vector is the concatenated with the *same* hidden state h_{t-1}, and it is passed to the RNN Cell at time step t to produce h_t. This type of attention is called **additive** attention.

[VSP+17] provide a brief comparison of the two types of attention. They determine multiplicative attention to be *faster* and more *space-efficient*. However, the authors report that additive attention performs better than multiplicative attention when the **hidden state size** is set to *large* values.

4.4 Transformers

The attention mechanism from last section 4.3 lead to big improvements in the performance of sequence-to-sequence models in Neural Machine Translation (see [BCB15] and [LPM15]). Three years later, a team of Google engineers proposed an entirely new model, which processes sequence data *entirely* by using attention. The model was first introduced in [VSP+17] and it is called the **Transformer**. Figure 11 shows the architecture of the Transformer model. Transformers also use the

encoder-decoder design, but do not use any **recurrent** networks in their modules. The rest of this subsection will explain the elements of the transformer in more depth — namely, **Positional Encoding** (subsection 4.4.1), **use of attention** (subsection 4.4.2 and **residual connections** 4.4.3. Subsection 4.4.4 summarizes and lists out the flow of data through the model, while clearing up any leftover points. Finally, 4.4.5 briefly delves into how the Transformer model is trained, as well as showing how inference can be done with it.

4.4.1 Positional Encoding

As mentioned in the beginning of this section, the Transformer model does not use *recurrent* networks in its design. Therefore, input sequences are passed in their entirety — not on a time step basis, as shown with recurrent encoder-networks in section 4.2. Position information in sequences is, however, important metadata that can influence how a sequence is processed. For example, consider the sentence "*It's okay **not** to eat*" versus the sentence "*It's **not** okay to eat*". The first one expresses that it is acceptable for **someone** not to eat (e.g., in the presence of others), while the latter one could act as a warning against, e.g., poisonous mushrooms.

In order to keep position information intact for the Transformer model, the authors of [VSP+17] suggest adding **positional encodings** to the embedded representations of the input. The formulas for generating positional encodings are show in equations 42 and 43. They encode every **latent element** in an embedded input representation with dimension d_{model}. They do this by using alternating **sine** and **cosine** waves with **decreasingly lower** frequencies.

$$PE_{(pos,2i)} = sin(\frac{pos}{10,000^{2i/d_{model}}}) \tag{42}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10,000^{2i/d_{model}}}) \tag{43}$$

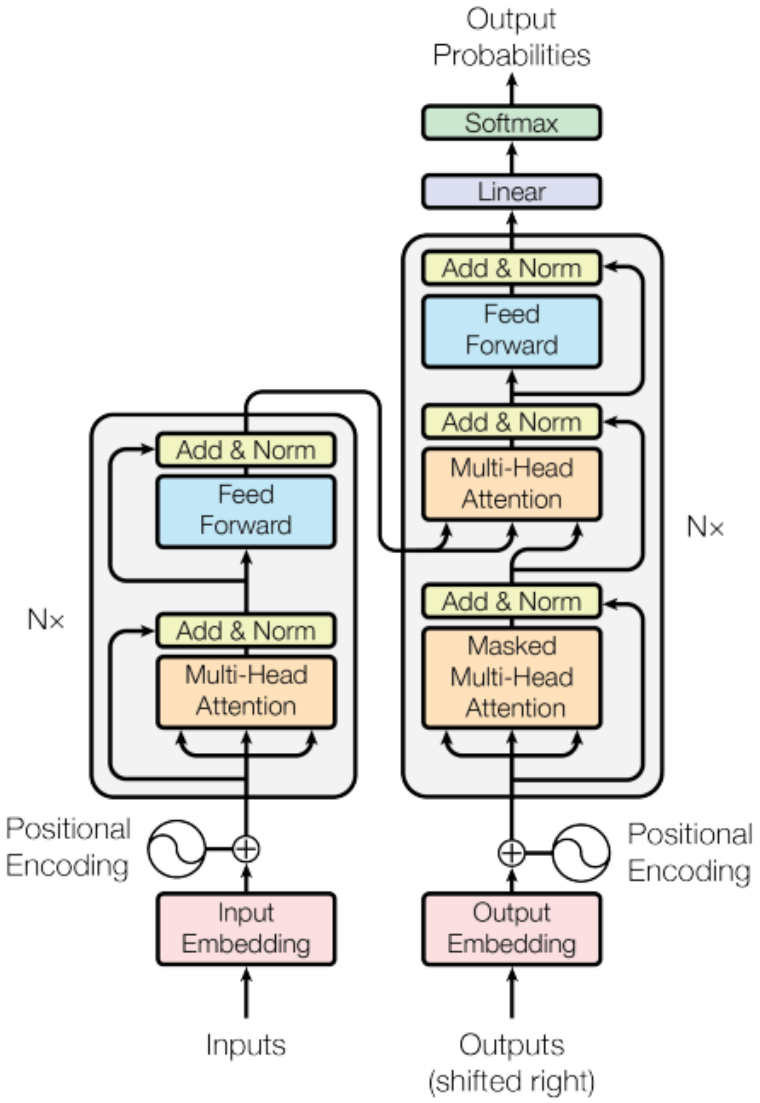


Figure 11: Architecture of the Transformer model; image taken from [VSP+17]

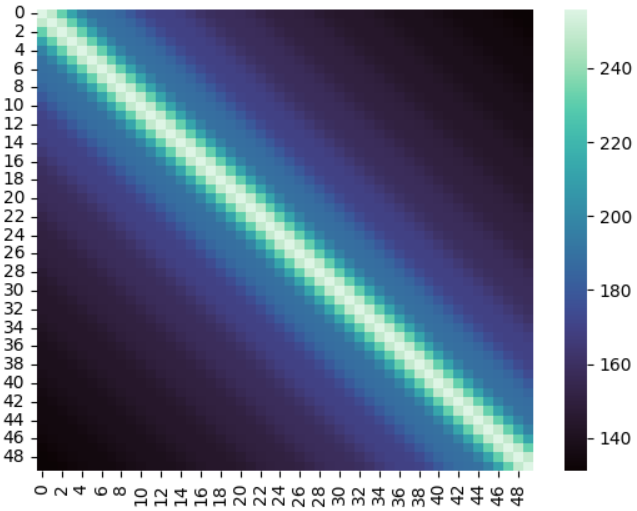


Figure 12: Distance (**dot product**) of positional embeddings with seq. length 50

The authors of the paper do not delve into details why using alternating sine and cosine waves creates a good positional encoding. They argue that by using these **fixed** encodings, instead of **learned** ones (i.e., learned by the Transformer during training), they expect the model to be able to extrapolate longer sequences during inference. Moreover, they hypothesize that these encodings would help the model attend elements of the sequence by **relative position** — e.g., character $\#i$ should put attention on character $\#i - 5$. This is due to the *linear nature* of the function, where for a fixed offset k and position pos, PE_{pos+k} can be found by a linear transformation of PE_{pos}. The exact transformation is derived and can be inspected in the excellent article [Den19].

On further inspection, one may find additional desirable attributes in the sine and cosine positional encoding. Indeed, the encoding is flexible — encoding values for longer sequences can always be generated by substituting the waves from equations 43 and 42 with an even bigger frequency, and it does not lead to *exploding gradients* by virtue of its outputs being in the range [-1; 1] (compare the latter with using integer

values to indicate order to grasp the problem). Finally, if one calculates the **distance** between the positional encodings at different positions, the result is a symmetrical matrix with a smooth and gradually-decreasing gradient, as shown in figure 12. This attribute makes sense *intuitively* - the distance between directly neighboring time steps should be the most pronounced, as most attention *should* be paid there. For distant steps, the distance can also be shorter, as less attention is paid and the time steps can essentially be *merged*.

4.4.2 Use of Attention

The core elements of the Transformer model are the two Multi-Head Attention modules in the encoder and decoder modules, as well as the Masked Multi-Head Attention module in the decoder. Two of these modules carry out so-called **self-attention** — the two "at the bottom" of the network in figure 11 — while the third one performs **cross-attention**. Cross-attention is essentially the same mechanism as the attention mechanism in recurrent encoder-decoder networks (refer to section 4.3), except the authors use **Scaled-Dot Product Attention**.
In the upcoming subsections, I will explain what Scaled Dot-Product Attention is, as well as how it ties together into the Multi-Head Attention module. At the end, I will explain self-attention and go into why the self-attention module in the decoder has to be **masked**.

Scaled Dot-Product Attention

The Transformer model uses multiplicative attention with a dot product, as shown in section 4.3. However, the authors of the Transformer added a scaling factor of $\frac{1}{\sqrt{d_k}}$, where d_k stands for the hidden state size of the model (same as the character embedding dimension). They motivate the need for scaling by arguing that for large values of d_k, the *softmax* operation causes gradients to become minuscule and thus

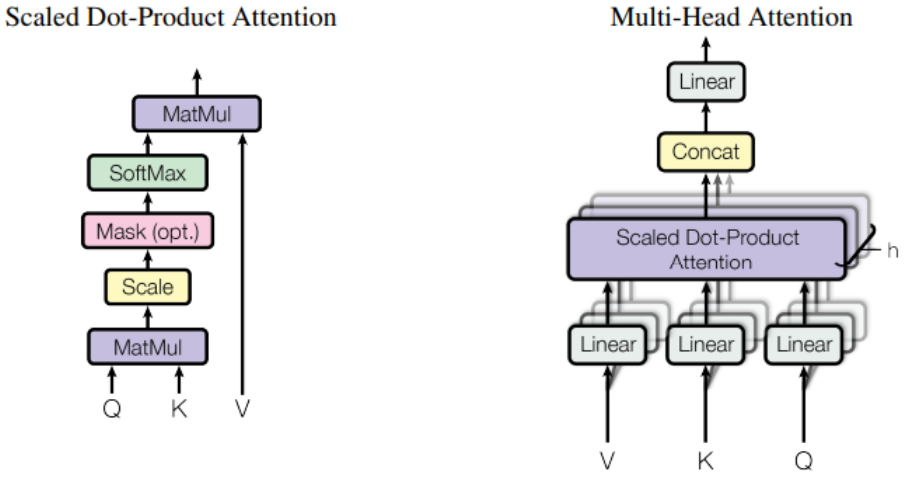


Figure 13: Multi-Head Attention module, as visualized in [VSP+17]

hinder training. The result is called Scaled Dot Product Attention and is shown in equation 44:

$$Attention(Q, K, V) = softmax(\frac{QK^\top}{\sqrt{d_k}})V \tag{44}$$

The keen reader will notice that the notation, which the authors use, is different from the one used in the previous section 4.3. This is due to the fact that the authors *re-framed* attention as an *information retrieval* task, with Q being a **query**, K being keys, and V being values. The result of attention is then the sum of **weighted values**, where the weight of every value determines how good the query "*passes*" to a certain key.

If one removes the scaling factor $\frac{1}{\sqrt{d_k}}$, equivalence with the previous version of attention in recurrent encoder-decoder networks can be shown by setting $Q = h_t$ and $K = V = \bar{\boldsymbol{h}}_s^\top$.

Multi-Head Attention

The Multi-Head Attention module consists of **multiple stacks** (also called: **attention heads**) of **Scaled Dot-Product Attention**. The authors of the paper explain this

decision by claiming that different heads will extract different attention information, which can finally be concatenated to give a better understanding of the dependencies in the sequence. Formally, the output of the Multi-Head Attention module, with input vectors Q, K, and V, as well as h attention heads is shown in formula 45.

$$\begin{aligned} MultiHead(Q, K, V) &= Concat(head_1, \ldots, head_h)W^O \\ \text{with } head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{45}$$

Self-Attention

Intuitively explained, **self-attention** runs the attention mechanism on a single sequence itself. The reasoning behind doing this is that the result of the operation transforms the representation of the sequence from its original dimension to a *latent* one, where the inner-time step dependencies are accentuated. Indeed, the authors in [VSP+17] compare self-attention with other methods of extracting dependency information, namely by using *recurrent* or *convolutional* layers. They claim that self-attention is able to learn **long-term** dependencies best, with the addition of them being *parallelizable*.

Self-attention is inherently **bidirectional**. For a sequence with length n, it creates a *symmetrical* $n \times n$ matrix M, with m_{ij} being the attention value of time step i with respect to time step j. This is desirable with the Multi-Head Attention module in the encoder, as well as the cross-attention Multi-Head Attention module in the decoder, as time step i benefits from context in both directions. For the self-attention module in the decoder, however, this becomes an issue. Indeed, if allowed to attend "in the future" during training, the transformer model might grow dependent on *upcoming* time steps. This can, in turn, lead to drastic performance drops during inference, as the **autoregressive** nature of the Transformer will prevent the model from attending future time steps.

The fix to this problem is applying a **mask**, which nullifies the attention weights of

subsequent time steps. This is visualized on the left part of figure 13, with the pink *Mask* square representing the step, where the mask sets the attention weights to be 0.

4.4.3 Residual Connection

A **residual connection** (also called: **skip connection**) in a neural network is a *link* between the start and the end of some neural network module. In the case of the Transformer, there are residual connections around every Multi-Head Attention and Feed Forward module, as shown in figure 11.
The first wide-spread successful usage of residual connections was in the field of computer vision, in the paper [HZRS15]. There is still an ongoing debate in the literature about the effect of residual connections, but they have shown promising empirical results, as shown in the aforementioned paper. It is likely because of this that the authors of the Transformer models also decided to include residual connections into their model.

4.4.4 Data Flow

The encoder module of the Transformer model accepts an input sequence X in its entirety. As shown in figure11, the input sequence first passes through an **Input Embedding** layer, which maps the input to a *latent* representation. The *Input Embedding* layer generates *learned* embeddings for each token in the vocabulary, as explained in the beginning of section 4.2. The authors also add that they multiply each element of latent embedding with $\sqrt{d_{model}}$. A likely explanation (none given by the authors) is that the multiplication is done to *exaggerate* the values of the input embeddings, so that their meaning is not lost when added with the positional embeddings. Next, positional information is included by point-wise adding **positional**

embeddings, which results in the final version of the input embeddings X_{emb} (as shown in equation 46).

$$X_{emb} = PE(Emb(X) * \sqrt{d_{model}}) \tag{46}$$

Afterwards, X_{emb} is copied into three tensors and passed to the first **Multi-Head Attention** module, in order to compute *self-attention*. The result from the module is added to X_{emb} via a residual connection and is normalized with layer normalization (refer to [BKH16]). Formally, the passing of the first "sub-module stack" transforms X_{emb} into X_{sa} (sa stands for self-attention) as shown in equation 47:

$$X_{sa} = LayerNorm(X_{emb} + MultiHead(X_{emb}, X_{emb}, X_{emb})) \tag{47}$$

Afterwards, X_{sa} is put through another "sub-module stack", consisting of a feed-forward network, residual connection and layer normalization. The resulting transformation, which outputs X_{enc}, is shown in 48:

$$X_{enc} = LayerNorm(X_{sa} + FFN(X_{sa})) \tag{48}$$
$$\text{with } FFN(x) = max(0, xW_1 + b1)W_2 + b2$$

The result of the encoder module — X_{enc} — is then passed to the decoder module to be used for *cross-attention*, akin to how a context vector is passed from the encoder to the decoder in a recurrent sequence-to-sequence network.

Similarly to the encoder module, the decoder module puts the **whole** output sequence through an embedding and positional encoding layer first, as shown in equation 49:

$$y_{emb} = PE(Emb(y) * \sqrt{d_{model}}) \tag{49}$$

An important detail is that the output sequence has to be **shifted to the right**. This is done, in order to preserve the *autoregressive* quality of an encoder-decoder network.

Formally, this transforms the output sequence $y = y_1 y_2 \dots y_m$, with sequence length m, to the sequence $y_{shift} =< START > y_1 \dots y_{m-1}$. This shift forces the transformer model to predict the **successive** token for each time step, i.e., $<START> \rightarrow y_1$, $y_1 \rightarrow y_2$, all the way up to $y_{m-1} \rightarrow y_m$. $<START>$ is the same meta token, which was used with recurrent encoder-decoder networks from section 4.2. Likewise, $<END>$ and $<PAD>$ tokens can be used to enable the Transformer to work with **variable-length** inputs and outputs.

The operations in the decoder module then continue with y_{emb} passing through the *masked* self-attention module, before being added to itself via a residual connection, and normalized. The resulting vector y_{sa} is shown in equation 50:

$$y_{sa} = LayerNorm(y_{emb} + MultiHead(y_{emb}, y_{emb}, y_{emb})) \tag{50}$$

The output of self-attention is then given to the final Multi-Head Attention module, which computes the **cross-attention** values between y_{sa} and X_{enc}. This is the module, which connects the *inputs* and *outputs* of the network, relying on the attention mechanism to highlight the dependencies between them. The resulting vector y_{ca} (ca stands for cross-attention) is computed with equation 51:

$$y_{ca} = LayerNorm(y_{sa} + MultiHead(X_{enc}, X_{enc}, y_{sa})) \tag{51}$$

The final part of the decoder module is another stack, consisting of a feed-forward network, finished with a residual connection and layer normalization. The result y_{dec} can be determined by following equation 52:

$$\begin{aligned} y_{dec} &= LayerNorm(y_{ca} + FFN(y_{ca})) \\ \text{with } FFN(x) &= max(0, xW_1 + b1)W_2 + b2 \end{aligned} \tag{52}$$

The output of the decoder module y_{dec} is then passed through an additional *classifier* layer, as shown in section 4.2. The layer has dimensions $d_{model} \times N$, with N being the

number of tokens in the task *vocabulary*. Optionally, as shown in figure 11, *softmax* can also be applied to the classifier layer to normalize the predictions to a probability distribution in the range (0; 1).

4.4.5 Training and Inference

The last subsection 4.4.4 made it clear that the Transformer takes output sequences in their *entirety* and "simulates" autoregression by shifting them one time step to the *right*. This attribute of the model makes training *faster* than compared with recurrent encoder-decoder networks, as the decoder module needs to do only **one pass** with the output data. Indeed, let $y^i =< START > y_1^i y_2^i \dots y_{m-1}^i y_m^i < END >$ be the **expected** output sequence of a data sample from an arbitrary dataset with sequence length m. Then, the Transformer needs the following two versions of this sequence for training: $y_{inp}^i =< START > y_1^i \dots y_{m-1}^i y_m^i$ and $y_{target}^i = y_1^i y_2^i \dots y_m^i < END >$. y_{inp}^i is passed as the "input" to the decoder layer, while y_{target}^i is used as the expected *target*. This essentially means **teacher forcing** is always used with the Transformer model, as y_{inp}^i — the sequence that is passed to the decoder module — is a shifted version of the expected output itself.

As far as inference is concerned, the decoder module again needs a sequence in its *entirety*. However, this is not possible with inference, as there is no expected output — the model itself needs to generate it. To circumvent this, the model starts by using an output sequence y with sequence length m, fully comprised out of $< START >$ tokens, as shown in box 53:

$$y = \underbrace{< START >< START > \cdots < START}_{\mathbf{m} \text{ times}} \tag{53}$$

After passing the "*dummy*" start sequence through the decoder, the $< START >$ token at time step **2** is replaced by the model's prediction for time step **1** $\hat{y}_1$. Afterwards, the modified sequence, shown in box 54, is fed through the Transformer to

generate the prediction for time step **2**.

$$y =< START > \hat{y}_1 \underbrace{\cdots < START}_{\textbf{m-2 times}} \tag{54}$$

This process essentially emulates *autoregression*, as it was shown in figure 8 for the recurrent encoder-decoder network. It builds up the prediction sequence $\hat{y} = \hat{y}_1\hat{y}_2 \ldots \hat{y}_m < END >$ time step by time step and returns it at the end.

4.5 BERT

Bidirectional **E**ncoder **R**epresentations from **T**ransformers, or **BERT**, is a *language representation* model. BERT was first published in [DCLT18] by a team of Google engineers, just like the *Transformer* model. That is no coincidence, as BERT uses the **Encoder module** from the Transformer as the basis in its architecture, as explained in subsection 4.5.2 further in this section.

Before introducing BERT, this chapter will briefly cover the foundations that it lies on. In particular, subsection 4.5.1 will go over the concept of **word embeddings**. Afterwards, subsection 4.5.2 will present the **architecture** of the BERT model and formalize the flow of data through the model. Next, subsection 4.5.3 will explain the BERT **tokenization** process, i.e., how text sequences are turned into smaller *tokens*, before being fed to the model. The subsection 4.5.4 closes out the section with an explanation of how the BERT model was *pre-trained*.

4.5.1 Word embeddings

As indicated in section 4.1, deep learning models can not work with the *raw* string representations of text — they need **numerical representations**. The concept of **word embeddings** builds on this notion — rather than just creating flat one-hot

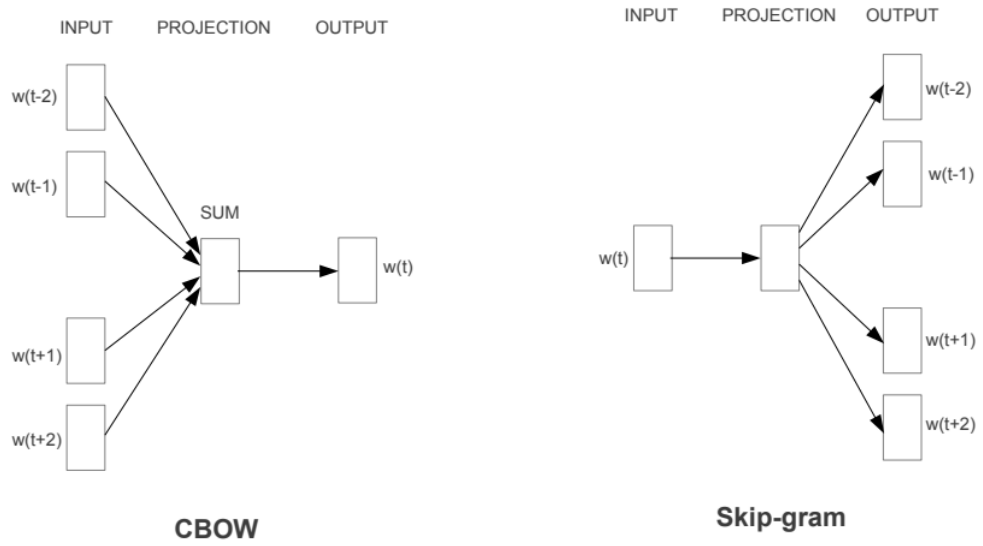
encoded vector representations without any context, *word embeddings* maps words with similar meanings *close* to one another.
A trivial example of this is to think of a dummy dataset, consisting of all the **letters** of the English alphabet, all **digits** from 0 to 9, and a small set of random punctuation marks (*'.'*, *'?'*, *'!'*, *'*'*, *'+'*). Then, a *well-made* set of word embeddings would be expected to map all letters to vectors with *similar* values, and also do the same for the digits and punctuation.

There is a choice to make with word embeddings — namely how **big** the size of the *latent* space should be. The size of the latent space is a **hyperparameter**, akin to determining the number of neurons in a hidden layer in a feed-forward neural network. A *smaller* latent size indicates that the word embeddings have less **capacity** — i.e., there is less *"space"*, in which to project all the words. This might be desirable if working with a *smaller* collection of words, but not with a bigger word list. On the other hand, a *bigger* latent size offers more *capacity* to fit all word embeddings, but it also brings a higher *memory cost*. In practice, it is often the case that different word embeddings with varying latent size are generated. It is then left up to the down-stream user to evaluate the performance of the different latent sizes and match them with their *memory requirements*.

Word2Vec

A team of Google employees proposed the first widely-spread *approaches* for generating word embeddings in [MCCD13]. The models of the approach, as well as the resulting set of word embeddings, are ubiquitously knows as *word2vec*, named after the repository, under which the code is shared[1]. The two approaches from the paper are called "**Continuous Bag of Words**" and "**Skip-grams**". I will briefly explain how the "**Continuous Bag of Words**" approach works, which is visualized in figure

[1] `https://code.google.com/archive/p/word2vec/`



(a) CBOW training process

(b) Skip-gram training process

Figure 14: The two approaches of generating Word2Vec embeddings; taken from [MLS13]

14a. The "**Skip-gram**" approach, which is shown in figure 14b, is omitted for the sake of brevity — I refer the reader to the original paper [MCCD13].

The continuous bag of words (CBOW) model uses a simple two-layer feed-forward network (referred in the original paper as **Neural Net Language Model**). The input layer has V neurons, with V also being the number of different words in the vocabulary. The single hidden layer then has N neurons, and the output layer again has V nodes. The hidden layer does not have an activation function, while the output layer uses *softmax* to normalize the output distribution over all vocabulary words. Formally, the weight matrix between the input and hidden layer is $W_1 \in \mathbb{R}^{V \times N}$ and the one between the hidden and output layer - $W_2 \in \mathbb{R}^{N \times V}$. W_1 is nothing more than an **embedding** layer, which we saw used in encoder networks in sections 4.2 and 4.4. Indeed, the resulting set of word embeddings after training the model is the result of multiplying the **one-hot encoding** of each word in the vocabulary with W_1. The multiplication produces a vector of N values, which is the latent representation

— i.e., word embedding — of the arbitrary word.

Next, a **sliding window** approach is used to create training samples for the CBOW model. The *sliding window* approach involves determining some *window* size c, and then using c words to the *left* and *right* of some word to predict it. Explained in an example, let $c = 3$ and some document dataset contain the sentence *"You dropped your crown, my queen"*. Box 4.1 then shows how the sliding windows approach is used to create training samples. It is important to notice how each input sample has a **symmetrical** number of words — exactly $\lfloor \frac{c}{2} \rfloor$ from the left and the right. Indeed, the word **Continuous** from "Continuous Bag of Words" comes from the fact, that the input samples contain words from **both** directions of the target one.

You dropped your crown, my queen → (("you", "your"), "dropped")
You dropped your crown, my queen → (("dropped", "crown"), "your")
You dropped your crown, my queen → (("your", "my"), "crown")
You dropped your crown, my queen → (("crown", "queen"), "my")

Box 4.1: Creation of a CBOW training sample; red represents the **context** words (input); green represents the **target** word (output)

After generating training samples, the CBOW model can be trained by using **cross-entropy**. Formally, the forward pass through the model is shown in equation 55, while the computation of the loss value is shown in equation 56. In the two equations, $\hat{y}_w$ is used to represent the prediction for the **target** word in the *middle* of the sequence, while x_{c_i} is used to represent the i-th **context** word. The sum operation in the forward pass sums up the *latent embeddings* of all context words *point-wise*, as shown in figure 14a. The loss function is a simplified version of the multi-class cross-entropy function, first introduced in 13.

$$\hat{y}_w = softmax(W_2 * \sum_{i=0}^{c-1}(W_1 * x_{c_i} + \boldsymbol{b}_1) + \boldsymbol{b}_2) \tag{55}$$

$$L(\hat{y}_w, y_w) = log(\hat{y}_w) \tag{56}$$

4.5.2 Architecture

Figure 15 shows the design of the BERT model, as originally proposed in [DCLT18]. It features **three** encoding layers. Two of these — **Input** and **Positional** encoding — are also used for the regular Transformer model, as explained in section 4.4. The new **Segment** encoding layer is related to the format of the **input** sequence. Indeed, figure 15 shows the input to BERT consisting of two **different** sequences — $\boldsymbol{X} = x_1 \dots x_T$ and $\boldsymbol{y} = y_1 \dots y_{T'}$. This format allows BERT to be used in a wider range of downstream tasks, e.g., **question answering**. The distinction between the two sequences is first made by the meta *[SEP]* token, and second by the newly-introduced "**Segment encoding**" layer. The new layer produces **learned** embeddings (same as the *input* embeddings), which indicate whether a subword is a part of the *first* or *second* sequence.

The presented input format is *not* enforced. Indeed, for **token classification** tasks (e.g., OCR error detection), the input to the BERT model gets rid of the second sequence $\boldsymbol{y}$ and ends with a *[SEP]* token. As the input size is **fixed**, the rest of the input sequence is filled with meta *[PAD]* padding tokens.

The **base** BERT model itself, without the encoding layers before it, consists of 12 *Transformer encoder modules* stacked on top of each other. The authors also propose a **large** BERT, which has 24 stacks.

Each "Transformer encoder block" contains the two submodules, which are visualized in the figure 11. Namely, it contains a *self-attention module*, with a residual connection and layer normalization, as well as a *feed-forward module*, again followed by a residual connection and layer normalization. Compared to the original Transformer model, however — where the vector of encoder hidden states $\boldsymbol{h}_{enc}$ was passed to the decoder for cross-attention — $\boldsymbol{h}_{enc}$ is passed to a further copy of an encoder module as input. Formally, if we let $\boldsymbol{X}_{emb}$ be the final version of the input sequence after the three encoding layers, the resulting output vector of BERT $\boldsymbol{h}_{BERT}$ is calculated by (see equation 57). For a more detailed description of $TransEnc_i$, refer to the beginning

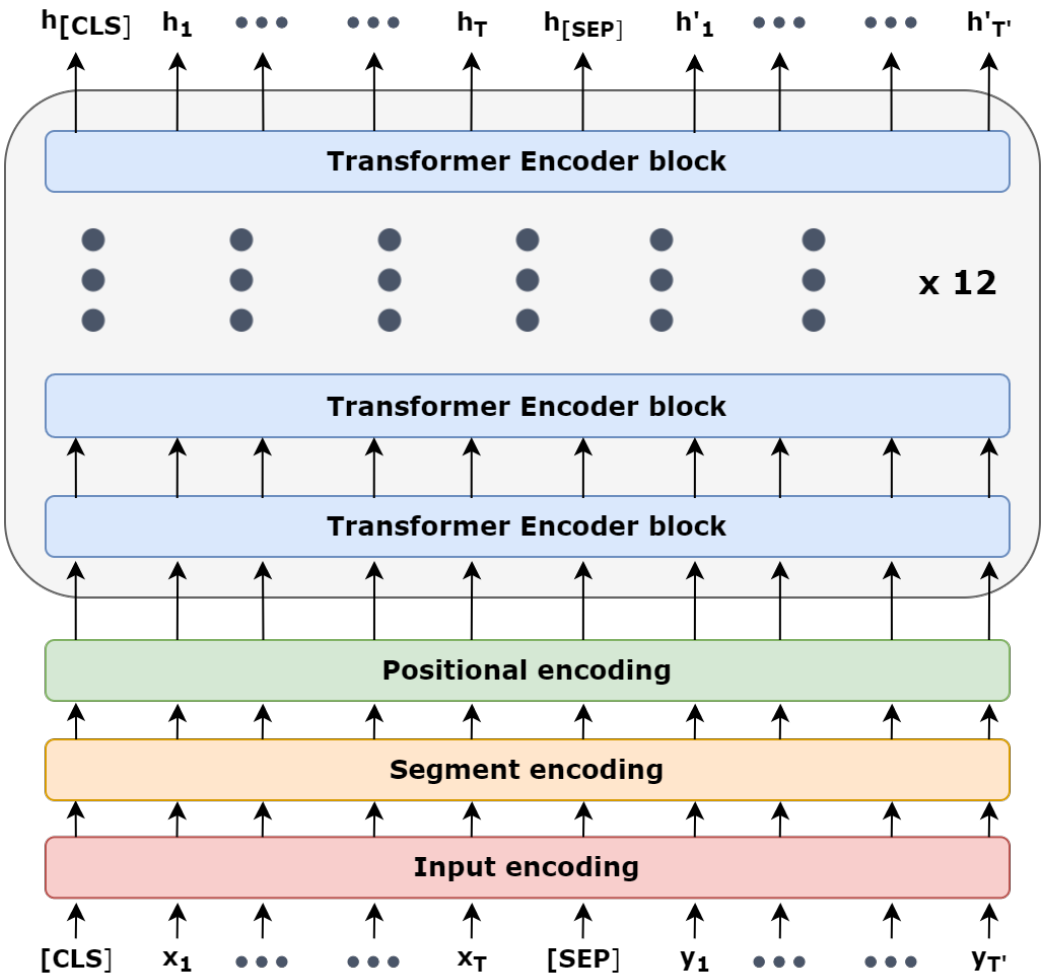


Figure 15: Bidirectional Encoder Representations from Transformers (**BERT**) model architecture

of subsection 4.4.4.

$$\boldsymbol{h}_{BERT} = TransEnc_{11}(TransEnc_{10}(\dots(TransEnc_0(\boldsymbol{X}_{emb})\dots) \tag{57}$$

Comparison vs. previous embeddings

BERT improved the state-of-the-art results on **11** NLP tasks, as discussed in [DCLT18]. The first reason for this success is that the subword embeddings from BERT are **context-dependent**. To better understand what this means, consider the earliest generation of word embeddings — i.e., word2vec and GloVe from subsection 4.5.1. There, the word embeddings are **context-independent**. Regardless of whether the word *cell* appears in the sentence *"You used to call me on my **cell** phone"*, or *"The prisoner went back to his **cell**."*, the word embedding will be the **same**.
The second reason for the good performance of BERT is its **bidirectionality**. Indeed, the authors point out in their paper that another Transformer-based language representation model — **OpenAI GPT** (proposed in [RN18]) — achieves worse results than BERT due to it processing sequences **from left to right**.
The third and final reason for BERT's success is its usage of the Transformer encoder block. In this regard, a comparison can be made with ELMo (first published in [PNI+18]) — another type of *context-dependent bidirectional* word embeddings. ELMo embeddings were generated by using **bidirectional** LSTM networks, which do not capitalize on attention the way a Transformer encoder block does. In fact, the evaluations of the 11 NLP tasks in [DCLT18] show that even OpenAI GPT — a *unidirectional* Transformer-based language model — outperforms ELMo.

4.5.3 Tokenization

In order for BERT to be a good language representation model, it needs to support a **large** collection of words. Until now, we have only seen how to store **whole**

words in a *vocabulary* (refer to section 4.1). If those tokens are chosen to be *words*, however, (compared to, e.g., characters) it becomes unfeasible memory-wise to build a dictionary so huge, so that the model supports as many words as possible. In order to alleviate this problem, a **subword tokenization** approach can be used.
A **subword** is an arbitrary token — it could be a whole *word*, just a *character*, or a **frequent** sequence like *"ing"* (from, e.g., *playing* or *singing*). Being able to *break down* words into subwords provides a more flexible dictionary, as compound forms of words become trivially representable (e.g., *playing* → *"play + ing"*, *plays* → *"play + s"*).

There are different techniques of creating subword dictionaries. Two of the commonly used ones are called **Byte Pair Encoding**, or **BPE**, and **WordPiece**. WordPiece was first proposed by a team of Google engineers in [SN12], while BPE was presented by a team from the University of Edinburgh in [SHB15]. *WordPiece* is the approach, which BERT uses. The two techniques are, however, very similar — only differentiating in **one step**. For this reason, I will first introduce how *Byte Pair Encoding* works, as I find it is easier to understand [2]. At the end of the explanation, I will also show in which step WordPiece differs. For every *other* step of BPE, WordPiece functions the same way.

BPE takes the **size** of the vocabulary as a *hyperparameter*. It then goes through the **characters** of all text sequences of the provided collection of documents and appends them to the first version of the vocabulary. For example, assume that all words from the collection of documents are counted and grouped into:

$$('hug', 10), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)$$

Then, the initial version of the vocabulary contains all characters from the five

[2]The example for BPE was taken from `https://stackoverflow.com/a/70172964/18210589`

words[3]:

$$'h','u','g','p','n','b','s'$$

Next, BPE starts **merging** characters into **subwords**, based on their *frequency*. This effectively means that the BPE algorithm considers the **pair** combinations of all tokens in the vocabulary and keeps track of their *count*. Applied to the example data collection from above, the count tallies up to:

$$('hu', 15), ('ug', 20), ('pu', 17), ('un', 16), ('bu', 4), ('gs', 5)$$

As the most frequently encountered character pair, *'ug'* is appended to the vocabulary to get the second version:

$$'h','u','g','p','n','b','s','ug'$$

'u' and 'g' are also merged in all words in the collections that contain them:

$$('h''ug', 10), ('p''ug', 5), ('pun', 12), ('bun', 4), ('h''ug''s', 5)$$

The count of all combinations of tokens is once again calculated, and the top-scoring pair is appended to the vocabulary. This process is repeated until either the vocabulary size is hit, or there are no more words left to merge (usually — the former).

WordPiece shares the entire procedure as BPE, except for the **merging** step. Indeed, WordPiece also keeps track of the counts of subword pairs, but does not use the *frequency* as a decision criterion, but **highest likelihood**. The *highest likelihood* method merges token pairs, which have a bigger probability of being found *together*, than *apart*. Sadly, the original implementation of WordPiece has not been shared with the public, so we have no way of knowing how the merge step is implemented for certain.

[3]In practice, the initial vocabulary is bigger and contains ASCII and UTF characters.

The keen reader will have already noticed that the hyperparameter of vocabulary **size** is critical to a good subword dictionary. Too big and the vocabulary append every character and subword from every word in the collection, which imposes a *memory* problem. Too small and the vocabulary is nothing more than a *character-level* vocabulary. In the case of BERT, the vocabulary of the *uncased* version (i.e., input samples are always cast to lowercase before fed to the model) contains 30,522 subwords, wile the *cased* version (i.e., case-sensitive) contains 28,996 subwords. Additionally, the BERT authors chose to use "##" as a prefix to all subwords that are a continuation of a previous subword. For example, passing *Plovdiv* through the BERT tokenizer gives back *['P', '##lov', '##di', '##v']*.

4.5.4 Pre-training

Pre-training a model means nothing more than training a model on an additional task, before moving on to training on the *desired* task. In the case of the BERT model, the authors decided to **pre-train** on two tasks - **Masked-Language Modelling** and **Next Sentence Prediction**. The resulting model from training on the two tasks is then provided to end users, who can use its language representation abilities to accomplish a further task (e.g., OCR error detection).
The authors explain in [DCLT18] that the reason for adding the task of *Next Sentence Prediction* is to boost the performance of BERT on down-stream "multi-input" (i.e., the input contains two sequences $\boldsymbol{x}$ and $\boldsymbol{y}$) tasks, like for example **question answering**. As this is not relevant for this paper, I will instead focus on *Masked-Language Modelling*, which provides BERT its word representations. In fact, later *variations* of BERT completely omit the task of *Next Sentence Prediction* in their pre-training (e.g., RoBERTa in [LOG+19]), opting instead for large *batches*.

Masked-Language Modelling is, in essence, the same task that the *Continuous Bag of Words* model tried to solve in the original word2vec approach (see subsubsection 4.5.1. Indeed, it consists of predicting *masked* words in arbitrary text sequences —

e.g., *"I [MASK] Plovdiv"* → *"I love Plovdiv"*). The usage of a mask is necessary, as the self-attention module in the encoder block is *bidirectional*, which means that target words can "see themselves".

Two *huge* datasets were chosen for pre-training on both tasks — BooksCorpus (published, with 800 million words, and English Wikipedia, with 2,500 million words. In order to create the training samples for the Masked LM task itself, the authors chose to artificially "modify" 15% of all tokens in each sequence. Additionally, in order to avoid *overdependence* on the [MASK] meta token, it is only used in 80% of the cases. The leftover 20% is split evenly by two events — changing the token to a **random** one, and leaving the token **unchanged**.

5 String Operations

This chapter will cove several concepts, related to working with **strings**, or text sequences. In particular, section 5.1 will cover the concept of **string distance** and **similarity**. Afterwards, section 5.2 delves into **Needleman-Wunsch** — a global sequence alignment algorithm from the sequence of bioinformatics. Finally, section 5.3 will cover how one can use Needleman-Wunsch to align a **pair** and a **triple** of strings, so that the maximum number of characters match.

5.1 String Distance and Similarity

Finding the distance between two strings s and t (short for **source** and **target** string) involves determining the **minimal set** of **edit operations** to transform the one string into the other. An "edit operation" is a term for a single operation, which *edits*, or changes a symbol in a string. In 1966, Levenshtein offered a simple set of edit operations in [Lev66]. The set remains ubiquitous to this day and consists of **insertions** (e.g., "*Plovdiv*" → "*Plovediv*"), **deletions** (e.g., "*Plovdiv*" → "*Plodiv*") and **substitutions** (e.g., "*Plovdiv*" → "*Plofdiv*"). By using this set of operations, one can determine the **edit distance** between two strings.

It is worth noting that there also exist different *sets* of edit operations. Damerau also included **character transposition** in his work [Dam64] in 1964, which features *swapping* the places of two characters (e.g., "*Plovdiv*" -> "*Plodviv*"). I will, however,

be restricting myself to the base case of Levenshtein operations. For convention, I also note that the edit distance is sometimes called **Levenshtein distance**.

The process of transforming some string s into another string t involves changing characters from s with *edit operations*, until one gets t. This is also the reason the starting string s is called the **source** string, while the resulting string t is called the **target** string. Box 58 shows three different sequences (also called: **chains**) of edit operations, which transform the source string *Plovdiv* to the target string *loved*. It is obvious from the example that there can be **multiple** different optimal edit combinations. The *edit distance* then is the number of operations, which it takes to complete the string transformation in any one of the **optimal** edit combinations.

$$\begin{aligned} &\textbf{Plovdiv} \rightarrow \text{lovdiv} \rightarrow \text{lovdi} \rightarrow \text{lovd} \rightarrow \textbf{loved} \\ &\textbf{Plovdiv} \rightarrow \text{lovdiv} \rightarrow \text{lovdi} \rightarrow \text{lovei} \rightarrow \textbf{loved} \\ &\textbf{Plovdiv} \rightarrow \text{Plovediv} \rightarrow \text{Plovedi} \rightarrow \text{Ploved} \rightarrow \textbf{loved} \\ &\dots \end{aligned} \tag{58}$$

In practice, the distance between two strings is found by using *dynamic programming*. Dynamic programming is a paradigm in computer science for solving problems that can be broken down into smaller "sub-problems" and **recursively** tackled. In the case of calculating the Levenshtein distance, this means recursively finding the distance between incrementally bigger substrings until reaching the end result. [Dam64] is the first known publication to provide a description of such an algorithm, while [Nav01] offers a more thorough dive into the topic, with more rigorous proofs.
A key observation, which makes this process make sense, is equation 59. In it, ED() is the function for calculating the edit distance, and ED(x_n, y_m) indicates calculating the edit distance between the **substrings** $x_1x_2 \dots x_n$ and $y_1y_2 \dots y_m$.

$$ED(s_i, t_j) \leq ED(s_{i-1}, t_{j-1}) + 1 \tag{59}$$

Proving 59 is simple. Assume we already know $ED(s_{i-1}, t_{j-1}) = x$. Then, there are four scenarios for $ED(s_i, t_j)$:

1. Source string s is **longer** than target string $t \implies ED(s_i, t_j) = x + 1$, because of **deletion** of last symbol

2. Source string s is **shorter** than target string $t \implies ED(s_i, t_j) = x + 1$, because of **addition** of last symbol

3. Strings have the same length, but **different** last symbols $\implies ED(s_i, t_j) = x+1$, because of **substitution** of last symbol

4. Strings have the same length, and **same** last symbols $\implies ED(s_i, t_j) = x$, because nothing has changed

Dynamic programming algorithms use rule 59 to progressively determine the edit distance between s and t. Indeed, the algorithms work "**bottom-to-top**", by first calculating $ED(s_0, t_1), \ldots, ED(s_0, t_m)$ and $ED(s_1, t_0) \ldots, ED(s_n, t_0)$, where $|s| = n$, $|t| = m$, and x_0 represents the **empty string**. For these calculations, the trivial rule 60 is used.

$$ED(x, y_0) = ED(y_0, x) = |x| \tag{60}$$

After determining the base cases, the dynamic programming algorithms continue calculating the edit distance values of bigger substrings from both strings, using the formula from 61:

$$ED(s_i, t_j) = min \begin{cases} ED(s_{i-1}, t_j) + 1 \text{ (delete a character from s)} \\ ED(s_i, t_{j-1}) + 1 \text{ (insert a character from t)} \\ ED(s_{i-1}, t_{j-1}) + 1 \text{ (substitute a character in s from t)} \\ ED(s_{i-1}, t_{j-1}) \text{ (nothing has changed)} \end{cases} \tag{61}$$

	ϵ	**l**	**o**	**v**	**e**	**d**
ϵ	0	1	2	3	4	5
P	1	1	2	3	4	5
l	2	1	2	3	4	5
o	3	2	1	2	3	4
v	4	3	2	1	2	3
d	5	4	3	2	2	2
i	6	5	4	3	3	3
v	7	6	5	4	4	4

Table 1: Result table from running a dynamic programming algorithm for finding the edit distance between "*Plovdiv*" and "*loved*"; ϵ represents **blank string**; yellow path is an **optimal** chain of edit operations

Table 1 shows the table of edit distances, which dynamic programming algorithms build to find the edit distance between $s = Plovdiv$ and $t = loved$. In it, a cell with coordinates (i, j) represents $ED(s_i, t_j)$. As such, the bottom-right corner of the table is the same as $ED(Plovdiv, loved)$, which is indeed **4**.

The yellow path, which is marked in the table, is one of the optimal chain of edit sequences, which can be used to transform "*Plovdiv*" into "*loved*". The transformation can be carried out by following these rules:

1. If moving **diagonally** from (i, j) to (i+1, j+1), substitute character $\#i$ in the source string with character $\#j$ from the target string (or do nothing if the characters are identical)

2. If moving **to the right** from (i, j) to (i, j+1), insert character $\#j$ into the source string

3. If moving **downwards** from (i, j) to (i+1, j), delete character $\#i$ from the source

In the case of the path in table 1, the transformation process is shown in box 62:

$$\begin{array}{c} \text{Plovdiv} \rightarrow \text{-lovdiv} \rightarrow \text{-lovdiv} \rightarrow \text{-lovdiv} \rightarrow \text{-lovdiv} \rightarrow \text{-lov-iv} \rightarrow \text{-lov-ev} \rightarrow \text{-lov-ed} \\ (0,0) \rightarrow (1,0) \quad \rightarrow (2,1) \quad \rightarrow (3,2) \quad \rightarrow (4,3) \quad \rightarrow (5,3) \quad \rightarrow (6,4) \quad \rightarrow (7,5) \end{array} \tag{62}$$

5.1.1 String Similarity

The edit distance can theoretically also be used to measure *similarity* between strings. If 2 strings have a distance of 0, then they are **identical**. If they have an edit distance **greater than 0**, then they have different symbols between them. The higher the distance, the more different two string are.

Using the raw edit distance, however, does not give a concrete idea of what **fraction** of the symbols in a string are the same. To remedy this, [MV93] proposes using the **normalized edit distance**, which is also knows as the **Levenshtein ratio**. The Levenshtein ratio can be calculated with equation 63 and returns a value in the range [0; 1]:

$$ratio(s,t) = ED(s,t)/max(|s|,|t|) \tag{63}$$

5.2 Needleman-Wunsch

Needleman-Wunsch is a **global alignment** algorithm from the field of bioinformatics, first published in [NW70] in the year 1970. It was originally created to solve the problem of **sequence alignment** — the task of aligning the nucleotide bases of two proteins, so that their similarities can be inspected for genetic similarity.

The Needleman-Wunsch algorithm is a **dynamic programming** algorithm, and it is pseudocode is shown in listing 5.1. At its core, the algorithm tries to **maximize** the **alignment score** between the two strings.

The algorithm accepts 4 parameters: the two sequences that need to be aligned (*str1*

Listing 5.1: Pseudocode of the **Needleman-Wunsch** algorithm

```
needleman_wunsch(str1, str2, sub_matrix, gap_penalty):
  n, m = len(str1), len(str2)

  # +1 to leave space for the empty string.
  matrix = [[0 for _ in range(n+1)] for _ in range(m+1)]
  origins = [[None for _ in range(n+1)] for _ in range(m+1)]

  # Initialize first row and column by accumulating gap pen.
  for i in range(1, n):
    matrix[i][0] = i * gap_penalty
    origins[i][0] = (i-1, 0)
  for j in range(1, m):
    matrix[0][j] = j * gap_penalty
    origins[0][j] = (0, j-1)

  # Populate matrix.
  for i in range(1, n):
    for j in range(1, m):
      matrix[i][j] = max(
        # Add gap in second string.
        matrix[i-1][j] + gap_penalty,
        # Add gap in first string.
        matrix[i][j-1] + gap_penalty,
        # Concatenate nucleotides from both sequences.
        matrix[i-1][j-1] + sub_matrix[str1[i]][str2[j]]
      )
      origins[i][j] = coordinates of best score

  return traceback_alignment(origins, str1, str2)
```

and *str2*), a *substitution matrix*, and a *gap penalty*. The substitution matrix and the gap penalty are the two parameters that influence how the alignment is going to be done. The *gap penalty* is a numeric value — usually, a *negative* number. Every time a "gap" is inserted into the alignment, the gap penalty is subtracted from the overall *alignment score* (see lines 10, 13, 20, and 21 in listing). The *substitution matrix* is motivated by the field of bioinformatics and contains values for the substitutions of different nucleotide bases. As an example, if a nucleotide base is **not** changed (A in sequence 1 stays A in sequence 2), then the assigned value may be 1. On the other hand, if the base is changed (A in sequence 1 becomes G in sequence 2), the assigned value might be -1. By providing a way for the alignment to be directly influenced by the substitution matrix (see line 22 in the listing), researchers can assign *weights* that have biological justification.

The algorithm follows the basic formula of a **dynamic programming** algorithm. It first creates an (n+1)×(m+1) matrix, in which the alignment scores of different *substrings* of the sequences will be saved (lines 5–14). Next, it goes through the cells of the matrix and populates their values based on the recursive formula, shown in lines 19-26. The **origin** — meaning the coordinates of the cell, from whose score the current one was calculated — of each cell is also saved. After filling in all the entries in the matrix, the origin matrix is passed to a helper function "*traceback_sequence*", which builds the two aligned sequences.

The code for "*traceback_sequence*" is shown in listing 5.2. The algorithm starts in the bottom-right corner of the supplied origin matrix, which was built from the Needleman-Wunsch algorithm (see previous paragraph), as shown in lines 2 and 3. Afterwards, the algorithm follows the **origin coordinates** of each cell. If the origin of a cell sits on the diagonal, then the bases of both sequences are appended (lines 5-7). If it lies on the left of the current cell, then the base of the **second** sequence is appended, but a *gap* in the first (lines 11-13). And, if it lies above the current cell, the base of the **first** sequence is appended, while a **gap** is appended to the second (lines 8-10). At the end of the algorithm, the aligned sequences are reversed, as the

Listing 5.2: Helper function pseudocode for creating alignments from a **Needleman-Wunsch** matrix

```
traceback_alignment(origins, str1, str2):
  aligned_str1, aligned_str2 = ''
  i, j = len(str1), len(str2)
  while (i>0 or j>0):
    if (i > 0 and j > 0) and origins[i][j] == (i-1, j-1):
      aligned_str1 += str1[i]
      aligned_str2 += str2[j]
    elif i > 0 and origins[i][j] == (i-1, j):
      aligned_str1 += str1[i]
      aligned_str2 += '-'  # Gap in second string.
    elif j > 0 origins[i][j] == (i, j-1):
      aligned_str1 += '-'  # Gap in first string.
      aligned_str += str2[j]

    i, j = origins[i][j]

  return aligned_str1[::-1], aligned_str2[::-1]
```

algorithm essentially worked from the back to the front — and returned.

5.3 String Alignment

String alignment is the problem of *arranging* strings, such that as many symbols across the strings match with each other. To illustrate this with an example, consider the four alignments in box 64 of the string "*Plovdiv*" and "*loved*", with '-' being a placeholder blank character.

Plovdiv	Plovdiv	Plovdiv	Plov-div	(64)
loved--	--loved	-loved-	-loved--	

It is easy to visually see in the example that the right-most alignment is the best — *Plov-div* and *-loved--*. The right-most alignment makes it so the two strings have

	ϵ	**l**	**o**	**v**	**e**	**d**
ϵ	0	0	0	0	0	0
P	0	0	0	0	0	0
l	0	1	1	1	1	1
o	0	1	2	2	2	2
v	0	1	2	3	3	3
d	0	1	2	3	3	4
i	0	1	2	3	3	4
v	0	1	2	3	3	4

Table 2: Result table from applying alignment-specialized **Needleman-Wunsch** on "*Plovdiv*" and "*loved*"; ϵ represents **blank string**
yellow path is an **optimal** chain of operations

4/7 shared characters, while the other three have *0/7*, *0/7* and *3/7* respectively. It is important to notice that aligning strings is **not** guaranteed to keep the length of the bigger string. Indeed, the best alignment also inserts a blank '-' character in the middle of the word, in order to maximize the number of shared characters.

The acute reader will already have made the connection of the string alignment problem with the **Needleman-Wunsch** algorithm. Indeed, it has been shown that is possible to set the gap penalty and the substitution matrix of the algorithm, so that Needleman-Wunsch minimizes the **edit distance** between two arbitrary strings (see [Sel74]).

For the purpose of **string alignment**, however, the main task is different: having as many symbols as possible between the two strings **match**. This can be achieved by setting the **gap** and **mismatch** penalties to be **0**, and the value for **matches** to be **1**. This set of values forces the Needleman-Wunsch algorithm to only prioritize *matching* characters between two strings, which leads to an optimal pair alignment. Of course, as with the edit distance algorithm from section 5.1, it is possible to have **more than one** optimal alignments to a pair of arbitrary strings. In that case, one

can just compute the **Levenshtein distance** of each alignment and return the one with the lowest distance.

To verify the example from the beginning of the section ("*Plovdiv*" and "*loved*"), table 2 shows the matrix, generated by Needleman-Wunsch that is set to optimize the number of matched characters. The reader can verify for themselves that applying the helper trace back algorithm from listing 5.2 results in the optimal alignment *Plov-div* and *-loved--*.

String Triple Alignment

The Needleman-Wunsch algorithm can be extended to handle **more than two** sequences. This is going to prove useful down-the-line in the evaluation of error correction on character-level. The extension of the algorithm leaves the *structure* of the code from listings 5.1 and 5.2 the same, but adds more *cases*[1]. The additional cases come from running the algorithm in a **hypercube** (i.e., in **three** dimensions), rather than a 2D matrix. In particular, the 3D extension adds a new dimension, whose elements I refer to as "ailes". With the addition of *ailes*, each cell in the hypercube has **seven** possible origin cells:

1. From the diagonal (i, j, k) → (i-1, j-1, k-1)
2. From above (i, j, k) → (i-1, j, k)
3. From the left (i, j, k) → (i, j-1, k)
4. From "backwards" (i, j, k) → (i, j, k-1)
5. From above-left (i, j, k) → (i-1, j-1, k)

[1]The full code is available to check under src/alignment_utils.py in the provided code.

6. From “backwards-above” (i, j, k) → (i-1, j, k-1)

7. From “backwards-left” (i, j, k) → (i, j-1, k-1)

The other major change to the 3D extension of the algorithm concerns the **cost function**. The *cost function* $cost(str1_i, str2_j, str3_k)$ calculates the overall **cost** of having three symbols aligned in a particular way. It relies, as in the 2D version, on a **gap penalty** and values for **mismatches** and **matches**. It then calculates the **sum-of-pairs** (motivated from [Edg04]) between all three combinations — i.e., it determines the *cost* for each pair of symbols (depending on gap/match/mismatch) and sums them up. The resulting **alignment cost** is then used as the *score* of that particular symbol alignment (see lines 20-25 in listing 5.1). As an example, assuming a gap and mismatch cost of **0**, and a match value of **1**, the character triple of (l, l, l) would have an alignment score of **3** (1 + 1 + 1), while the character triple of (l, t, None) (the last is possible if the third string is shorter and the alignment process already went through it) would have a score of **0** (0 + 0 + 0).

After running the extended Needleman-Wunsch algorithm on a triple of strings, one might get multiple possible alignments. In that case, the **average** Levenshtein distance between the three pairs of aligned strings is computed, and the alignment with the highest average is returned. The *averaging* operation is used to discredit such alignments, where two sequences are much more alike between each other than they are to the last one.
It is important to note that running the extended Needleman-Wunsch algorithm is **costly**. In particular, assuming the longest string of the triple has length n, the 3D algorithm has a time and space complexity of $\mathcal{O}(n^3)$. There exist other algorithms for **approximate** multiple sequence alignment in bioinformatics, which can carry out the task faster — refer to [CMC+15] — but they are out of the scope of this paper.

"Fast" pair alignment

I mentioned in the previous subsection 5.3 that the extended Needleman-Wunsch algorithm for *three* strings is **expensive** computationally. Indeed, this computational cost is also present when aligning string *pairs* with the algorithm, as the time and space complexity of the algorithm is $\mathcal{O}(n^2)$. This can become a problem for aligning *large* text sequences, like the ones from the datasets from the ICDAR competitions on Post-OCR correction (refer to chapter 2). The long sequences there make the data pre-processing scripts (explained later in chapter 7) for the datasets take an *extremely* long time, made worse by the Python implementation[2].

In order to circumvent this issue, I will use an alternate algorithm to align the text sequences of the samples in the data preprocessing scripts — an algorithm I will call *"fast" pair alignment*. The fast pair alignment capitalizes on a C extension for Python called **Levenshtein-C**[3]. The package is called a *"C extension"*, as it was implemented in the **C** language, but made usable for Python programs as well. Although the package uses the same dynamic programming solution in the background, the implementation in C makes the execution of the Needleman-Wunsch algorithm much faster.

The algorithm first determines the minimum set of *Levenshtein operations*, which are required to transform one sequence into another. Then, an alignment between the two sequences can be constructed by going through all **delete** and **insert** operations and doing the following:

In the presence of a *delete* operation, insert a padding *'-'* symbol at the index of deletion in the **correct** string. The intuition behind this is that a deletion operation signals that the correct sequence is one symbol **shorter** than the erroneous one, and as such needs to be padded. In the presence of an *insert* operation, on the other hand,

[2]Python is a notoriously slow programming language, due to the same overhead structures, which make it easy to use. This can be seen in the benchmark vs C on `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gcc.html`

[3]The homepage of the extension is: `https://github.com/ztane/python-Levenshtein`

one needs to add a padding *'-'* symbol at the index of insertion in the **erroneous** string. The reasoning for this is *opposite* of deletion — if a character has to be inserted in the erroneous string, then the correct one is one symbol **longer**.

The results from the *"fast" alignment* algorithm are equivalent to running the Needleman-Wunsch algorithm for the **minimization** of the *edit distance* between two strings. In it, the accompanying *gap* and *mismatch* penalties are then set to be **1**, while the score for *matches* is set to be **0**. As such, this alignment algorithm does not guarantee finding the **optimal** alignment between two sequences, but its results are often a good approximate (e.g., the alignment of *"Plovdiv"* and *"loved"* is (*"Plovdiv"*, *"'-lov-ed'"*)).

6 Baseline algorithm

As previously discussed in chapter 2, a simple approach to Post-OCR correction is using a dictionary. This involves compiling a huge list of valid words and comparing each one of them with erroneous tokens to find the most suitable correction candidate. In practice, iterating through the entries of such a collection every time a token has to be corrected would render the program unusable because of how slow it is. To counter-act that, I am going to use a data structure from the field of Information Retrieval — Q-gram indices.

6.1 Q-Grams

Before moving to Q-gram indices, one first has to know the concept of Q-grams. Let $q \in \mathbb{N}_{>0}$. Then, a Q-gram of length q of a word is just an arbitrary substring with the specified length. If we consider all Q-grams of length 3 of the word "Plovdiv", we would get:

$$\text{Plovdiv} \rightarrow [\text{Plo, lov, ovd, vdi, div}]$$

An important property is that there are always $|x| - q + 1$ Q-grams of length q for an arbitrary string x. This comes in handy when Q-grams are inserted into the context of fuzzy search.

6.2 Fuzzy Search

Fuzzy searching is another name for the task of **approximate string matching**. A widespread example of fuzzy searching is search engines trying to correct user queries (usually found after "Did you mean:"). Formally, the task requires finding a correction of an erroneous query by minimizing the **edit distance**. This description should immediately draw parallels to Post-OCR correction, but I will be explaining the connection in a later section.

Fuzzy search is expensive, given that good results are desired. If we denote the length of a dictionary with correct words as n and the time to compute the edit distance between two strings x and y as $t(x, y)$, then the total time $T(x)$ of finding a correction for an erroneous token x is:

$$T(x) = \sum_{i=1}^{n} t(x, y_i) \leq n * t(x, \arg\max_{y} |y|)$$

The naive approach from above naturally seems unsatisfying. There should exist only a given subset of words that has to be considered as potential correction candidates. Words like *cat* or *car* look like valid substitutions for "cad", while *dog* or *carpet* do not. If there is a way for the latter candidates to be pruned early in the process, the whole operation will become much faster. This is exactly where Q-grams fit into the puzzle.

We can use the number of common Q-grams as an auxiliary metric for similarity between strings. It is intuitive that similar strings with a small edit distance should have more shared Q-grams. Let us consider an example with an erroneous token x = botle and two correction candidates: y_1 = battle and y_2 = cattle, by using 2-grams (Q-grams of length 2):

$$\text{x = botle} \rightarrow \text{[bo, ot, tl, le]}$$
$$y_1 = \text{battle} \rightarrow \text{[ba, at, tt, tl, le] (2 shared Q-grams)}$$

$$y_2 = \text{cattle} \rightarrow [\text{ca, at, tt, tl, le}] \text{ (2 shared Q-grams)}$$

Looking at the number of shared 2-grams, we can see that both replacements have two - "tl" and "le". This, however, seems flawed, as both the input "botle" and the first candidate "battle" start with the letter *b*. This was not taken into consideration in the example above. Upon further inspection, we can see that all letters in all words appear in exactly 2 q-grams (identical to length q) *except* the start and end letters. In order to fix this issue, a **padding** of length $q - 1$ is appended to both sides of the strings. Returning to the example again, it now properly shows y_1 as the most suitable replacement:

$$\text{x} = \$\text{botle}\$ \rightarrow [\$\text{b, bo, ot, tl, le, e}\$]$$
$$y_1 = \$\text{battle}\$ \rightarrow [\$\text{b, ba, at, tt, tl, le, e}\$] \text{ (4 shared Q-grams)}$$
$$y_2 = \$\text{cattle}\$ \rightarrow [\$\text{c, ca, at, tt, tl, le, e}\$] \text{ (3 shared Q-grams)}$$

What is left to determine is a reasonable lower threshold of common Q-grams for a given maximum edit distance δ. For this, consider the padded versions x_p and y_p of two arbitrary strings x and y. The padding alters the number of Q-grams from $|x| - q + 1$ to $|x| + q - 1$, because it adds $q - 1$ Q-grams to both sides of the string:

$$|x| - q + 1 + 2 * (q - 1) = |x| - q + 2q - 2 + 1 = |x| + q - 1$$

As pointed out in the last paragraph, each character appears exactly **q** times in all Q-grams of a word. From this, it follows that at most $\leq \delta * q$ Q-grams must be affected for a maximum edit distance of δ. To show this, we will use a proof by induction. Let x be an arbitrary string of length n and the Q-gram size $q = 3$. The specified value of q lets the visualization of the proof be easier to understand, but it holds for any arbitrary value.

Consider the **base case** of $\delta = 1$ and let the edit operation happen at position t. In

the case of an insertion at position t, let x and the modified string $x‘$ be:

$$x = x_1 x_2 \dots x_{t-1} x_{t+1} \dots x_n$$
$$x‘ = x_1 x_2 \dots x_{t-1} x_t x_{t+1} \dots x_n$$

Then, consider how the Q-grams of both strings would look like, as shown in eq. (65). It is easy to see that the Q-grams of the two strings can differ by $\leq q$ (in the case of the visualization, 3) — and exactly by the Q-grams, which contain the new symbol, inserted at position t. The $\leq$ property holds, as successive repeating letters can lower the difference (e.g., see vs seee).

$$\begin{aligned} &\text{3-grams of } x : \{\dots \\ &\quad (x_{t-3}x_{t-2}x_{t-1}), \\ &\quad (x_{t-2}x_{t-1}x_{t+1}), \\ &\quad (x_{t-1}x_{t+1}x_{t+2}), \\ &\quad (x_{t+1}x_{t+2}\dots x_{t+q}), \\ &\dots\} \end{aligned} \quad \Longleftrightarrow \quad \begin{aligned} &\text{3-grams of } x‘ : \{\dots \\ &\quad (x_{t-3}x_{t-2}x_{t-1}), \\ &\quad (x_{t-2}x_{t-1}x_t), \\ &\quad (x_{t-1}x_t x_{t+1}), \\ &\quad (x_t x_{t+1} x_{t+2}), \\ &\quad (x_{t+1}x_{t+2}\dots x_{t+q}), \\ &\dots\} \end{aligned} \tag{65}$$

Next, consider the case of **deletion** at position t. Mirrored from the previous case, let x and the modified string $x‘$ be:

$$x = x_1 x_2 \dots x_{t-1} x_t x_{t+1} \dots x_n$$
$$x‘ = x_1 x_2 \dots x_{t-1} x_{t+1} \dots x_n$$

Then, the Q-grams of x and $x‘$ are, also, the exact opposite of eq. (65). As is the case there, it follows that both strings can have $\leq q$ different Q-grams.

For **substitution** at position t, regard the definitions of x and the modified string

x‘:

$$x = x_1 x_2 \dots x_{t-1} x_t x_{t+1} \dots x_n$$
$$x‘ = x_1 x_2 \dots x_{t-1} x‘_t x_{t+1} \dots x_n$$

Both strings then have the same amount of Q-grams, as visualized in eq. (66). Similar to the previous two cases, the set of Q-grams is disjoint in the q Q-grams, which include the replaced symbol at position t. Different from the cases of insertion and deletion, however, a *substituted* character guarantees that the three Q-grams, marked in red in eq. (66), will be different, and thus the $<$ property does not hold.

$$\begin{array}{lcl} \text{3-grams of } x : \{\dots & & \text{3-grams of } x‘ : \{\dots \\ \quad (x_{t-3}x_{t-2}x_{t-1}), & & \quad (x_{t-3}x_{t-2}x_{t-1}), \\ \quad (x_{t-2}x_{t-1}x_t), & & \quad (x_{t-2}x_{t-1}x‘_t), \\ \quad (x_{t-1}x_t x_{t+1}), & \Longleftrightarrow & \quad (x_{t-1}x‘_t x_{t+1}), \\ \quad (x_t x_{t+1}x_{t+2}), & & \quad (x‘_t x_{t+1}x_{t+2}), \\ \quad (x_{t+1}x_{t+2}\dots x_{t+q}), & & \quad (x_{t+1}x_{t+2}\dots x_{t+q}), \\ \dots\} & & \dots\} \end{array} \tag{66}$$

For the induction step, assume that $\leq \delta * q$ Q-grams are affected with an edit distance δ. A key observation to make is that the worst case can be achieved only if the edit operations are spread out and occur every $q-1$ characters. If two modified symbols are found in a proximity of $q-1$ characters, their affected Q-grams overlap. This is shown in eq. (67), where the left side visualizes the Q-grams of an arbitrary substring $x_1 = x_i x_{i+1} x‘_{i+2} x_{i+3} x_{i+4} x‘_{i+5} x_{i+6} x_{i+7}$ with evenly spread out edit operations and the right side - $x_2 = x_i x_{i+1} x‘_{i+2} x_{i+3} x‘_{i+4} x_{i+5} x_{i+6} x_{i+7}$ with overlapping ones. I use

x‘ to mark the symbols, introduced by edit operations.

$$
\begin{array}{ll}
\text{3-grams of } x_1 : \{\ldots & \text{3-grams of } x_2 : \{\ldots \\
\quad (x_i x_{i+1} x\text{‘}_{i+2}), & \quad (x_i x_{i+1} x\text{‘}_{i+2}), \\
\quad (x_{i+1} x\text{‘}_{i+2} x_{i+3}), & \quad (x_{i+1} x\text{‘}_{i+2} x_{i+3}), \\
\quad (x\text{‘}_{i+2} x_{i+3} x_{i+4}), & \quad (x\text{‘}_{i+2} x_{i+3} x\text{‘}_{i+4}), \\
\quad (x_{i+3} x_{i+4} x\text{‘}_{i+5}), & \quad (x_{i+3} x\text{‘}_{i+4} x_{i+5}), \\
\quad (x_{i+4} x\text{‘}_{i+5} x_{i+6}), & \quad (x\text{‘}_{i+4} x_{i+5} x_{i+6}), \\
\quad (x\text{‘}_{i+5} x_{i+6} x_{i+7}) & \quad (x_{i+5} x_{i+6} x_{i+7}) \\
\ldots\} & \ldots\}
\end{array}
\quad \textbf{vs.} \quad (67)
$$

As showcased in eq. (67), having spread out edit operations at every $q-1$ characters is the worst-case scenario. But, as shown already in the induction base case, every edit operation can affect, by itself, $\leq q$ Q-grams. This means that in the worst case of δ spread out edit operation modifications, the maximum amount of Q-grams changed is $\leq \delta * q$. $\square$

Now, by using the proof from above, we can calculate the minimum number of common Q-grams ($comm(x, y)$) between x_p and y_p:

$$
\begin{aligned}
comm(x_p, y_p) &\geq max(|x_p|, |y_p|) + q - 1 - \delta * q \\
&\geq max(|x_p|, |y_p|) - 1 + (1 - \delta) * q \\
&\geq max(|x_p|, |y_p|) - 1 - (\delta - 1) * q
\end{aligned}
\quad (68)
$$

6.3 Q-Gram Index

Q-indices are data structures, which leverage eq. (68) in order to execute fuzzy searches in reasonable time. They are built by first determining the Q-gram size q and then processing a collection of words. Each word is then processed by assigning

a numerical ID to it and storing it in a flat list, where the index corresponds to its ID. Afterwards, all Q-grams of the token are extracted and inserted in a "index" (i.e. mapping), which maps each Q-gram to a list of integers — the numerical IDs of all words that contain the given Q-gram. This trade-off of using memory for the expense of speed during inference is what makes the dictionary approach feasible in practice. If we are using a 3-index with the word collection of {"plovdiv", "love"}, the status of the Q-index after inserting the words is shown in Listing 6.1.

Listing 6.1: Status of hypothetical Q-index

```
{
  "$p": [0],
  "pl": [0],
  "lo": [0, 1],
  ...
}
```

Pseudocode for the correction procedure of erroneous tokens with a certain maximum edit distance follows immediately below in Listing 6.2. After trimming the candidates with the auxiliary metric from eq. (68), the correction with the lowest edit distance and biggest frequency is returned. Frequency denotes the number of times a given word was inserted into the Q-index, when it was built. The higher the frequency, the more commonly used the word is.

The helper function *merge_entry_lists* is described in Listing 6.3. In short, it aggregates the entry lists of all matched Q-grams and returns the entries (words, which the Q-index has saved and knows), which have at least one common Q-gram.

Listing 6.2: Pseudocode for error correction with a Q-index

```
correct(q_index, q_size, erroneous, max_dist):
  q_grams = get_q_grams(erroneous, q_size)

  entry_lists = []
  for q_gram in q_grams:
    if q_gram in q_index:
        entry_lists.append(q_index[q_gram])
  candidates = merge_entry_lists(entry_lists)

  for candidate in candidates:
    n, m = len(erroneous), len(candidate.word)
    threshold = max(n, m) - 1 - (max_dist - 1) * q_size
    if candidate.shared_q_grams < threshold:
      remove candidate from consideration

  return candidate w/ lowest edit dist. and biggest freq.
```

Listing 6.3: Pseudocode for helper function *merge_entry_lists*

```
merge_entry_lists(entry_lists):
  flat_list = flatten(entry_lists)
  flat_list.sort()

  res = [(1, flat_list[0])]
  for entry in flat_list:
    if entry != res[-1][1]:
      res.append(1, entry)
      continue

    res[-1][0] += 1

  return res
```

7 Error Correction Models

This chapter will explain the exact mechanics of framing the Post-OCR correction problem as an NMT one. In particular, section 7.1 covers the format of the data, which will be fed into the *encoder-decoder* models, as well how it is generated. Afterwards, 7.2 introduces the **character-level** vocabulary, which is used to encode and decode the error correction samples. Finally, 7.3 illustrates how the data flows through the encoder-decoder networks.

7.1 Data

The data for error correction is split into different groups with respect to the assigned **context size**. The *context size* is a tuple of integers (c_p, c_s), which controls how many tokens there are to the *left* (c_p) and *right* (c_s) side of the **focus** token. There is always exactly **one** focus token per error correction sample, and it is encased in meta $<\boldsymbol{TGT}>$ tokens (*meta tokens* will be explained more in-depth in the upcoming section 7.2 on preprocessing).

OCR: "and comfort me <TGT>too<TGT> Ihe bright shining"
GT: "and comfort me <TGT>too,<TGT> Ihe bright shining"

OCR:"<TGT>Zn<TGT> this"
GT: "<TGT>In<TGT> this"

OCR: "<TGT>w if e<TGT>"
GT:"<TGT>w@ife,<TGT>"

OCR: "<TGT>andtheir<TGT>"
GT: "<TGT>and their<TGT>"

Box 7.1: Error correction samples with varying context sizes

Box 7.1 offers examples of error correction samples with varying context sizes - **(3, 3)**, **(1, 1)**, and **(0, 0)**. Context size *(0, 0)* is a special case, which I will refer to **isolated** correction. For a better visualization of the different parts of an error correction sample, I will use blue to mark the *preceding context tokens*, purple to mark the *succeeding context tokens*, and red to mark the *focus token*.

As seen by lines 1 and 2 in the box, the sequence *"and comfort me **too Ihe** bright shining"* contains two erroneous tokens. Despite having multiple mistakes, however, the error correction samples that get created by the sequence always feature exactly *one* focus word.

Additionally, the context size is not always *strictly* obliged to, when there are not enough tokens. For example, line 3 in box 7.1 was generated with context size *(1, 1)*, but there were no context tokens available to the left of the focus word "*<TGT>Zn<TGT>*". In cases like that, the error correction sample is still considered valid — the missing context is just padded to match the sequence length of the rest of the sample (if used in a batch).

7.1.1 Data generation

Aligned OCR: Chortts. - Ri choo ral, on mischief that girl was bent.
Aligned GT: Chor@us.@-@Ri choo ral, on mischief that girl was bent.
Extracted token pairs: [("Chortts. - Ri", Chor@us.@-@Ri)]
Isolted. corr. sample: ("<TGT>Chortts. - Ri<TGT>", <TGT>Chor@us.@-@Ri<TGT>)
Context. corr. sample (1, 1):
("<TGT>Chortts. - Ri<TGT> choo", <TGT>Chor@us.@-@Ri<TGT> choo)

Box 7.2: Creation of error correction samples

Box 7.2 illustrates how error correction samples are generated. In particular, one first needs to extract all *pairs* of **erroneous** tokens, together with their **correct** versions, out of an arbitrary sequence. For this, the sequence is stripped down to its **sentences** by using the pretrained *"en_ core_ web_ sm"* model with spaCy[1]. It is worth noting that the sentence *splitting* is not required for the generation of error correction samples, but makes it easier to generate error detection data down-the-line (refer to chapter 8).

After splitting the sequence pair into *pairs* of sentences, the words of each sentence pair are aligned with each other, with the alignment algorithm shown in 5.3. The alignment then allows implementing the approach from [AC18], where **token pairs** are extracted from the two sentences by splitting on **matching** whitespace characters. The split on *matching* whitespace characters is **vital** to the correct extraction of token pairs. It also makes it possible to extract erroneous tokens with **indentation** mistakes. Box 7.2 illustrates this with an example. Consider the naive approach of splitting the *erroneous* sequence by whitespace characters, and using the **indices** of each token to extract the corresponding *correct* tokens from the erroneous sequence. Then, instead of having the correct token pair (*"<TGT>Chortts. - Ri<TGT>"* and *"<TGT>Chor@us.@-@Ri<TGT>"*), we would end up with **three** *incorrect* pairs — (*"<TGT>Chortts.<TGT>"*, *"<TGT>Chor@us."*), (*"-"*, *"-"*), and (*"Ri"*, *"Ri"*). By

[1]spaCy is an NLP package with pre-trained models: `https://spacy.io/`

splitting only on the matching whitespace character after *"Ri"*, however, the erroneous token (with an absent whitespace character) can correctly be extracted.

The next step of generating error correction samples involves **filtering** the extracted token pairs. The idea behind the *filtering* process is to eliminate all token pairs, which are too **"dirty"** to be corrected. Such token pairs might include tokens that were misaligned in their original sequences, erroneous tokens with **no** respective ground truth, and others. A more in-depth exploration of all employed tokens is offered in the following subsection 7.1.2.

All token pairs that survive the filtration process are valid and are turned into error correction samples. By default, the data preprocessing scripts create one **isolated** correction sample from each token pair, as well as one **context** correction sample with *context size* **(1, 1)** (i.e., one preceding and one succeeding token). The scripts also support a command-line argument, which can be used to change the context size to arbitrary integer values ≥ 0.
For the generation of an *isolated* correction sample, both the erroneous token and the correct token are *encased* with $<TGT>$ meta tokens. The encased erroneous token is then used as an **input** sample, while the correct token as an **output** sample. One might argue the meta tokens are, in this case, superfluous, as the **entirety** of the erroneous token (i.e., the **input** sample) is open for correction. While this is true, the $<TGT>$ tokens are the **link** between the error correction models and the error detection model, which is going to be introduced in the upcoming chapter 8. As such, the meta tokens are present across all error correction samples, regardless of context size.
As far **context** correction samples are concerned, the supplied *context size* (c_p, c_s) determines how many tokens will be appended to the left and right of the **target** (or, **focus**) token. The target token itself is, again, encased in $<TGT>$ meta tokens, in order to signalize to the model which one of the tokens from the input sample needs to be corrected. The *context tokens* are extracted from the *erroneous* sequence, from

which the target token also comes. Although this might lead to the context tokens *themselves* having mistakes (see box 7.1), the error correction model only needs to handle the *focus* token in the middle.
The reader might question why it is necessary to include the context of a *context correction sample*, instead of only predicting the correction of the focus word. This decision was influenced by Amrhein et al.'s paper [AC18], in which they point out that "forcing the MT system to produce the context on the target side as well, produced better results" (section 4.3 in the paper).

7.1.2 Filters

As explored in the previous subsection 7.1.1, extracted token pairs undergo a *filtration* process. In this subsection, I will introduce all filters that I use, explain what kind of *dirty* token pairs they are supposed to catch, and also give examples of the token pairs they eliminate in practice.
It is important to note that not **all** filters are used for **every** dataset. Indeed, some datasets are dirtier than others and require the usage of more filters, in order to weed out the maximum amount of dirty token pairs.

"No mistake" filter

This filter is applied by default to all token pairs, which are extracted from a pair of sentences (as shown in subsection 7.1.1). The *"no mistake"* filter removes all token pairs, in which the erroneous and the correct token are the **same**.
This filter essentially conditions the error correction model to **always** expect an input sample to contain *one* erroneous token that needs to be fixed. This is important to keep in mind for the rest of the paper, as it also sets expectations for the performance of the error *detection* model. Indeed, always expecting an erroneous token suggests

using a detection model with high **precision** (i.e., the tokens, which the error detection models predicts to be erroneous are **indeed** erroneous) would work best.
An alternative approach would be to also train the error correction model on token pairs *without* mistakes, and having an error detection model with high **recall**. The high *recall* would mean that the detection model catches a high percentage of the **actual** erroneous tokens in a sample (this is the approach, employed in [SN20]). I will refrain from exploring this alternative approach further in the boundaries of this paper, and instead focus on testing the limits of a **"purely-erroneous"** correction models.

"Unknown GT" filter

The **"Unknown GT"** filter is specific to the ICDAR2017 dataset (introduced in chapter 9). Indeed, the authors of the dataset explain in [CDCM17] that some sequences feature OCR texts, which are **miss-aligned** with the ground truth. To indicate this, the authors decided to use the meta token '#'. The design of the dataset then leads to many token pairs, where the ground truth (i.e., the expected **output** sample) consists entirely out of '#' tokens — as shown in box 7.3. These token pairs are appropriately filtered out, as they do not carry any knowledge for the error correction model.

OCR: FLASH IS THE mum. wmm My
GT: ##########################

OCR: rockery. 30 A LADY'S TOUR IN CORSICA.
GT: ####################################

OCR: !MC z -tomize
GT: ###############

Box 7.3: Examples of *filtered-out* samples with **unknown GT**

"ICDAR hyphen" filter

Similar to the previous subsection, the **"ICDAR hyphen"** filter is indeed only used on the datasets from the ICDAR Post-OCR correction competitions — i.e., **ICDAR2017 monograph**, **ICDAR2017 periodical**, and **ICDAR2019**. The filter was implemented due to a suggestion from the authors in [CDCM17] and [RDCM19] to ignore tokens with **hyphens** when evaluating on the **test set**. It is important to note, however, that I will still use tokens with hyphens when **training** the error correction models, as I find that the expected corrections make sense.

Box 7.4 illustrates some tokens with hyphens, which are from the **training** sets of the ICDAR datasets. They are **not** filtered out by this specific filter and are considered valid samples for training. As we can see by pair #1 in the box, tokens with hyphen can have mistakes, which are completely unrelated to "hyphen correction" — eliminating them from the training dataset would result in a loss of valuable samples. Pairs #2, #3 and #4 are, on the other hand, classic examples of hyphen correction, where the hyphen was either wrongly inserted (pairs #2 and #3), or there was a problem with the surrounding indentation (pair #4).

Box 7.5, on the other hand, visualizes some hyphenated tokens, which are **filtered out** from the **test** sets. Similarly to the training tokens with hyphens from the previous paragraph, filtering out *all* tokens with hyphens in them also eliminates some **valid** post-correction samples (pair #2). However, the filtered-out samples also include samples, where the hyphen correction is completely *misleading*, often correcting a correct token to an erroneous one (**opposite** of the task in hand). This can be seen with pairs #1, #3 and #4 in box 7.5, where the OCR text contains *correct* tokens, and the hyphen correction process results in *erroneous* ground truth samples.

OCR: l-am'dlately

GT: immediately

OCR: and-Moderation,

GT: and Moderation,

OCR: de-scription,

GT: de@scription,

OCR: . - Charles

GT: .@-@Charles

Box 7.4: Examples of **training** samples with hyphens from the ICDAR datasets

OCR: title page

GT: title-page

OCR: chil-dren.

GT: chil@dren.

OCR: Albemarle street.

GT: Albemarle-street.

OCR: Ludgate-@street,

GT: Ludgate- street,

Box 7.5: Examples of *filtered-out* **test** samples with hyphens from the ICDAR datasets

Maximum length filter

The **maximum length** filter catches out tokens which exceed a custom-defined *threshold*. Although varied across the different datasets, this threshold takes default values of **15** or **20**. The length is measured in **characters**. It does not matter whether the length of the erroneous or the correct token is taken, as they both have the **same** length due to the nature of the token extraction process (refer to the start of this subsection).

The rationale behind this filter is that most valid English words are relatively **short**. Indeed, longer English words are often **compound** words, which contain a hyphen to join two individual words (e.g., *double-decker*). As such, any tokens that exceed the custom set threshold are considered to feature non-valid English words, and are thus ignored. This can be confirmed by inspecting box 7.6, which shows examples of the tokens that this filter eliminated across the different datasets.

One group of eliminated tokens through the maximum length filter are token pairs, in which big chunks of text have to be deleted, as shown in pairs #1 and #3 in box 7.6. It is important to note that some *hyphenated* tokens from the ICDAR training datasets are *also* caught by the maximum length filter, as shown in pair #2. Pairs #4 and #5 show another big group of eliminated samples by the length filter — URLs or otherwise non-natural language.

The only long samples that this particular lets through are ones that only have errors, connected to whitespaces — e.g., *"the@question@will"* → *"the question will"*, or *"car pet"* → *"car@per"*.

OCR: Overflows f t @ « ’

GT: Over@flows@@@@@@@@!!

OCR: Corsicans. - Valery,

GT: Corsicans.@-@Valery,

OCR: Independentlyo@lthes@orevaits,vouchii,if.’aerge’icallv

GT: Independentlyof these ##

OCR: DOCTORS(DOCID.NAME.SEX,SPECIALTY)

GT: DOCTORS(DOCID,NAME,SEX,SPECIALTY)

OCR: http://crl.ransu,edu/shiraz

GT: http://crl.@nmsu.edu/shiraz

Box 7.6: Examples of *filtered-out* **long** samples

“ASCII-sensitive” filter

The **“ASCII sensitive”** filter first *normalizes* the token pair. The normalization process involves transforming any **non-ASCII** characters to purely ASCII (e.g., German ’ü’ to English ’u’). For this, I use the external library **unidecode**[2].
After normalizing the characters of both tokens in the pair, it is checked whether they are **identical** — if they are, then the token pair is considered **“ASCII-sensitive”**. ASCII-sensitive token pairs then only feature character substitution(s) with respect to non-ASCII characters. Such token pairs are filtered out, as the error correction models are specialized to work with the **English** language, which does not feature mistakes of this type. Box 7.7, for example, shows some token pairs, which the filter caught across the different datasets.

[2]Homepage of the library is: `https://pypi.org/project/Unidecode/`

As expected, this filter captures cases, where valid ASCII English words have to be "corrected" to some foreign representations of the word (e.g., pairs #1 and #3). As seen from pair #2, the filter also captures cases where non-ASCII **punctuation marks** have to be substituted.

OCR: severes
GT: sévères

OCR: 27'
GT: 27′

OCR: appeared
GT: appêared

Box 7.7: Examples of *filtered-out* **ASCII-sensitive** samples

"Meaningless addition/deletion" filter

The **"meaningless addition/deletion"** filters catch **compound tokens**, parts of which have to be entirely **deleted**, or **added**. The usage of the filters is more easily seen in boxes 7.8 and 7.9. As seen there, all the eliminated samples fit one of two situations:

1. The **erroneous** token contains multiple words, some of which have to be entirely **deleted**

2. The **correct** token contains multiple words, some of which have to be entirely **added**

Token pairs like these are considered too *dirty*, as their operations are completely **unpredictable**. Technically explained, the data preprocessing scripts achieve this by calculating the *fraction* of **padding** symbols in each token. Then, if the fraction

exceeds a custom-set *threshold* (per default: **0.51**) in the **erroneous** token, it is safe to assume that some word has to be **added** to the correct version (as shown in box 7.8). Vice-versa, if the fraction exceeds the threshold in the **correct** token, then some part of the erroneous token has to be **deleted** (refer to box 7.9).

OCR: and@@@@@

GT: and when

OCR: fell@@@@@

GT: fell.....

OCR: wit@@h@@@@@@@

GT: Pet Children.

OCR: out@@@@@@

GT: out sail,

OCR: t@@

GT: the

Box 7.8: Examples of *filtered-out* **meaningless addition** samples

OCR: thus much

GT: @@@@@much

OCR: that opinion

GT: @@at@@@@@@@@

OCR: ourknowfcê*

GT: @@@@@@@@@@@

OCR: and was leaning

GT: @@@@@@@@leaning

OCR: our very

GT: ou@@@@r@

Box 7.9: Examples of *filtered-out* **meaningless deletion** samples

Mismatch filter

The last filter is called the **mismatch** filter. It calculates the Levenshtein **similarity** (refer to subsection 5.1.1) between the tokens in each pair and eliminates them if it is below the user-defined threshold. As with the other filters, the threshold was manually adapted to each dataset, so that it catches as many *dirty* samples as possible, without also eliminating valid token pairs. To this extent, the default **"match threshold"** values were set to be either **0.33** or **0.41**.

Box 7.10 shows an excerpt of the eliminated token pairs across all datasets. It is easy to see that this filter catches two main types of "dirty" samples: **mismatches** (e.g., pairs #1, #3, and #5) and **semantic corrections** (e.g., pairs #2 and #4). The latter type is interesting, as both the erroneous and correct tokens are *valid*

words. Nonetheless, semantic substitutions are *unpredictable,* the same way addition or deletion of whole tokens was in subsection 7.1.2.

OCR: '_\[' dfle
GT: Table

OCR: obligatory
GT: Begin

OCR: liliTiibcr
GT: number

OCR: purpose
GT: occurrences

OCR: f(mn(l
GT: Found

Box 7.10: Examples of *filtered-out* **mismatch** samples

7.1.3 Types of mistakes

In this subsection, I will briefly explain the different types of errors, which are encountered in Post-OCR correction. I will be using the terminology from Nguyen et al.'s paper [NJC+19], which offers a more in-depth analysis of the distribution of OCR errors and how it compares with that of *spelling correction.* I will be using the terminology later on in the paper, when I am exploring the results of the experiments.

The first type of error is called a **non-word** error. As explained back in chapter 2, a *non-word* error involves correcting an erroneous word, which is not a **valid**

English word. A trivial example of this is the pair of the erroneous token *"**c**ar"* and its correction *"**o**ar"*.

The second error type is the **real-word** error. *Real-word* errors describe token pairs, where both the erroneous and correct tokens are *valid* English words. However, the erroneous token does not fit within its surrounding **context**, and is thus considered a mistake. For an example of this, consider the two sentences *"I am driving my car."* and *"I am driving my far."* Although both *"car"* and *"far"* are correct words, the second sentence does not make any sense *semantically*.

I discussed in chapter 2 how standard *vocabulary-based* approaches to Post-OCR correction struggle with real-word errors. Indeed, using a vocabulary to check for valid words and only correcting unknown ones makes it practically impossible to correct real-word errors. Additionally, it would also be hard for a human to correct a real-word error without any *surrounding context*. Due to this, I hypothesize that both the baseline algorithm with Q-gram indices (refer to chapter 6), and sequence-to-sequence models with **isolated** correction samples, will struggle with real-word errors.

The third and fourth types of errors concern erroneous word **boundaries**. In particular, there is the **incorrect split** error, and the **run-on** error, as notated in [NJC+19].

The *incorrect split* error features token pairs, where the erroneous token is a split version of the correct token. An example of this is the erroneous token *"car pet"* and its correct alternative *"carpet"*. The keen reader might also have observed that an incorrect split might end up splitting a valid word into **two or more** valid *sub-words*, as in the example above.

The *run-on* error is the opposite of the incorrect split error. It features pairs of tokens, where the erroneous token has one or multiple missing whitespace characters when compared to the correct version. For an example, consider the mirrored version of the incorrect split example, with the erroneous token *"carpet"* and its correction *"car pet"*.

7.2 Character-level Vocabulary

The **character-level** vocabulary maps *characters* to unique **integer encodings** (as explained in section 4.1 on how to work with strings in machine learning). It contains different sets of characters, depending on whether the correction process is chosen to be **case-sensitive** or not. Indeed, the case-*insensitive* version contains only the *lowercased* characters of the English alphabet, while the case-*sensitive* version additionally includes their uppercased variants. Furthermore, the vocabulary also contains all single-digit numbers from 0 through 9, as well as all ASCII characters, which are considered **punctuation marks**[3].

Additionally, the vocabulary includes a set of **meta tokens** — [*<PAD>*, *<START>*, *<END>*, *<UNK>*, *<TGT>*]. The *<PAD>* meta-token is used to *pad* input samples to the maximum sequence length of the error correction models. This is helpful when, for example, using **batches** of input samples during training. The different sequences in the batch are not required to have the same length, which would make it impossible to cast their encoded representations into a **matrix**. This is remedied by appending *<PAD>* meta tokens after the end of every sequence with length $\leq max\_length$.

The *<START>* and *<END>* tokens are **essential** to the correction process. As explained in section 4.2 about encoder-decoder models, the autoregressive nature of the models requires a *<START>* token as the first (or **zeroth**) time step. The *<END>* token, on the other hand, allows for variable-length inputs and outputs (for more information, refer to 4.2).

The *<UNK>* token is pretty straight-forward. Indeed, it is used as a substitution for every character, which is **not known** to the character vocabulary. In theory, this meta token should be incredibly rare to spot, as the filtration process of the token pairs (look up subsection 7.1.2) normalizes all input samples to make sure they only contain **ASCII** characters. Nonetheless, it is possible for undesirable characters like the **control characters** in ASCII[4] to slip through from some Post-OCR correction

[3] All ASCII punctuation marks: !"#$%&'()*+,-./:;<=>?@[ ]^_‘{|}~

[4] I refer to the first 32 characters of ASCII as *control characters*: `https://www.asciitable.com/`

datasets. In those particular cases, the *<UNK>* token helps to bring all of these characters under one *"term"*.

The *<TGT>* meta token is used in both error *correction* and *detection* samples to mark the **target**, or **focus** tokens. In order to get a better intuition about the effect of the *<TGT>* token, imagine a *context correction* sample without one. Indeed, in such cases, the encoder-decoder network would have to learn to both **detect** where the erroneous tokens are, and propose valid **corrections** for them. Additionally, if using a sequence-to-sequence model **exclusively**, it would be infeasible to put *<TGT>* meta tokens around the erroneous tokens while training, as the same would not be possible to do in *inference*. Indeed, the *<TGT>* meta tokens are the ones that accomplish the "link" between the error detection and correction models — the error detection model first *marks* the erroneous tokens with *<TGT>* meta tokens, and then passes them on with the appropriate context to the correction model for them to be *corrected*.

7.2.1 Correction Sample Encoding

The character vocabulary, introduced in the start of this section, is used to **encode** the error correction samples, before they are fed to the sequence-to-sequence models. The process for this is simple: it first starts by splitting the error correction sample into individual **characters**. If we consider the erroneous sequence *"In <TGT>thee<TGT> paper"*, this first step results in the sequence being split into the list of characters [*'I'*, *'n'*, *' '*, *'<TGT>'*, *'t'*, *'h'*, *'e'*, *'e'*, *'<TGT>'*. *' '*, *'p'*, *'a'*, *'p'*, *'e'*, *'r'*]. Notice that meta tokens are considered by the character vocabulary to be their own **individual** tokens.

Afterwards, each character from the list is encoded with their uniquely assigned integers from the vocabulary. If we consider a **case-sensitive** vocabulary, which includes both upper- and lowercased English letters, the encoded vector of integers would look like this: [84, 63, 7, 4, 69, 57, 54, 54, 4, 7, 65, 50, 65, 54, 67]. Notice that

in this example, the integer encoding **4** represents the meta *<TGT>* token. The final steps include adding the encodings for the *<START>* and *<END>* tokens to the front and back of the vector, resulting in [**1**, 84, 63, 7, 4, 69, 57, 54, 54, 4, 7, 65, 50, 65, 54, 67, **2**]. Optionally, the user can also supply a **maximum sequence length**, which forces the vector of integers to be *padded* with the integer encoding of *<PAD>* — **0**.

It is worth noting that the data preprocessing scripts do not feature a **"case-sensitivity"** filter (refer to section 7.1.1). This is done, in order to facilitate both case-*sensitive* and case-*insensitive* error correction models. If a case-**insensitive** model is chosen, however, casting a sample *pair* (input sample — erroneous sequence, output sample — correct sequence) might fix all mistakes in the input sample. In order to keep the error correction model **purely-erroneous**, any sample pairs that become identical after casting their characters to lowercase, are discarded.
Additionally, it might happen that some sample pairs are longer than the user-supplied **maximum sequence length**. Truncating an error correction sample is *undesirable*, as it might happen to split a token down in the middle or break up the *focus token* in the middle. As such, any *"long"* samples are also dropped during the encoding process.

7.2.2 Correction Sample Decoding

The decoding process is the reverse of the encoding process from the previous subsection 7.2.1. Indeed, the character vocabulary goes through the vector of integers one by one and uses a **reverse-dictionary** to look up the corresponding character for each integer encoding. Integer-by-integer, the resulting string is built-up and returned at the end.

7.3 Data Flow

The flow of the encoded error correction data is essentially identical to the one explained in section 4.2 about how encoder-decoder networks function. Indeed, the encoder-decoder model accepts the **encoded** vector of integers, which was produced by the character vocabulary in section 7.2. Then, the encoder processes the tokens (i.e., *time steps*) one-by-one, in order to create the *context vector c*. c is then passed to the LSTM cell in the decoder module, where the output sequence is generated, again, one step at a time.

As one of the main focus points of this paper is to evaluate the role of **attention** on *character-level* Post-OCR correction, the LSTM encoder-decoder networks also feature an attention mechanism, as described in section 4.3. The *classifier* layer, which is seen on top of figure 8 for **inference**, and on top of figure 7 for training with **teacher forcing**, is combined with **softmax** to produce a normalized distribution with a probability for each of the characters from the *character* vocabulary. The process is also identical for doing Post-OCR correction with a **Transformer** model as well (refer to section 4.4).

8 Error Detection

Error detection is often done *together* in a combined step alongside error correction. This is a perfectly valid strategy, as the sequence-to-sequence models from last chapter 7 would, in theory, only need to learn to copy over all tokens that are not encased in meta $<TGT>$ tokens. On the other hand, as chapter 2 alluded to, there is also promise in regarding error detection as a completely independent step (see [SN20] and [RDCM19]). This chapter will explain how the error detection task can be done as a separate step, leveraging the **BERT** model, explained in section 4.5. The desired effect of this is then twofold: improving the overall performance on error detection and also positively influencing error correction by using error detection as a "filter" to only pass down erroneous samples to error correction models.

The chapter starts off with section 8.1, which explains how data samples are generated for the error detection task, and visualize what they look like. It also shows how the error detection samples can be used to create error correction samples, which accomplishes the link between the two steps. Then, section 8.2 explains how to mark erroneous target tokens in data samples — and showcases its influence on sample preprocessing. Furthermore, section 8.3 shows how the **BERT** model is used to predict the erroneous tokens in each sample. It also touches on what it means to *fine-tune* BERT for the error detection task, and what approaches there exist. Section 8.4 follows this with an elaboration on how the predictions of the BERT model are decoded.

Row, brothers row, the stream runs fast, The rapids are near, and the day-light's past.
I <TGT>ihou'd,<TGT> not have spoke in that manner, had I known it was <TGT>you;<TGT> and I knew you only by report.
The <TGT>seeond<TGT> line in each doublet is a new transition <TGT>chaaacteristic<TGT> of a QDM.
9 < < 3.
T. vent.
O.
<TGT>Chortts. - Ri<TGT> choo ral, on mischief that girl was bent.

Figure 16: Subset of the error detection data; red marks the target tokens

8.1 Data

An excerpt of the data, which the error detection model uses, is shown in figure 16. As displayed there, the error detection samples consist of correct sentences from the different datasets, with some of their words replaced by erroneous tokens. The erroneous tokens are *always* marked with meta $<TGT>$ tokens. This, however, does not necessarily mean that the rest of the sentence does not contain any other mistakes — some datasets also feature erroneous "correct" words. These words are not, however, regarded by the model as targets.

The creation of the error detection samples is easy and executed in parallel with the creation of error correction samples. An example of the whole process is visualized in box 8.1. In particular, a **list** is kept of all tokens pairs, which were extracted from a sentence and **not filtered** (refer to section 7.1.1). This record includes an erroneous token itself, as well as the *beginning* and *end character* indices of its corresponding token in the **erroneous** sentence. After all "valid" erroneous tokens of a sentence are marked, the list is sorted *backwards* with respect to the ending index of the marked tokens. Then, the **relevant** (i.e., **unfiltered**), erroneous tokens are marked for detection in the erroneous sentence by encasing them with $<TGT>$ meta tokens. As seen by some examples in box 8.1, not every error detection sample contains target *erroneous* tokens. Additionally, all **filtered out** tokens are essentially *ignored* in the

error detection sample. Even though they are present in the error detection sample itself, it is **not** expected from the model to mark them as erroneous targets, as they would be hard to predict from even a human.
As a final pre-processing step, it is also important to strip all meta **padding** symbols from the error detection sample — the detection sample will not have anything to align against at *inference* time.

OCR: Chortts. - Ri choo ral, on mischief that girl was bent.

GT: Chor@us.@-@Ri choo ral, on mischief that girl was bent.

Marked words: [{"start_ind": 0, "end_ind": 14, "err_token": "Chortts. - Ri"}]

Err. det. sample: <TGT>Chortts. - Ri<TGT> choo ral, on mischief that girl was bent.

Box 8.1: Creation of an error detection sample

The *<TGT>* meta tokens, which are appended to the left and right of every erroneous token, makes it easy to link the two tasks of error detection and correction together. Indeed, an error detection sample can be used to make error correction samples with or without *context*. If the error correction model is not using context, then the error detection model can pass on only the tokens, which it predicted to be erroneous. If we take the example from box 8.1, this means passing *<TGT>Chortts. - Ri<TGT>*. If the correction model, on the other hand, uses context of one word to the left and right, then *<TGT>Chortts. - Ri<TGT> choo* is passed (the additional context does not help much in this case).
This process of *passing* data from the error detection to the correction model is the essence of the two-step process, as explained in [SN20]. By first determining the erroneous tokens with a separate model, the error correction model can focus entirely on correcting the words, encased in *<TGT>* meta tokens, which has shown to boost its performance (refer to chapter 2).

8.2 Marking Mode

Consider the sentence *I love Plovdiv!* and its erroneous version, *I love Plovediv.* As explained in subsection 4.5.3, the BERT model uses a **tokenizer** to cast words to a custom *subword* level. This means that in this example, *I love Plovdiv!* becomes *['I', 'love', 'P', '##lov', '##di', '##v', '!']*, while *I love Plovediv.* becomes *['I', 'love', 'P', '##love', '##di', '##v', '.']*. The desired output from the model would then be *I love <TGT>Plovediv.<TGT>*, indicating that the tokens (plural, as punctuation marks are regarded by the BERT tokenizer as independent) between the *<TGT>* meta tokens are **erroneous**.

The previous section 8.1 explained how data samples like *I love <TGT>Plovediv.<TGT>* are created. In order for the BERT model to be able to train, however, it also needs an **expected output** vector. In the case of error detection, the output vector should indicate which words (and by extension — its subwords) are erroneous and should be marked with *<TGT>* tokens, and which ones are error-free. I will refer to this technique as the **"start w/ cont." marking mode**, and I will dedicate subsection 8.2.1 to explain how it works.

8.2.1 "Start w/ Cont." Marking mode

The **"start w/ cont."** marking mode uses the integer class **1** to mark all *correct* subwords, **2** to mark the **start** of an erroneous token, and **3** to mark any **remaining subwords**. Integer class **0** is reserved for all meta tokens, which BERT adds to a tokenized sequence (i.e., *[CLS]* and *[SEP]*), as well as all meta sequence padding tokens *<PAD>*.

If we take our previous example of *I love <TGT>Plovediv.<TGT>* again, the expected output $\boldsymbol{y}$ is shown in equation 69. The first 2 in the vector stands for the subword *'P'*, while the rest of the subwords are marked with 3s - *['##love', '##di',*

'##v', '.'].

$$\boldsymbol{y} = [1, 1, 2, 3, 3, 3, 3] \tag{69}$$

It is important to note that integer classes **2** and **3** are vital to how the expected output $\boldsymbol{y}$ is processed. Indeed, if we consider the modified vector $\bar{\boldsymbol{y}}$ in equation 70, the last class being changed from **3** to **2** translates into the error detection being *I love <TGT>Plovediv<TGT><TGT>.<TGT>* This is **undesirable** in the case of this paper, as punctuation marks are counted as *parts* of their corresponding words. As such, the "closing" <TGT> meta token is expected to be **after** the full stop.

$$\hat{\boldsymbol{y}} = [1, 1, 2, 3, 3, 3, 2] \tag{70}$$

8.3 Prediction

Figure 17 visualizes how the BERT model can be used for OCR error detection. Compared to the vanilla version of BERT (see figure 15), this error detection model has a dropout and linear layer with the softmax activation function on top of it. The linear layer at the end maps the BERT output $\boldsymbol{h}_{BERT}$ of every token to a normalized probability for the four integer classes. In the aforementioned figure 17, for example, the symbolized output vectors predict integer class **0** (misc. tokens) for the *[CLS]* and *[SEP]*, **1** (non-erroneous) for x_1 ('I'), x_2 ('love'), and **2** and **3** for the start and continuation of the erroneous token at x_3 ('P'), x_4 ('##love'), x_5 ('##di') and x_6 ('##v').

The workflow of the model is the same for **both** training and inference. The input $\boldsymbol{X} = [\text{CLS}]x_1x_2 \dots x_T[\text{SEP}]$ represents some arbitrary text sequence, which was already preprocessed by the BERT tokenizer. $\boldsymbol{X}$ is then fed to the BERT model (refer to 4.5.2 for more details), which returns a vector of latent representations $\boldsymbol{h}_{BERT}$. The layers on top of BERT then use $\boldsymbol{h}_{BERT}$ to *adapt* the model to the process OCR

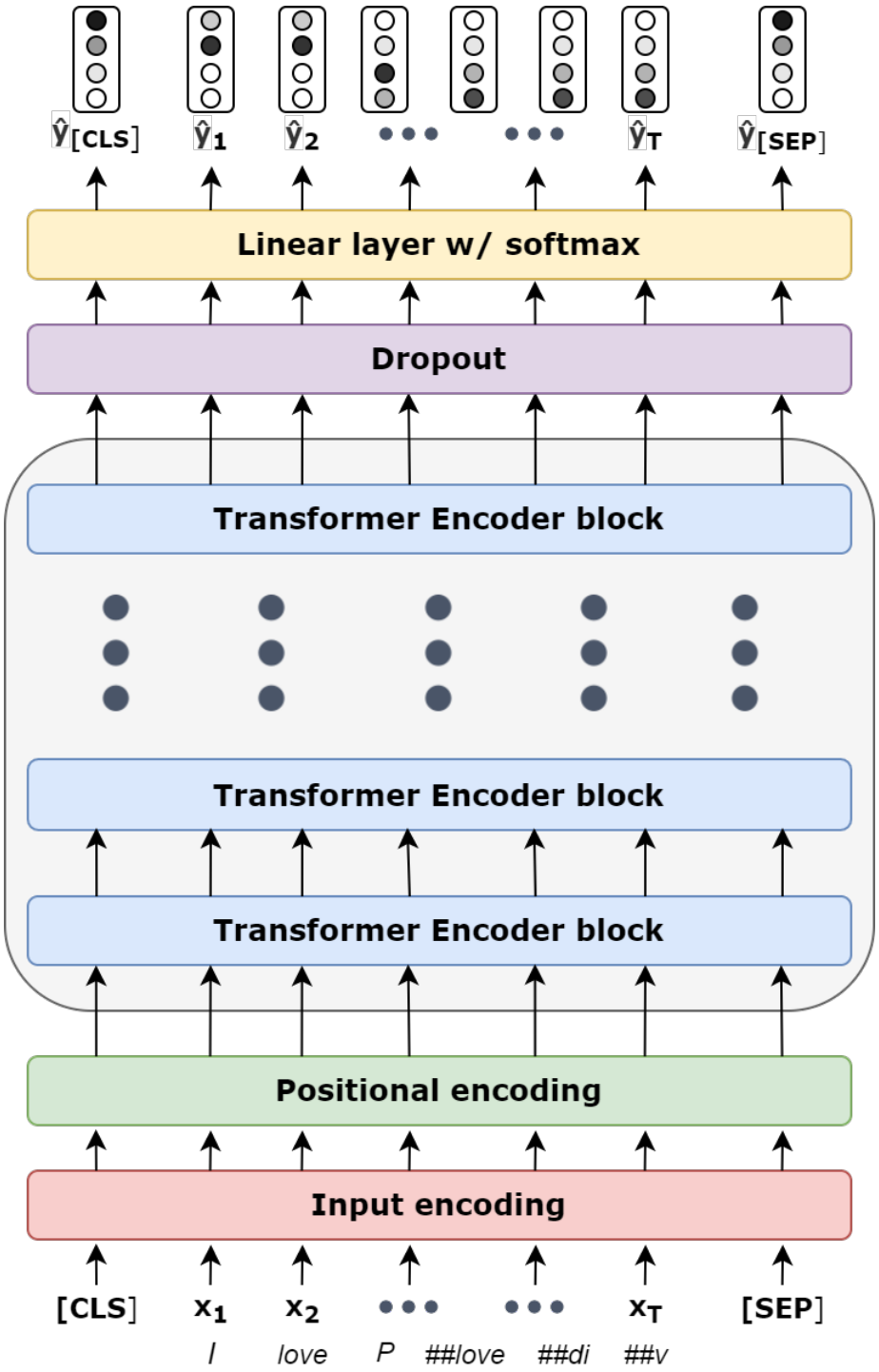


Figure 17: Using **BERT** for OCR error detection using **"start w/ cont."** marking mode; **bold** colors in the output vectors represent high probabilities

error detection. The prediction vector $\hat{\boldsymbol{y}}$, which the error detection model returns, is ultimately computed by equation 71. In the equation, W_{cls} and b_{cls} stand for the parameters of the *classifier* linear layer on the top of the model.

$$\hat{\boldsymbol{y}} = softmax(W_{cls}(dropout(\boldsymbol{h}_{BERT})) + b_{cls}) \tag{71}$$

Figure 17 also showcases how the flow of the data would look like if the erroneous sequence *I love Plovediv* is passed to the model. Above each model output $\hat{y}_{[CLS]}$, $\hat{y}_1 \dots \hat{y}_{[SEP]}$, there is a visualization of the desired probabilities, which the error detection model should ideally predict for each token. Namely, integer class **0** (i.e., *miscellaneous* tokens) is expected for the meta tokens *[CLS]* and *[SEP]*, alongside any other ones (e.g., *[PAD]*). Then, the first two tokens should be assigned a high probability of being *non-erroneous* (i.e., integer class **1**).

For the subwords of the erroneous word *Plovediv*, the expectation is that **all** subwords will be correctly marked with their expected class. Coming back to the example in figure 17, we can see that the model correctly predicts that the subword *"P"* is the **start** of an erroneous token, and the rest of the token subwords are all marked as **continuations**. If, however, one of the subword tokens was classified incorrectly, the whole *meaning* of the prediction would be changed. Consider the example where the subword '##di' was misclassified with integer class **1**, and this leads to '##v' being misclassified with *2*. Then, the decoding procedure from the following section 8.4 would end up decoding the model prediction as *I love <TGT>Plove<TGT>di<TGT>v<TGT>*, which would cause a lot of issues to the error correction model to work with.

The classification probabilities can then be plugged into a **cross-entropy** loss function, and the resulting loss value can be used to train the model. For **inference**, the classification probabilities are passed to another step in the error detection *pipeline*, discussed in the upcoming section 8.4.

8.3.1 Fine-tuning

As explained in subsection 4.5.4, the original model was *pre-trained* on **Masked-Language Modelling** and **Next Sentence Prediction**. It is then possible to *extract* word embeddings out of BERT — i.e., by averaging the embeddings for all subwords of a word — and use them in other applications, akin to how the original **word2vec** was used. This approach of using knowledge from *language representation models* is called **feature-based**. The other way of *capitalizing* on a pre-trained BERT model is called **fine-tuning**, which is what I will focus on in this subsection. Fine-tuning a model deals away with having to create *task-specific* models, and instead focuses on adjusting the pre-trained parameters of a model to a new task. This is exactly what is shown for our error detection model in figure 17. Additionally, further research in [PRS19] has shown that fine-tuning outperforms the feature-based approach in many NLP tasks, and is *never* worse in others. In the domain of *token classification* — i.e., the task, which OCR error detection falls under — fine-tuning is shown to perform only *very slightly* better. However, fine-tuning is the most widespread approach when it comes to the BERT, and thus I will also be preferring it.

There is a choice to make with fine-tuning in regard to which module of the BERT model to **freeze** during the process. *Freezing* a module stops gradients being computed for its layers, effectively *disconnecting* it from the update step while training. It is generally accepted (see [SQXH19]) that the *"lower"* Transformer blocks of a BERT model contains more **general knowledge**, which gets more *specific* further up. It would therefore be intuitively reasonable to freeze the lower layers of a BERT model and use the upper ones, combined with the classifier layer, to fine-tune to the task of OCR error detection. On the other hand, [HR18] shows that fine-tuning an **entire** pre-trained model gives best results, when the dataset of the down-stream task is big enough. I will therefore be exploring this question in the domain of OCR error detection in the experiments of this paper (refer to 10).

8.4 Decoding

As mentioned in the previous section 8.3, the error correction *pipeline* decodes the predictions of the fine-tuned BERT model and marks the erroneous tokens by the predictions for their subwords.

Let the original input sequence be $X =$ "*I love Plovediv*" and the prediction of the error detection model be $\hat{\boldsymbol{y}} = [0\ 1\ 1\ 2\ 3\ 3\ 2\ 0]$. In order to decode the model's prediction, the input sequence X is first split into its individual words by *whitespace*. Next, each word is put through the BERT **tokenizer**, in order to get its *tokens* (i.e., *subwords*). If any of the tokens of the word were predicted to be the *start* of an erroneous entity, then a $<TGT>$ meta token is appended to its **left** side. Afterwards, the loop keeps going through the subwords of the samples until one of the three integer classes [**0**, **1**, or **2**] is encountered. Each one of those classes mark the *end* of an erroneous entity, and start a *new one*. In order to mark the end of the erroneous entity, an enclosing $<TGT>$ meta token is inserted to the last token with integer class **3** (or **2**, if it was a *single-subword* erroneous entity).

If we come back to our example, let the tokenized version of the input sequence be $X_{tokenized} =$ *['[CLS]', 'I', 'love', 'P', '##love', '##di', '##v', '[SEP]']*. Comparing this to the output vector $\hat{\boldsymbol{y}}$, we can see that the single-token words *'I'* and *'love'* have not been predicted to be erroneous by the model. The word *'Plovedi'*, on the other hand, has 3 *correctly* marked erroneous subwords — namely *'P'*, *'##love'*, and *'##di'*. *'##v'*, however, was predicted to be its own single-subword erroneous entity, which would result in the decoding procedure returning "*I love* $<TGT>$*Plovedi*$<TGT><TGT>$*v*$<TGT>$".

Classification threshold

The BERT detection model initially returns numerical values for each class, where a higher numerical value represents higher confidence that the given class is correct.

The naive way to then determine the predicted classes from the model would be to just take the index of the *biggest* value. For example, consider the model returned the following dummy output for a given token in the sequence: *[25.2 -2.1 0.1 13.3]*. Clearly, the maximum numerical value of this vector is 25.2, so the predicted class is **0**. There is one obvious caveat with this technique — classes with *low* initial confidence can end up being the ones predicted because they are bigger than the values for every other class. This can lead to problems with decoding and evaluation of the prediction down-the-line, as it might result in given tokens unnecessarily being marked as erroneous.

An easy way to deal with this problem is using a **classification threshold**. With the classification threshold, erroneous classes (i.e., error start and continuation) have to be predicted with a high enough confidence, in order to be considered as such. For example, in their paper [SN20], Schaefer and Neudecker describe that "only character encodings with an error probability of >99% were treated as erroneous". The probability itself (i.e., confidence) can be gotten by applying **softmax** (refer to 3.2) to the predicted class vector. I will also be employing a classification threshold when working with the BERT detection model myself. Manual tests have revealed that not using one, and just taking the index of the biggest value, leads to a lot of superfluous target entities, which brings the overall performance of the model down.

9 Datasets

In this chapter I will introduce all datasets, with which I have worked in the boundaries of this paper. To start the chapter off, section 9.1 will present two collection of statistics, related to the error *correction* and *detection* samples from each dataset. The section acts as an overview, and should give enough information on the data, which is generated from all datasets.

Afterwards, the five successive sections will cover each one of the datasets in more details, and explain their format, sources and particularities. In particular, section 9.2 starts with the two datasets from the ICDAR2017 competitions on Post-OCR correction. Next, section 9.3 does the same for the dataset from the competition's re-run in 2019. Then, 9.4 introduces a manually curated dataset, which was created from the *ACL Anthology Reference corpus*. Furthermore, 9.5 introduces two **spelling correction** datasets. Although not usable in the context of Post-OCR correction (their underlying distributions are shown in [NJC+19] to be different), it can nonetheless be used to explore if both tasks are compatible with each other. Section 9.6 introduces another dataset, consisting *purely* of OCR errors, again on the basis of the ACL corpus.

Finally, section 9.7 finishes off the chapter by explaining how the error statistics from the OCR datasets can be used to generate an arbitrary amount of **artificial** error correction and detection data, by using the help of an external dataset of arXiv documents.

	Set type	# token pairs	# OCR/GT chars	# edit operations	CER
ICDAR2017 monograph	Training	29,044	182,332/178,231	9,524 insertions (5.22%) 13,625 deletions (7.47%) 18,714 substitutions (10.3%)	∼23.49%
	Validation	7,968	46,648/48,218	4,440 insertions (9.52%) 2,870 deletions (6.15%) 4,165 substitutions (8.93%)	∼23.8%
	Testing	12,285	67,649/68,124	2,422 insertions (3.58%) 1,947 deletions (2.88%) 12,085 substitutions (17.86%)	∼24.15%
ICDAR2017 periodical	Training	27,481	157,210/156,002	7,263 insertions (4.62%) 8,471 deletions (5.4%) 31,537 substitutions (20.1%)	∼30.3%
	Validation	2,996	18,472/18,080	696 insertions (3.77%) 1,088 deletions (5.9%) 3,299 substitutions (17.86%)	∼28.1%
	Testing	6,936	38,810/38,599	1,587 insertions (4.1%) 1,798 deletions (4.63%) 8,541 substitutions (22%)	∼30.9%
ICDAR2019	Training	3,918	24,965/24,829	1,297 insertions (5.2%) 1,433 deletions (5.74%) 4,698 substitutions (18.82%)	∼29.93%
	Validation	1,090	6,576/6,554	355 insertions (5.4%) 377 deletions (5.73%) 1,148 substitutions (17.46%)	∼28.7%
	Testing	1,284	8,003/8,014	430 insertions (5.37%) 419 deletions (5.24%) 1,531 substitutions (19.13%)	∼29.67%
"ACL benchmark"	Validation	381	2,651/2,597	89 insertions (3.36%) 142 deletions (5.36%) 433 substitutions (16.3%)	∼25.6%
	Testing	405	2,741/2,651	61 insertions (2.22%) 151 deletions (5.5%) 459 substitutions (16.75%)	∼25.3%
"Matthias artificial"	Validation	8,182	48,646/47,665	2,517 insertions (5.17%) 3,498 deletions (7.19%) 4,992 substitutions (10.26%)	∼23.1%
	Testing	7,953	46,716/45,720	2,392 insertions (5.12%) 3,388 deletions (7.25%) 4,896 substitutions (10.48%)	∼23.4%
"Matthias realistic"	Validation	3,993	16,297/16,042	972 insertions (5.96%) 1,227 deletions (7.53%) 2,670 substitutions (16.38%)	∼30.4%
	Testing	4,003	16,130/15,585	903 insertions (5.6%) 1,448 deletions (8.98%) 2,604 substitutions (16.14%)	∼31.8%
"Pure OCR Errors"	Training	68,185	501,926/465,594	7,522 insertions (1.5%) 43,837 deletions (8,73%) 86,497 substitutions (17.23%)	∼29.6%
	Validation	17,047	126,309/117,292	1,892 insertions (1.5%) 10,906 deletions (8.63%) 21,852 substitutions (17.3%)	∼29.5%
Overall	Training	128,588	866,424/824,647	25,606 insertions (3.35%) 67,366 deletions (7.72%) 141,446 substitutions (16.33%)	∼28.3%
	Validation	41,657	265,599/256,448	10,961 insertions (4.13%) 20,108 deletions (7.57%) 38,559 substitutions (14.52%)	∼27%
	Testing	32,866	180,049/178,693	7,795 insertions (4.33%) 9,151 deletions (5.08%) 30,116 substitutions (16.73%)	∼31.6%

Table 3: Collection of error correction statistics for each dataset in the paper

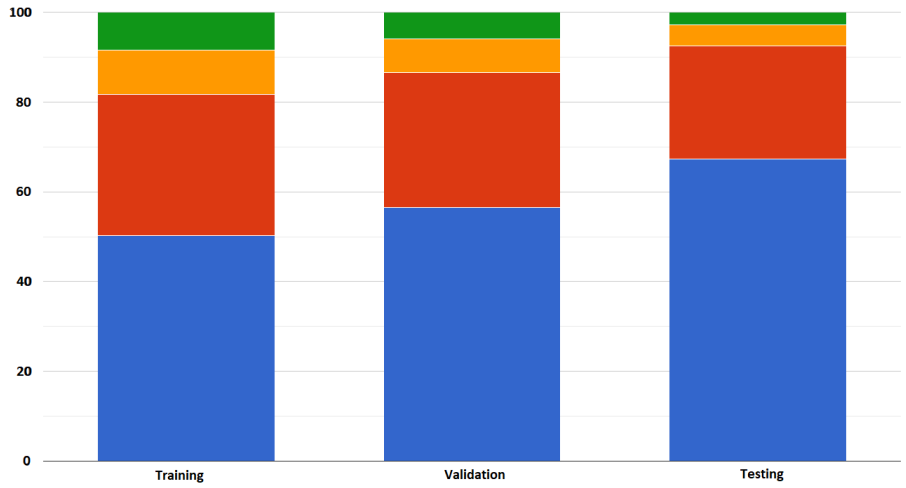


Figure 18: Distribution of error types depending on the **edit distance**; Y-axis value is % of overall number of token pairs;
blue represents **single**-mistake errors;
red represents **double**-mistake errors;
yellow represents **triple**-mistake errors;
green represents **multi**-mistake errors

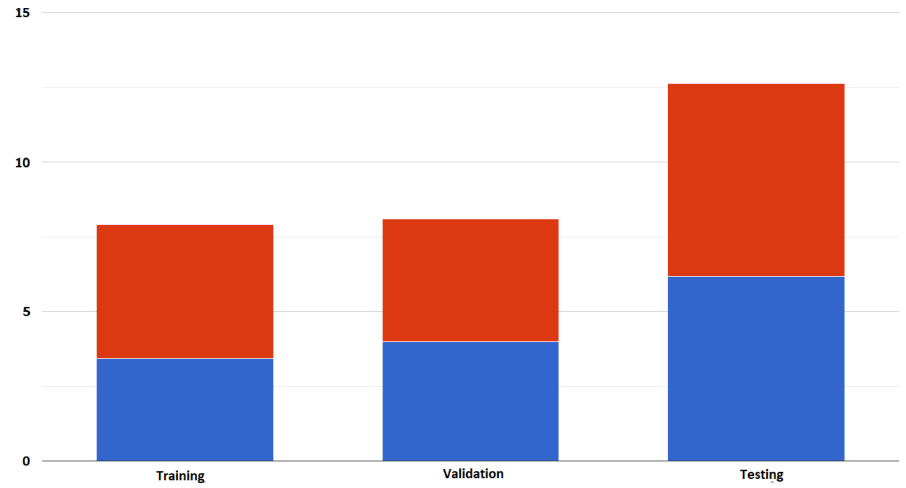


Figure 19: Distribution of **word boundary** errors; Y-axis value is % of overall number of token pairs;
blue represents **run-on** errors; red represents **incorrect split** errors

9.1 Overview

I will be using five different datasets in the boundaries of this thesis, with two of them containing two separate *sub-datasets*, which I will be treating as their own. All-in-all, this amounts to **seven** sub-datasets in total. *Five* of the seven sub-datasets contain data about Post-OCR correction mistakes. They will be used as the **core** for training and evaluating the error correction models. Those sub-datasets are (in no particular order): **ICDAR2017 monograph** and **periodical** (two sub-datasets in the over-arching ICDAR2017 dataset), **ICDAR2019**, **"ACL Benchmark"**, and **"Pure OCR errors"**. As for the remaining two sub-datasets, they are both contained in a **spelling correction** dataset I will refer to as "Matthias Benchmark". The two spelling correction datasets will **not** be used for training the error correction model, and instead only for *evaluation*. I decided to add this dataset, in order to further confirm or deny the conclusion from [NJC+19], which determines that the distribution of OCR errors is **different** from the one of spelling correction.

9.1.1 Error Correction Statistics

Table 3 offers *character-level* statistics for each of the aforementioned datasets. The information is split into five columns - **set type**, number of extracted **token pairs**, number of characters in all *input* (i.e., **OCR-ed** texts) and *output* sample (i.e., **GT** texts), number of **edit operations**, and finally — **Character Error Rate** for a particular set of samples. The second column — "# token pairs" — stands for the amount of extracted token pairs, which survived the *filtration* procedure, described in section 7.1. Inadvertently, this number corresponds with the number of *isolated* and *context* error correction samples that were generated from that particular dataset, for all datasets **except** "Pure OCR Errors". The "Pure OCR Errors" is handled differently than all other datasets, as it only contains a **list** of erroneous tokens and their respective corrections (e.g., *"correst)onds"* → *"corresponds"*). As such, there is

no **free-flowing** text, from which to build context correction and error detection samples. Section 9.7 later in this chapter discusses a technique to circumvent this issue. For the purposes of this overview, however, it is only important to know that "# token pairs" applies **only** to the **isolated** error correction samples or the "Pure OCR Errors" dataset.

The third column — "# OCR/GT chars" is self-explanatory. It offers information about how many characters there are as a whole across all input (i.e., OCR) and output (i.e, GT) samples of a dataset. This data is connected to the information in the fourth column — "# edit operations". Indeed, by looking at the difference between the number of *OCR* and *GT* characters, one can draw conclusions about the types of errors in a dataset. For example, if we look at the statistics for the **ICDAR2017 monograph** training set, there are **4,101** more characters in total across the input samples. This suggests that there would be many **deletion operations** in the dataset, in order to transform all OCR into their respective GT samples. This can be confirmed by looking at the "# edit operations" column, where it says that 7.47% (or, 13,625) of all OCR characters are *deleted*. The difference between the two deletion values — 13,625 from the fourth column, and 4,101 from the third — can be explained by the presence of **insertion** operations. Indeed, if we subtract the number of insertion operations from the deletion ones, we arrive at the original value of 4,101 characters ($13,625 - 9,524 = 3,770$).

The last column in the table stands for the **Character Error Rate** (or **CER**) of the dataset. As explained in section 7.3, this metric provides an idea of how just how *erroneous* a given dataset is. Coming back to the example about the **ICDAR2017 monograph** dataset, the character error rate of approximately **23.49%** means that one out of every **four** characters is an OCR mistake.

To supplement the general statistics in table 3, I also present the distributions of different error **types** in figures 18 and 19. In particular, the former visualizes the distribution of errors based on their **edit distance**, while the latter focuses on visualizing the fraction of **word boundary** errors (i.e., *run-on* and *incorrect split*

errors). The values in the graphs are grouped and visualized by the *set type*, i.e., whether the data belongs to one of the *training*, *validation* or *testing* sets.

Figure 18 shows clearly that the **majority** of errors only have a *single* mistake (i.e., the token pair has an edit distance of **1**). If the edit distance is increased, the percentage of overall tokens respectively drops. The green bar, which represents **multi**-mistake errors (i.e., **more than three**), makes up the smallest parts of errors in each dataset. Note that the fractions of error types sum up to 100% — the overall number of token pairs for each set type. This observation is also shared in section 4.1 of [NJC+19], in which Nguyen et al. conclude the same relationship. The code behind the paper is, however, not available, and thus I can not guarantee that both papers arrived to the exactly same values. Still, the ratios between the different error types is the same between the two papers, and we can thus safely assume that the observation holds true.

The first thing that may draw the reader's attention in figure 19 is the Y-axis values. Indeed, the percentage of word boundary errors is *low* throughout all three different set types, with an overall **8.34%** for the training set, **8.26%** for the validation set, and **12.22%** for the testing set. Among the *types* of word boundary errors themselves, **incorrect split** mistakes occur slightly more often. The ratios for the different sets are **3.55%** run-on to **4.79%** incorrect split error in the training set, **4.06%** to **4.2%** in the validation set, and **6.1%** to **6.12%** in the testing set. Nguyen et al. covers word boundary errors in section 4.5 o [NJC+19]. As with the error types based on edit distance, I can not provide a guarantee that the exact values between the two papers match, due to the lack of published code. However, Nguyen et al. also observe that *incorrect split* errors are more often observed across all datasets they test out. Additionally, they find that the fraction of word boundary errors itself is *low* compared to the sizes of the respective datasets — generally laying in the range of **15%- 22%**. The reported values in figure 19 are lower than that, laying in the range of **8%-13%**. This can be, however, explained by the addition of the *"Pure OCR Errors"* dataset, as the dataset has a **tiny** fraction of word boundary errors

	Set type	# samples	# overall tokens	# focus tokens	# focus entities
ICDAR2017 monograph	Training	22,868	806,627	57,785 (7.16%)	29,004 (∼2 tokens per entity)
	Validation	4,175	129,048	14,187 (11%)	7,968 (∼1.8 tokens per entity)
	Testing	4,243	206,482	17,260 (8.36%)	12,285 (∼1.4 tokens per entity)
ICDAR2017 periodical	Training	10,328	388,974	44,641 (11.48%)	27,481 (∼1.6 tokens per entity)
	Validation	1,391	48,035	5,171 (10.77%)	2,996 (∼1.7 tokens per entity)
	Testing	3,796	115,495	10,646 (9.2%)	6,936 (∼1.5 tokens per entity)
ICDAR2019	Training	1,506	47,026	6,899 (14.67%)	3,918 (∼1.8 tokens per entity)
	Validation	362	12,047	1,809 (15%)	1,090 (∼1.7 tokens per entity)
	Testing	648	16,796	2,142 (12.75%)	1,284 (∼1.7 tokens per entity)
"ACL benchmark"	Validation	1,100	25,288	882 (3.49%)	381 (∼2.3 tokens per entity)
	Testing	952	21,637	1,027 (4.75%)	405 (∼2.5 tokens per entity)
"Matthias artificial"	Validation	2,373	73,020	10,744 (14.71%)	8,182 (∼1.3 tokens per entity)
	Testing	2,373	70,636	10,367 (14.68%)	7,953 (∼1.3 tokens per entity)
"Matthias realistic"	Validation	2,357	64,047	4,289 (6.7%)	3,993 (∼1.1 tokens per entity)
	Testing	2,344	61,601	4,296 (7%)	4,003 (∼1.1 tokens per entity)
Overall	Training	34,702	1,242,627	109,325 (9.28%)	60,403 (∼1.8 tokens per entity)
	Validation	11,758	351,485	37,082 (10.55%)	24,610 (∼1.5 tokens per entity)
	Testing	14,356	492,647	45,738 (9.28%)	32,866 (∼1.4 tokens per entity)

Table 4: Collection of error detection statistics for each dataset in the paper; the **token** statistics are made on the basis of the "BERT-Cased" tokenizer

across its training and validation sets (i.e., **sub 1%**).

9.1.2 Error Detection Statistics

Table 4 provides information about the error **detection** samples, which were extracted from **six** of the seven *sub-datasets*. As explained in the previous subsection 9.1.1, the **"Pure OCR Errors"** dataset does not allow the creation of error detection samples by itself. Section 9.7 further in the chapter explains how this can be achieved by

using a *clean* external dataset, but the amount of generated context correction and error detection samples is *dynamic* and dependent on **user-set** thresholds.

Table 4 is split into five columns - **set type**, **number of error detection samples**, **number of *tokens* across all samples**, **number of *focus tokens*** overall, and finally — **number of *focus entities***. The first column is self-explanatory — it describes how many error detection samples there are in a particular dataset (the process of error detection sample generation is described in section 8.1.
The "# overall tokens" column gives information about the number of overall **tokens** there are across all error detection samples in a dataset. As described in the caption of the table, the tokens for the evaluation were generated by a **"BERT-Cased"** tokenizer. Although using a **"BERT-Uncased"** tokenizer changes the marginal values in the tables, the important percentages (i.e., percent of focus tokens and entities) stays the same.
The third column — "# focus tokens" — provides data about the percentage of all **target**, or **focus** tokens (i.e, subwords) that have to be caught by the error detection model. In particular, all tokens of words, which are encased by meta $<TGT>$ tokens in the error detection sample, are considered to be *focus tokens*. For example, if we return to the recurring example in chapter 8 of *"I love $<TGT>$Plov**e**div.$<TGT>$"*, the tokens *'I'* and *'love'* are *not* considered focus tokens, while [*'P'*, *'##love'*, *'##di'*, *'##v'*] are.
Sticking to the example above, the whole token *"$<TGT>$Plov**e**div.$<TGT>$"* is considered to be **one** token entity. The last column — "# focus entities" — then gives information about how many such entities there are across all detection samples of a dataset. This data is interesting, as it can be used as an **auxiliary** metric about how *dirty* the target tokens are. Indeed, if we look at the statistics for the two *spelling correction* sub-datasets — "Matthias artificial" and "Matthias realistic" — the *token per entity ratio* is low (approximately 1.3 and 1.1 respectively). This means that the tokens in the focus entities themselves generally consist of a **single subword**, which is only the case for short and common English words (e.g., *'eye'*, *'might'*, *'dye'*). In

comparison to that, the statistics for the **ICDAR2017 monograph** dataset say that one focus entity contains *two tokens* on average. This indicates that the errors in the Post-OCR datasets result in more *unusual* words, which have to be tokenized into multiple subwords by the **"BERT-Cased"** tokenizer, in order to be supported.

9.2 ICDAR2017 Datasets

The International Conference on Document Analysis and Recognition (ICDAR) hosted a competition on Post-OCR correction in 2017 ([CDCM17]). It required submitting an approach (not necessarily a deep learning model) for **detecting** and **correcting** OCR errors. The organizers provided a dataset of their own, based on OCR-ed texts from the National Library of France and the British Library and ground truth texts from multiple sources, including Gutenberg, Wikisource and Europeana Newspapers. The dataset contains documents in two languages — English and French. For the purposes of this paper, we will focus entirely on the former.
The English documents are split into two categories: **monograph** (i.e., books) and **periodical** (i.e., newspapers). As seen by the evaluation results in [CDCM17], the different document types are treated as two separate datasets — an approach that I will also adopt for this paper. Each document is stored in a text file with three lines: the first one features the OCR-ed text sequence, the second one contains an aligned version of it and the third contains an aligned version of the correct, ground truth sequence. The aligned sequences are the ones that are passed along for preprocessing, as the extraction of erroneous tokens requires both sequences to be aligned (refer to section 7.1.1 for more information on how error correction samples are generated).

There are 6 filters applied to the extracted erroneous token pairs from the two ICDAR2017 datasets (a full overview of all filters is offered in subsection 7.1.2). One of the filters — the *"ICDAR hyphen"* filter — was only used with the **test** datasets to ignore all hyphenated tokens (as per suggestion of the organizers themselves in

[CDCM17]). For the ICDAR2017 *monograph* dataset, 85% (36,972) of all 43,324 **training** extracted token pairs, as well as 83% (12,285) of all 14,832 **testing** ones, passed the filtration process. The distribution of the filtered-out samples, on the other hand, looks like this:

- Unknown ground truth filter: 7% (3105) training, 3% (399) testing
- Maximum length threshold of 15 characters: 1% (549) training, 1% (109) testing
- "Unicode-sensitive" filter: 2% (708) training, 0% (8) testing
- "Meaningless" add/del w/ threshold 0.51: 3% (1,089) training, 0% (64) testing
- Similarity threshold of 0.33: 2% (900) training, 1% (99) testing
- Hyphenated token filter: 13% (1859) testing

For the ICDAR2017 *periodical* dataset, 83% (30,477) of all 36,765 **training** extracted token pairs, as well as 72% (6,936) of all 9,575 **testing** ones, passed the filtration process. The distribution of the filtered-out samples itself is shown immediately below:

- Unknown ground truth filter: 11% (3917) training, 11% (1006) testing
- Maximum length threshold of 15 characters: 2% (737) training, 1% (58) testing
- "Unicode-sensitive" filter: 1% (440) training, 0% (8) testing
- "Meaningless" add/del w/ threshold 0.51: 1% (438) training, 1% (146) testing
- Similarity threshold of 0.33: 2% (755) raining, 2% (189) testing
- Hyphenated token filter: 13% (1232) testing

The first **three** rows in tables 3 and 4 offer data about the error *correction* and *detection* samples, generated from the **monograph** document collection of the ICDAR2017 dataset. As seen by the CER values in the error correction table, the monograph document collection contains the **least dirty** OCR samples out of all OCR correction datasets (the spelling correction datasets do *not count*). What is more to see is that the distribution of edit operations differs between the three different sets. In particular, the percentage of *substitution* operations in the testing set has increased by **40%**, compared to that in the training set, while the percentages of *insertion* and *deletion* operations have sunk by **24%** and **56%** respectively.
As far as the error detection data goes, the ICDAR2017 monograph dataset yields one of the lower ratios of **focus** to *regular* tokens. However, it also holds one of the highest *tokens per entity ratio*, coming in at almost **2** tokens per focus entity. These two observations suggest that the monograph dataset does not contain *many* mistakes, but the ones it does are difficult to correct.

Rows *3* through *6* of the overview tables 3 and 4 contain information about the **periodical** set of documents from the ICDAR2017 dataset. Compared to the monograph documents, the periodical dataset have a much higher CER value, scoring **30%** percent across all three set types. A **30%** Character Error Rate means that roughly every **third** character from the dataset is erroneous. This corresponds with the comments from the creators of the dataset in [CDCM17], where they assert that the OCR error rate is higher for periodical documents than monograph ones. The authors argue that this stems from the **dated origins** of the documents in the dataset. According to [CDCM17], the majority of the OCR-ed documents predate the 19th century. This is a two-fold problem, as on one hand the age of the documents naturally leads to quality degradation and dirtier OCR results. On the other, [AC18] shows that the dataset also features correct words in an archaic form of the English language, e.g., *compleated*.
The error detection statistics help complete the profile for the ICDAR2017 periodical dataset. As confirmed by the values in column "# focus tokens", the periodical dataset

has more overall tokens that need to be corrected. Compared to the monograph collection, however, the ratio of tokens per entity is lower, sitting at around **1.6** tokens per entity. This suggests that the mistakes from the ICDAR2017 periodical dataset should be *easier* to correct, although their number is *higher*.

9.3 ICDAR2019 Dataset

The last time a Post-OCR competition was held on the International Conference on Document Analysis and Recognition (ICDAR) was in 2019 ([RDCM19]). Similar to 2017 (refer to section 9.2), the objectives were error detection and correction and a collection of documents was provided for training and evaluation of the proposed solutions. The dataset features documents across 10 different languages in the same format of unaligned/aligned OCR-ed text and an aligned gold standard.
Compared to ICDAR2017, the data available for the English language is limited. On top of that, the OCR quality is even dirtier, as [RDCM19] claims that the error rate for some historical books hits 50% (once every two characters!). Additionally, the aligned OCR-ed and ground truth text for some English documents are shuffled around, and their texts do *not* overlap. I searched and manually flagged all examples of this I can find, and will therefore be skipping 14 documents from the training and 10 from the testing collection[1].

A subset of the filters from section 9.2 are applied to preprocess the ICDAR2019 extracted token pairs. At the end, 72% (5,008) of all 6,949 extracted training and 70% (1,284) of 1,826 testing token pairs remain usable. The distribution of the filters is described in the list below:

- Maximum length threshold of 15 characters: 7% (453) training, 3% (60) testing

[1] In particular, these are documents [3, 5, 16, 23, 27, 29, 63, 64, 85, 88, 119, 125, 127, 145] from the *training* English dataset, and [4, 9, 11, 13, 22, 27, 33, 35, 37, 39] from the *testing* English dataset

- "Unicode-sensitive" filter: 2% (150) training, 0% (4) testing
- "Meaningless" add/del w/ threshold 0.51: 5% (399) of training, 6% (106) testing
- Similarity threshold of 0.33: 14% (939) training, 13% (230) testing
- Hyphenated token filter: 8% (142) testing

The **correction** and **detection** sample statistics for the ICDAR2019 dataset are given in rows *7* through *9* in the overview tables (3 and 4). The ICDAR2019 holds CER values similar to the ICDAR2017 periodical dataset, which hints at it being a *dirty* dataset as well. Indeed, a quick manual observation of the dataset revealed that *old versions* of words are often found, alongside a high percentage of **named entities** (e.g., Pamphilus, Boro., S.E.). Additionally, it seems that in some comes the gold standard even makes correct words wrong (e.g., occasion → occaffon), which makes working with this dataset even harder. The detection statistic further consolidates the bad quality of this dataset, as it contains the **highest** percentage of *focus words* out of all OCR datasets.
The last thing worth mentioning about the ICDAR2019 dataset is its *size*. Indeed, compared to the English datasets from ICDAR2017, the one from 2019 is **tiny** (3,742 extracted tokens from ICDAR2019 versus 28,055 extracted tokens from ICDAR2017).

9.4 "ACL Benchmark" Dataset

The dataset was created "in-house" of the Chair of Algorithms and Data Structures and given to me to use as a starting point for my work. The dataset was manually created by taking OCR-ed texts from the ACL Anthology Reference Corpus [BDD+08] and correcting them. Because of the manual labor, it has very little samples in two sets — one for validation and another one for testing, each with only 500 samples.

The data is provided in files with free-flowing text sequences. As the sequences were manually corrected, which also eliminated any whitespace errors, the erroneous and correct versions are **not** aligned. Because of this, the alignment procedure from subsection 5.3 is used to first align the corresponding pairs of sequences. Then, the extracted token pairs are put through the filters in the list below. As expected, the manually-curated nature of the dataset stops the filters from removing many samples. Nonetheless, some extracted token pairs feature either extremely long sequence with tiny corrections (e.g., *"'John-g**l**ves-a('certa**l**n')-book-to-everybody'"* and *"'John-g**i**ves-a('certa**i**n')-book-to-everybody'"*), or samples that are too dirty for even a human to correct (e.g., *"7 n/,"* and *"Ti^m,"*).

1. Maximum length threshold of 20 characters: 2% (9) validation, 2% (9) testing

2. "Unicode-sensitive" filter: 0% validation, 0% of testing

3. "Meaningless" add/del w/ threshold 0.51: 1% (6) validation, 5% (19) testing

4. Similarity threshold of 0.33: 4% (18) validation, 3% (14) testing

Character-statistics about the leftover 381 (92%) validation and 405 (91%) testing token pairs are available in rows *10* through *11* in table 3. The dataset may hold a character error rate on the lower end, but its main complexity comes from the technical jargon, used in it. A lot of token pairs involve fixing one-symbol characters in formulas (e.g., *"?"* to α or *"c(t)"* to *"c(i)"*). More than that, the dataset also contains sequences and words in German, which I intentionally leave in, in order to check how an English model would deal with them.

This profile of the dataset can also be confirmed by looking at the error detection statistics in table 4. The percent of *focus token* is low (due to the manually-curated nature), but the **token per entity ratio** is the **highest** out of all OCR datasets. One explanation of this fact might be that the **BERT tokenizer** treats punctuation

marks as their **own** characters, and formulas and technical jargon use many of those (brackets, equal signs, etc.).

9.5 "Matthias Benchmark" Dataset

The "Matthias Benchmark" datasets were used in the master's thesis of my supervisor [Her19], in order to evaluate neural language models on the task of spelling correction. They contain sequences (can be more than 1 sentence) from Wikipedia documents in their correct and erroneous versions. The author used the two datasets as a benchmark and used two different approaches to create them:

- "Artificial" dataset: each word has a 20% chance to be affected by noise. If chosen, there is a 80% chance of inserting a single error and 20% chance of inserting two out of the following:
  - 80% - random edit operation is chosen uniformly (insert/delete/replace character or swap two neighboring ones)
  - 10% - a whitespace is inserted at a random position
  - 10% - the whitespace after the word is removed
- "Realistic" dataset: each word has a 20% chance to be affected by noise. If chosen, the number of typos is also randomly determined by a custom formula, which assigns an exponentially smaller probability of having more than 1 error. A typo collection ([Nor09]) from the Birkbeck spelling error corpus ([Mit80]) is then used to check for realistic misspellings of the sampled word with the exact amount of errors. If there is not such a misspelling, then no noise is inserted.

In the boundaries of this paper, I treat the two datasets as independent of one another and exclusively for validation and testing. This is done as an experiment to see if knowledge between the areas of spelling correction and Post-OCR is transferable. The "corrupt" and correct pairs of text sequences are aligned, before the affected token pairs are extracted from them. Following the preprocessing procedure from subsection 7.1.1, the following percent of pairs are filtered out (the first pair of percents represent the percent of filtered out validation and test samples from the artificial dataset, and the second —— from the realistic one):

1. Maximum length threshold of 20 characters: 0/0% and 0/0%

2. "Unicode-sensitive" filter: 0/0% and 0/0%

3. Padding fraction threshold of 0.51: 0/0% and 2/1%

4. Similarity threshold of 0.33: 0/0% and 1/1%

Rows *12* through *15* of the overview tables 3 and 4 offer statistics about the **correction** and **detection** sample, produced from the two *sub-datasets.* All-in-all, the "artificial" dataset produces 8,221 token pairs for validation and 7,953 for testing (50/50 split). Similarly, the "realistic" one leaves us with 3,993 pairs for validation and for testing — 4,003.

There are multiple conclusions that can be drawn from the error correction table. For one, the artificial dataset has a lower character error rate than the realistic one. This might come as surprising at first, but can be explained by the fact that the realistic dataset uses the "Peter Norvig" typo dataset (refer to [Nor09]) for its corruption of words. The "Peter Norvig" dataset, however, is very *dirty*, often featuring entirely wrong misspelling of correct words (e.g., *"straight"* to *"strate"*). Second, the distribution of edit operations seems almost uniform in the case of the artificial dataset, with a slight edge for deletions. This corresponds well with the method that was used to generate the dataset, as explained in the beginning of this subsection. The error

detection statistics do not offer any further surprising information, but just re-affirms the two observations from above.

9.6 "Pure OCR errors" Dataset

The "Pure OCR Errors" dataset is a collection of 110,000 pairs of erroneous OCR tokens and their error-free versions, combined with a weight to indicate how often that particular OCR-ed token was found:

$$\text{adwmced} \rightarrow \text{advanced} \rightarrow 4$$
$$\text{correst)onds} \rightarrow \text{corresponds} \rightarrow 4$$
$$\text{exe(:uLing} \rightarrow \text{executing} \rightarrow 1$$

Similar to the "ACL benchmark" dataset (refer to section 9.4), this one was created in-house at the Chair of Algorithms and Data Structures In comparison to the manual nature of the former, however, the "Pure OCR Errors" dataset was automatically created by comparing documents from the ACL Anthology Reference Corpus [BDD+08], which contain OCR mistakes from scanning, with a cleaned-up version of the same corpus. The paper for the latter is sadly not available any more under its former URL [2].

The "Pure OCR Errors" dataset is used exclusively for training and validation. It is one of the three main sources, which give insight into the distribution of OCR errors, together with both ICDAR datasets (see section 9.2 and section 9.3).
The OCR-ed tokens of the dataset are very dirty. In order to prepare the best quality data for the training of the error correction models, the preprocessing procedure below is executed. As the dataset is different from the **free-text** datasets from the previous sections, it uses a somewhat different preprocessing procedure:

[2] `https://web.eecs.umich.edu/~lahiri/acl_arc.html`

1. Load the token pair and do the following checks:

    a) If the mistake in the input (OCR) pertains to a missing apostrophe (e.g., shouldn?t → shouldn't), fix the correct version to properly reflect the change (often, the apostrophe is missing); this step fixes the apostrophe of **721** token pairs

    b) If mistake is correcting a compound word (e.g., carpet → car pet) it is often the case that the correct version contains a question mark instead of whitespace (i.e., car?pet); this step fixes **534** such cases

2. Remove any ASCII control characters from the input token; if it becomes blank, **skip** (removes 86 pairs, close to 0.1%)

3. If the token pair has a weight of more than 2, append to result directly; if not, continue by:

    - Checking if the input is longer than one character; as a rule, one-character inputs do **not** contain enough context to be able to deduce their correct version (removes 1603 token pairs, or 1.5%)
        - *"o"* to *"n"*
        - *"B"* to *"fi"*
        - *" "* to *"hy"*
    - Checking if the **correct** token is a *valid* English word with the external library **PyEnchant**[3]; this check is only done for tokens, which start with a *lowercase* letter, as ones with an *uppercase* start are considered to be **named entities** (removes 13,215 pairs, or 12.1%)

[3]Homepage of the library is: `https://pyenchant.github.io/pyenchant/`

  - *"mure(Is"* to *"mands"*

  - *"occu.rren.ee"* to *"occturence"*

  - *"overdeterminetion"* to *"overdetermination"*
    (false positive — unknown to PyEnchant)

- Checking the similarity between both tokens and skipping if it is lower than the user-defined threshold (per default: 0.5); the similarity is equal to the Levenshtein distance, divided by the length of the longer string (removes 9201 pairs, or 8.41%)

  - *"(:()rims"* to *"corpus"*

  - *"(:onsulte(1"* to *"Equivalencies"*

  - *"analysis"* to *"gemination"*

An overview of character-level statistics about the "Pure OCR Errors" dataset is provided in rows 16 and 17 in the overview table 3. All-in-all, 85,232 (78%) of all token pairs pass the filtration procedure, out of the original list of 109,337 pairs. The leftover clean pairs are split into an 80% training and 20% validation sets. Overall, the dataset has an almost 30% character error rate, which puts it at the same level as the ICDAR2019 dataset (see section 9.3). Additionally, the high ratio of substitution and deletion edit operations, compared to insertions, can be explained by the many double-character substitutions in the dataset (*"|)"* to *"D"*, *"in"* to *"m"*, etc.).

9.7 Artificial Sample Generation

As hinted at in the overview section 9.1, the **"Pure OCR Errors"** dataset does not support the creation of *context correction* or *detection* samples, as it is purely a list

of erroneous tokens and their corrections. However, the correction overview table 3 shows us that the "Pure OCR Errors" dataset is the **biggest** dataset by far, yielding more token pairs than every other OCR dataset **combined**. It is then essential that a technique is developed, with which the OCR errors from the dataset can also be used to generate *context correction* and *detection* samples, so that this information is not lost to the models that train on those types of data.
This section proposes a solution to this problem by using an **external** document dataset, made out of arXiv papers, in order to artificially generate context correction and detection samples. To that extent, subsection 9.7.1 will introduce the arXiv document collection, and explain the **text contents** from its papers can be leveraged for the artificial generation of samples. Then, subsection 9.7.2 goes on to describe a way to extend this technique, such that an arbitrary amount of new samples can be generated by using **error statistics** from other Post-OCR correction datasets (i.e., the ICDAR datasets and the "Pure OCR Errors" dataset).

9.7.1 arXiv Document Dataset

The arXiv dataset is a large collection of text files, provided to me for usage by the Chair of Algorithms and Data Structures. It was originally created in a work by Bast and Korzen [BK17] to fill in the gap of a benchmark for PDF text extraction. The tool for creating the dataset was afterwards deployed to produce an even bigger collection to be used for a sub-task of spelling correction in [BHM20]. The structure of the dataset is the following: it is split into folders, which group the documents by year and month of publication. Each file then corresponds to an arXiv paper with the ID "<yy><mm>.<4-digit identifier>".
The files themselves contains all free text from the corresponding paper, not including captions of figures and tables. Formulas are also "redacted", while citations and references are encased in square brackets and use the underlying LaTeX IDs. The start of each new "block", i.e., chapter, section, paragraph, etc., inserts a new line

and the text of the block is all contained in it.

When loading documents from the dataset, I ignore [formula] blocks entirely. Other LaTeX commands, however, like references ([\ref=...]) and citations ([\cite=...]), are often integrated in the text and their removal can disrupt the logical continuation of the sentence. Because of this, they are generically replaced with the name of *LaTeX command* itself (e.g., ([\ref=...]) becomes ([ref]). Additionally, I use the external library **unidecode**[4] to *ASCII-normalize* each line. I then skip lines with less than **four** words, as manual testing has revealed those are usually the names of the different chapters, sections and so on.

The rest of the file — which is the actual textual content of the paper — is loaded and split into sentences. The procedure in listing 9.1 shows how they are then used to produce samples with additional context. The algorithm requires two custom-set variables δ and t, as shown in line 4. δ is a threshold on how many new context correction samples have to be created in the "time span" of the last t yielded documents. This mechanism allows more fine-grained control on how the generation loop on line 10 should be.
The loop itself starts with two checks — one, which determines if there are any more ground truth tokens left to generate context samples with (line 12) and the other one being the delta-threshold explained above (lines 14-16). The first condition exists, because correct tokens and their erroneous versions are deleted every time they are matched to words from the arXiv documents. This prevents unending looping through the arXiv documents due to common words being misspelled.
In order to better explain this, consider just having one token pair from, e.g., the "Pure OCR Errors" dataset — the erroneous token *"fihe"* and its correction *"the"*. Then, the procedure from listing 9.1 would go through all arXiv documents, likely very often finding the word *"the"* in its sentence. This would result in having a context correction dataset, where each sample only features correction of erroneous *"fihe"*

[4] available under `https://pypi.org/project/Unidecode/`

token. Considering the full "Pure OCR Errors" dataset again, this is undesirable due to two factors: For one, *an unending cycle* through the arXiv documents, as it is unlikely that the technical papers from the dataset would have a corresponding correct token for **every** token pair from the "Pure OCR Errors" dataset. Secondly, it would *skew* the distribution of errors from the original dataset to heavily favor token pairs, which have a commonly-used correct token.

Getting back to the procedure in listing 9.1, an inner-loop then goes through the sentences of every arXiv document and splits them into their individual words (lines 18-19). Afterwards, the program tries to match each word to **ground truth** tokens from the "Pure OCR Errors" dataset, as shown in lines 23-24. Every time a word is matched, a new context error correction sample is built by using the words to the left and right in the arXiv sentence. This does not exclude cases where the matched word is the first or last in the sentence and thus misses a part of its context. As hinted to above, the correct token and its erroneous versions are removed from their corresponding lists to avoid unending loops (lines 30-31).
If any words from a given sentence were "matched", an error detection sample is also generated from it. This is done by substituting the correct tokens with their erroneous versions and encasing them in <**TGT**> meta tokens, ensuring consistency with the samples from all other datasets.

Control over fraction of target entities

There is one downfall with the described procedure, and it relates to error **detection** samples. Indeed, marking every matched erroneous word in a sentence and then turning it into an error detection sample leads to having samples with almost **all** erroneous tokens. Indeed, consider working with the collection of *common* token pairs [(*"fihe"* and *"the"*), (*"amd"* and *"and"*), (*"ln"* and *"In"*), and (*"inorning"* and *"morning"*)]. Then, consider the example sentence *"In the morning and evening I eat cookies!"*. The procedure from above would match the first *four* words with the whole

Listing 9.1: Pseudo-code for generating context samples from arXiv documents

```
Let isolated_correction_samples be all isolated correction
    samples of a dependent dataset
Let arxiv_generator be a generator, which pre-processes and
    yields the sentences of arXiv documents

Let δ be a threshold for new samples, which have had to be
    generated from last t documents

Split isolated_correction_samples into ocr_samples and
    gt_samples

delta_num_new_samples = 0

for doc_ind, doc_sentences in enumerate(arxiv_generator):
  # Break if no more GT tokens to generate samples with.
  if len(gt_samples) == 0: break
  # Break if too few context samples generated recently.
  if doc_ind != 0 and doc_ind % t == 0:
    if delta_num_new_samples < δ: break
    else: delta_num_new_samples = 0

  for sentence in doc_sentences:
    sentence_words = sentence.split(' ')
    num_words = len(sentence_words)
    for i in range(num_words):
      # Check if the word from the arXiv document came up as
          a ground truth token.
      word_gt_ind = gt_samples.find(sentence_words[i])
      if word_gt_ind is None: continue

      Create new context correction sample by using sentence
          words to left and right
      delta_num_new_samples += 1

      # Remove GT token and erroneous version from lists.
      ocr_samples.pop(word_gt_ind)
      gt_samples.pop(word_gt_ind)

    If any words from sentence were matched, create error
       detection sample with their erroneous substitutes
```

collection of token pairs, resulting in the error detection sample *"<TGT>ln<TGT> <TGT>fihe<TGT> <TGT>inorning<TGT> <TGT>amd<TGT> evening I eat cookies!"*. This trivial example is made even worse with the full "Pure OCR Errors" dataset, and can lead to error detection samples, filled **entirely** with focus entities. I propose two ways to deal with this problem: the first is by setting a **threshold** on the amount of words from a sentence that can be turned to focus entities. The second way, which is what I will be using in the boundaries of this paper, is using information from **other** error detection datasets — namely, the three ICDAR datasets. In particular, pre-calculated statistics about the **fraction** of *focus tokens* based on token **length** are used. For example, let the average fraction of erroneous tokens in an ICDAR2017 monograph sample with token length **35** is **0.071**). This means that, in average, $0.071 * 35 = 2.485$ tokens are erroneous. Statistics like that can then be *once-again* averaged over all OCR datasets, and be used in the generation procedure to impose boundaries on the number of focus entities per arXiv sentence.

9.7.2 Generation from Error Statistics

The arXiv document dataset can also be used **apart** from the "Pure OCR Errors" dataset to generate artificial data. Indeed, one can use statistics about what kind of errors are most often encountered with words of a certain length and artificially insert them into correct words. More particularly, **four** collections of data are required: **single-character** substitution statistics, **double-character** substitution statistics, edit operation **combination** statistics on a *token-level*, and edit operation **position** statistics on a *character-level*. All of these statistics can be extracted from the *isolated* correction samples from the ICDAR datasets, plus the "Pure OCR Errors" datasets, which allows re-using information from the four datasets to create arbitrary many new samples.

The statistics about edit operation **combinations** are contained in a *dictionary of dictionaries*, which holds the number of times a particular **combination** of *edit*

operations was encountered for a word with a specific length n. As an example, consider the word *"an"* with length 2. Then, some possible combinations of edit operations (in theory, there are **infinitely** many), which can artificially transform the correct token into an erroneous one, are:

1. Single insertion: $an \rightarrow ain$
2. Single deletion: $an \rightarrow n$
3. Single substitution $an \rightarrow bn$
4. Insertion + deletion: $an \rightarrow ain \rightarrow ai$
5. Insertion + substitution: $an \rightarrow ain \rightarrow abn$
6. Deletion + substitution: $an \rightarrow n \rightarrow b$
7. Double insertion: $an \rightarrow ani \rightarrow anim$
8. ...

The second piece of required information is the edit operation **positions** on a *character-level*. Manual tests during work on this paper revealed that simply choosing random character positions, at which to carry out edit operations from the combinations above, lead to very dirty artificial samples that were not representative of the original data. Moreover, certain combinations of edit operations can be "grouped" to indicate a larger operation. For example, if we switch the perspective and regard the transformation of a **correct** token into an *erroneous* one, an **insertion**, followed by a **substitution** operation, most often signify a *double-character* substitution, as in the token pair *"@pet"* $\rightarrow$ *"|)et"*.

The *positions data* is once again a *dictionary of dictionaries*, grouped by **sample length** first and by "combination" second. The combinations, in this case, are

contained in strings with the form *xxxinssubxdel*, with '*x*' characters marking a **blank** operation at the respective character position, and '*sub*', '*ins*', and '*del*' standing for the ubiquitous Levenshtein edit operations.

The last two required dictionaries — **single-character** and **double-character** substitution statistics — contain data on how often correct **characters** are mistakenly substituted by OCR systems with different letter combinations. An example of a *one-letter* substitution, that is a really common mistake with OCR systems (refer to [NJC+19]), is *"c"* → *"e"*. A *double-letter* substitution, on the other hand, turns **two** erroneous letters into **one** correct one — e.g., *"l)"* → *"p"*.

Combining the four dictionaries from above, one can use the following steps, in order to generate arbitrarily many *isolated* correction samples:

1. Determine hyperparameter γ, which controls the **fraction** of words, which should be artificially corrupted per *sentence* from the arXiv document

2. Go through the words of a sentence and generate a random number; if the number is $\leq \gamma$, artificially corrupt the word:

    a) First looking up in the dictionary of edit operation combinations; if there are recorded statistics about words with the matching length, choose one of the combinations by using their frequency as **weights** (i.e., if *single substitution* was encountered *2* times, while *single deletion 1*, then there is a two times bigger probability of choosing single substitution)

        - If there are not any combinations for the desired length, just skip the word

    b) Next, take the sampled edit operation combination (e.g., *"sub+2del"*) and check the *positions' data* dictionary for an appropriate template with positions for the edit operations

- If there are not any appropriate templates, just skip the word

c) Go through the characters of the **correct** word and:

i. Append it to the artificial result, if not at pre-determined location

ii. If at pre-determined location and an edit **deletion** has to be executed, skip the character

iii. If at pre-determined location and an edit **insertion** has to be executed, add a *random* character from the task's vocabulary (i.e., lower- and uppercased ASCII letters, digits, punctuation marks (without '@', due to its usage as padding), and whitespace)

- If the insertion is additionally followed by a **substitution** operation, insert a *double-character* substitution instead (e.g., $p \rightarrow$ *|)*); if the dictionary for double-character substitutions does **not** contain any candidates for the picked character, then skip

iv. If at pre-determined location and an edit **substitution** has to be executed, use the *dictionary with substitution statistics* to choose a "logical" replacement for the character; if the dictionary does **not** contain any candidates for the picked character, then skip

3. Create an *isolated* correction sample with the artificial word and its correct alternative

4. Create a *context* correction sample by appending the context to the left and right from the original arXiv sentence

5. After looping through all words, create an error *detection* sample by substituting in the artificially corrupted tokens and encasing them in $<TGT>$ meta tokens

Of course, the problem from subsection 9.7.1 can also be reproduced in the procedure above, if γ is set to be too high. It is also possible to use the same statistics, used in subsection 9.7.1, which record how many focus entities are expected in an error detection sample, based on its *token length*. If using the aforementioned statistics, γ is *ignored*.

Generation of "word boundary" errors

The procedure for generating data from error statistics from above has short-coming when it comes to generating *word boundary* errors — i.e., **run-on** and **incorrect split** mistakes (refer to subsection 7.1.3). Indeed, it is practically impossible in the case of the former, as *run-on* mistakes require **two** words, while the procedure from above always looks at exactly *one*. Moreover, the fraction of *incorrect split* mistakes is also non-representative, as it can only result from an insertion operation picking the whitespace character (out of **94** possible character candidates).

In order to remedy this, we can calculate further statistics from the four original Post-OCR correction datasets (i.e., ICDAR and "Pure OCR Errors" dataset). In particular, we can keep track of edit operations in token pairs, related to adding or removing *whitespaces*, and use them to determine the number of *run-on* and *incorrect split* mistakes for each dataset. Then, we can leverage the average fraction of word boundary errors to create additional errors of that type in the artificial dataset.
For this, a random number is generated every time some word from an arXiv sentence is picked to be artificially corrupted. If the number is lower than the fraction of *run-on*, then an **additional** sample is generated with the correct word, this time containing a *run-on* mistake. The same is done for *incorrect split* mistakes as well, by using an independent random number from the first one.
The generation of an *incorrect split* sample is trivial — a random character position is chosen in the word and a whitespace is inserted after it.
The creation of a *run-on* sample is trickier and needs to respect the sentence length.

Indeed, if the chosen word to be corrupted is the **last** word in the sentence, then it needs to be joined with the word *before* it. Similarly, words in the **beginning** of a sentence can only be joined with words *after*. In the leftover case of the word being in the middle, a coin can be tossed to pick the "merge direction" at random. The *merging* is nothing more than creating a single string out of the two words and deleting the whitespace between them. The resulting *run-on error* sample can then be used as the **input**, while the expected **output** is the original form of the two words, with whitespace between them.

10 Experiments

In this chapter, I will explain how I am going to be training the different approaches of this paper on the task of Post-OCR correction. Moreover, I will detail all experiments I will carry out, in order to evaluate how the models can be best prepared and achieve the best possible results on the task. In particular, section 10.1 will first explain how the evaluation process is done for both the error detection and correction tasks. Then. section 10.2 will cover the experiments I will employ for the **baseline** algorithm with *Q-gram indices*. Afterwards, section 10.3 will explain how the evaluations for the two **external baselines** — using *Google* as a Post-OCR correction engine, and the external *natas* library (refer to end of chapter 2). Next, I will list the experiments for the two **sequence-to-sequence** error correction models — *LSTM encoder-decoder* and *Transformer* — in section 10.4. Afterwards, section 10.5 will go into the experiments and hyperparameters of the *BERT* **detection** model.

10.1 Evaluation and Metrics

This section will explain the techniques, used to evaluate the different models on error detection and correction In particular, the first subsection 10.1.1 will explain how this is done for error correction, while the second one (10.1.2) — for error detection. The final subsection 10.1.3 will explain how I will evaluate the "**pipeline**" of an error detection and correction model — i.e., the *two-step* approach. The described evaluation approaches will not only be employed for deep learning models (i.e., BERT

and sequence-to-sequence models), but also for the baseline Q-index and the external natas model.

10.1.1 Error Correction Evaluation

There are two **types** of metrics that can be used to evaluate error correction models. The first type of metrics I will call **"Levenshtein metrics"**, as they involve calculating the Levenshtein distance and **Character Error Rate** (or, **CER**). In particular, I will be calculating the percent **improvement** or **worsening** of both metrics, when calculated for all pairs of all the **original**, input tokens, versus the **predicted**, output tokens from the models.

In order to explain this point further, consider the following dummy dataset, consisting of the given error correction samples with context size (1, 1): [(*"a <TGT>do rb<TGT> with"* versus *"a <TGT>wo@rd<TGT> with"*) and (*"are <TGT>(w0<TGT> mistakes"*) versus *"are <TGT>two<TGT> mistakes"*]. It is easy to see that the Levenshtein distance between the **focus** tokens of the first samples is **3**, and **2** for the second one. Then, the *overall* sum of Levenshtein distances across all correction samples in the dataset is **5**.

Now let the following two samples be the predicted corrections from the error correction model: *"a <TGT>wo@rb<TGT> with"* and *"are <TGT>lwo<TGT> mistakes"*. The Levenshtein distances of the *predictions* with respect to the expected target tokens are then **1** (*"wo@rb"* versus *"wo@rd"*) and **1** (*"lwo"* versus *"two"*). Then, the percent **improvement** of the sum of Levenshtein distances is $((5-2)/5)*100 = 60\%$.

The same process can be repeated for the other Levenshtein metric - **Character Error Rate**. Character Error Rate represents the percent of characters that have to be corrected, in order to *transform* an erroneous sample into its correct version. Intuitively, a CER of 10% would mean that there is a mistake every **ten** characters. The formal definition for Character Error Rate is given in equation 72. In it, S represents the number of required **substitutions**, I — number of required **insertions**,

D — number of required **deletions**, and C — number of **correct** characters. Notice, however, that the numerator of $S + D + I$ is equivalent to finding the Levenshtein *distance* between an erroneous and target token. Furthermore, the denominator $S + D + C$ is equal to the number of characters (i.e., the **length**) in the *target* token. Therefore, an alternative version of the CER formula is offered in equation 73.

$$CER = \frac{S + D + I}{S + D + C} \tag{72}$$

$$CER(x, y) = \frac{LevDist(x, y)}{|y|}, \text{ with } |y| \text{ being the length of sequence } y \tag{73}$$

[NJCD21] also gives an idea of how to compute information retrieval metrics for Post-OCR correction. In particular, those three information retrieval metrics are **precision**, **recall** and **F1 score**. Before diving into how the metrics are computed, however, one needs to explain the concept of **true positive**, **false positive**, **false negative**, and **true negative** predictions. As explained in Nguyen et al.'s survey paper [NJCD21], the predictions of an error correction model can be classified as follows:

1. character was **wrong**, and it was **corrected** $\implies$ **true positive** (**TP**)
2. character was **not wrong**, but it was still **changed** $\implies$ **false positive** (**FP**)
3. character was **wrong**, but was **not corrected** $\implies$ **false negative** (**FN**)
4. character was **not wrong**, and it was **not changed** $\implies$ **true negative** (**TN**)

The prediction classifications from above can then be used to predict the aforementioned information retrieval metrics. The formulas for precision, recall and F1 score are shown in equations 74, 75 and 76 respectively.

Precision is calculated by dividing the number of **"actually" correct** changes of the model by the number of its **overall** changes. As such, precision can be interpreted to

measure how *accurate* an error prediction model is in its prediction.

Recall, on the other hand, is determined by dividing the number of **"actually" correct** changes by the number of **all** changes — both successfully made, and missed ones. Intuitively, recall can be used to measure how *conservative* a model is with its prediction — i.e., whether it changes a lot of characters to be sure it has the largest chance of correction everything, or is more *calculated* in its predictions. **F1 score** is the **harmonic mean** of the *recall* and *precision* metrics. It is generally used to gauge the overall capability and performance of the model — the higher, the better.

$$Precision = \frac{TP}{TP + FP} \tag{74}$$

$$Recall = \frac{TP}{TP + FN} \tag{75}$$

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{76}$$

10.1.2 Error Detection Evaluation

I will supply **two** ways of evaluating the error detection models of this paper. The first one is the standard token-level evaluation technique, where each class from the detection prediction is compared to the expected target vector. As I will be using BERT for error detection, *token* here stands for the resulting strings from the *BERT tokenizer*. The comparison of prediction with target classes allows us to compute true positive, false positive and false negative predictions, as explained in the previous subsection 10.1.1. In the case of error *detection*, the "meaning" of the different kinds of predictions changes to:

1. token was erroneous, and the model found it $\implies$ **true positive**

2. token was *not* erroneous, but the model predicted it as such $\implies$ **false positive**

3. token was erroneous, but the model *did not* find it $\implies$ **false negative**

The sample classifications from above can then be used to compute the **precision**, **recall** and **f1 score** of a model.

The second type of evaluation I will be carrying out is on **entity-level**. By *entity*, I mean every "block" of predictions, which can be combined to build a sequence. To better understand this, let us use the **"start w/ cont."** marking mode (explained in section 8.2) and look at an example pair of prediction and expected output vectors — $\hat{\boldsymbol{y}} = [0, 2, 3, 1, 1, 2, 1, 2, 0]$ and $\boldsymbol{y} = [0, 1, 1, 2, 3, 2, 3, 3, 0]$. The exact input sequence is unimportant, as the evaluation process only works on the aforementioned class vectors.

The entities within the two output vectors are color-coded — each different color represents a different entity. As such, there are three entities in the *prediction* vector, at positions **(2-3)**, **6**, and 8 respectively. The corresponding entities in the *target* vector are at positions **(4-5)** and **(6-8)**.

In this dummy example, we can then clearly see that the predicted entity **(2-3)** is a **false positive** — meaning the model predicted an erroneous entity, where there was not one. Further, the target entity **(4-5)** is an example of a **false negative** — the model did **not** predict an erroneous entity, where there was in fact one. Lastly, there are two ways to evaluate the prediction for target entity **(6-8)**. One is what I will refer to as the **strict** approach, where the integer classes of the predictions have to **exactly** match the expected classes. In particular, this means that **(6-8)** would again be classified as a *false negative*, as the prediction vector predicted two **different** entities in that specific span. The other approach is the **non-strict** one, where the exact labels of a certain entity do **not** matter, as long as **at least** one matches. If using the latter approach, the two predictions at **6** and 8 would correctly count as one **true positive**, as **two** of the tokens in the target entity were predicted to be *erroneous*.

The *strict* approach bears similarities to the way error detection predictions are decoded, as shown in section 8.4. In order to remain consistent across decoding and evaluating, I will thus be focusing on the **strict** approach when evaluating.

I argue that the evaluation on an *entity-level* is a better indicator of model performance than on a *token-level*. Indeed, the presence of **word boundary** errors in Post-OCR correction makes it very important for an error detection model to properly mark where an erroneous entity *starts* and *ends*. Examples and defense of this argument are given in chapter 8, especially in sections 8.2 and 8.4. Let us briefly consider the cases, where the two evaluation methods might give back different results:
Take the example text sequence *"I am had ing a maiestic day,"* and its correct alternative *"I am having a majestic day."*. The tokenized version of the input is as follows: *['I', 'am', 'had', 'ing', 'a', 'ma', '##iest', '##ic', 'day', ',']*. It is easy to see that the expected target vector $\boldsymbol{y}$ is as shown in equation 77.

$$\boldsymbol{y} = [1, 1, 2, 3, 1, 2, 3, 3, 2, 3] \tag{77}$$

Let the model prediction $\boldsymbol{haty}$ be as shown in equation 78:

$$\hat{\boldsymbol{y}} = [1, 2, 2, 2, 1, 2, 3, 3, 1, 2] \tag{78}$$

Using the **token-level** evaluation method, the result is 4 *true positive* predictions (at positions 3, 6, 7, and 8), 1 *false positive* prediction (at position 2), and 3 *false negative* predictions (at positions 4, 9, and 10). This results in a **precision** value of 80%, a **recall** value of 57%, and an **f1 score** of 67%.
Using the **entity-level** evaluation technique gives back 1 *true positive* entity prediction (at positions 6-8), 1 *false positive* entity prediction (at position 2), and 2 *false negative* entity predictions (at positions 3-4 and 9-10 respectively). This results in a **precision** value of 50%, a **recall** value of 33%, and an **f1 score** of 40%.
As we can see, the two evaluation methods can give back substantially different results. However, I argue that *entity-level* evaluation is much more important in the case of **two-step** Post-OCR correction. This is due to the very nature of the pipeline, as marked erroneous entities by the detection model are passed directly to the error correction model for further processing. Thus, if an entity is *not* marked properly

by its boundaries, the whole pipeline will result in a worse state. Nonetheless, I will *also* be providing **token-level results** for BERT's performance in the pipeline, in order to remain consistent with the evaluation metrics of the other models I will be comparing against.

10.1.3 Two-step Approach Evaluation

In the Post-OCR competitions of the ICDAR events, the evaluation of the error correction task was done with **percent improvement of the sum of Levenshtein distances**. That way, it can easily be seen if the application of an error correction model ultimately led to an *improvement* or *worsening* of the erroneous texts overall. It is unclear whether this is also applied in [NJN+20], where Nguyen et al. use a similar *two-step* approach, consisting of a BERT error detection and an RNN encoder-decoder network with attention for error correction (it is not specified what RNN cells are used). In their paper, it is said by the authors that they measure the percent improvement "based on the difference of the original distance (between GT and OCRed text) and the corrected distance (between GT and the corrected text)". However, it is unclear to me whether **"text"** refers to the **whole** text sequences, or the "correction windows", which the error detection model builds for the error correction model to correct.

In order to emulate the metric from the ICDAR competitions as good as I can, I will be using the following procedure to evaluate the performance of the two-step approach:
First, I will run the BERT error detection model on the test error detection samples for each dataset. From the predictions of BERT, I will build error correction samples, akin to those shown in chapter 7. In order to avoid any confusion around the terminology, I will be referring to those samples as **detection-generated** samples. The *detection-generated* samples will then be "matched" with the **expected** correction samples for each dataset. This results in three distinct groups of correction samples:

- **Matched** correction samples — i.e., the error correction samples that the detection model properly predicted

- **Missed** correction samples — i.e., the *expected* error correction samples that the detection model *missed*, or whose **boundaries** it did not predict properly

- **Superfluous** correction samples — i.e., all *detection-generated* error correction samples, which were *not expected*

These three groups will then be used to compute the overall percentage improvement the pipeline model brings to a certain dataset. In particular, I will be using the *matched* (erroneous and target tokens) and *missed* sets of correction samples to compute the sum of Levenshtein distances of the **original** pairs of *whole* text sequences. I argue that this method properly depicts the sum of Levenshtein distance of all *whole* sequences — the sum distances for an **arbitrary** whole sequence pair comes from the distances of its erroneous tokens, which have to be corrected.
The sum of distances of the **predicted texts** will then be computed by using the sets of *matched* (predicted and target tokens) and *superfluous* correction samples. Additionally, I will be adding the sum of Levenshtein distances from the *missed* set of correction samples on top, for any errors that were missed by the detection model would stay present in the predicted texts as well.
For the sake of completeness, I will be giving information about all three sets of samples when presenting the results of the two-step approach models.

10.2 Baseline Experiments

In this section, I will explain how I will be using the baseline Q-index models for OCR error detection and correction. In particular, subsection 10.2.1 will start off by explaining the Q-gram size I will be employing. Then, subsection 10.2.2 will cover

on which datasets I will be building baseline models on, as well as introduce the hyperparameters I will be testing out.

10.2.1 Q-gram Size

I will be using a Q-gram size of 2 for the baseline models I train. For error detection, the Q-gram size does *not* matter, as errors are determined simply by checking the set of known words of a given Q-gram index. In the case of error correction, preliminary tests have shown that a size of 2 works best. Indeed, this was tested out by using a subset of 50,000 sentences from the Europarl corpus ([Koe05]), which contains transcriptions of European Parliament proceedings. 20% of all *words* from the sentences (36,801 words) were then artificially made dirty, following a "lite" version of the artificial data generation procedure, outlined in subsection 9.7.2. In particular, it is chosen for each word at random whether to execute **1**, **2**, or **3** edit operations, and at which character positions. Then, a random edit operation out of **insertion**, **deletion** and **substitution** was chosen at each character position with a uniform probability. If an *insertion* was chosen, then a random ASCII character was inserted. Similarly, if a *substitution* was chosen, the symbol at the given character position was substituted with a random ASCII character. Finally, if a *deletion* was chosen, the symbol at the given character position was skipped.
Afterwards, the list of artificially dirty words was fed to three different Q-gram index models, with a Q-gram size of **2**, **3** and **4** respectively. The maximum distance of the corrections was set to be **three**, staying consistent with the artificial dirtying procedure from above. The corrected predictions of the three models were compared against the original versions of the dirty words, in order to inspect which model produced the most *right predictions*. It was also counted which model was **strictly** better than the other two ones — i.e., how many times only one specific model gave the correct prediction. The resulting counts are as follows, with the overall number of such cases being **735**:

1. Q-gram size **2**: 229 (31.16%); strictly best in: 151 (19.46%)

2. Q-gram size **3**: 148 (20.14%); strictly best in: 14 (1.9%)

3. Q-gram size **4**: 86 (11.7%); strictly best in: 38 (5.17%)

Table 5 shows a manually-picked subset of the "clashes", in which the different models gave different correction predictions. In particular, having a *low* Q-gram size lower the risk of skipping valid correction candidates because of no shared Q-grams. Take, for example, the *first* line from the table. There, the clean word *have* was artificially dirtied to *oav9*. The Q-gram index with Q-gram size 2 then successfully managed to correct the word back to its original representation, while the indices with Q-gram size 3 and 4 proposed *olavi* — a name from the Europarl dataset. The key observation here is that only a Q-gram size of 2 allows the Q-gram index to consider *have* as a proper correction candidate. Indeed, the Q-grams of different sizes for *oav9* are as follows:

1. Q-gram size **2**: ['$o', 'oa', 'av', 'v9', '9$']

2. Q-gram size **3**: ['$$o', '$oa', 'oav', 'av9', 'v9$', '9$$']

3. Q-gram size **4**: ['$$$o', '$$oa', '$oav', 'oav9', 'av9$', 'v9$$', '9$$$']

A quick manual inspection of *oav9* against *have* shows that the only shared substring between the two words is *"av"*. As seen from above, only using a Q-gram size of 2 allows matching of the substring — which is what leads to only the respective Q-gram index proposing the valid correction. This concept can also be seen in lines 2, 4 and 5 in table 5.

On the other hand, line 3 from the box also shows a case, where having a low Q-gram size leads to the *wrong* prediction. In that case, the deletion of the letter **'b'** from the word *about* results in the Q-gram index with size 2 to consider *"ou"* as one of the shared Q-grams. This leads to the model proposing *you* instead of *about*, as the word

Clean	Dirty	Size 2	Size 3	Size 5
have	oav9	have	olavi	olavi
tobacco	Otoacc	tobacco	tobacco	topic
about	aouA	you	about	about
also	8als\	also	also	-
politeness.	oliDteness	politeness.	lateness	lateness

Table 5: Examples of cases, in which Q-gram index models with different Q-gram sizes gave different correction candidates; **'-'** indicates a blank prediction

was more frequently seen by the model. The statistics from chapter 9, however, have shown that edit *deletions* are much rarer than substitutions, which minimizes the chance of the error happening.

It is worth noting that using a Q-gram index with a Q-gram size of 2 is most computationally expensive. During the aforementioned tests, the model with Q-gram size 2 took almost **6 minutes** to correct all 36,801 artificially dirty words. For comparison, the other two models took about **3 minutes** — two times as fast.

10.2.2 Hyperparameter Combinations

I will "train" the baseline models, which use Q-gram indices, on two different datasets:

1. The **correct** tokens from the *isolated* correction samples from all training datasets — i.e., ICDAR2017 *monograph* and *periodical*, ICDAR2019, and the "Pure OCR Errors" datasets. This results in 55,764 unique tokens, and 2,047 Q-grams without skipping non-English words, and 28,302 unique tokens and 1,044 Q-grams otherwise.

2. 4000 random documents from the arXiv document collection. This results in 371,383 unique tokens, and 3,787 Q-grams, when not skipping non-English words, and 76,481 unique tokens and 950 Q-grams otherwise.

It is important to note that the *correct* tokens of the aforementioned datasets are not always valid corrections. Indeed, the ICDAR datasets often contain multi-word tokens (e.g., *hyphenated* tokens like *"share,-You're"*, or the correct tokens of *run-on* mistakes like *"deep now,"*), archaic forms of words (e.g., *"syres"*, instead of *"sires"*), or flat-out bad corrections (e.g., *"musicke"* → *"muffcke,"*). As explained in subsection 7.1.2 about the filtration process of error correction data, I tried to limit the amount of such cases to a minimum, while not reducing the **complexity** of the task,

I will try out the training process by varying the value of one additional hyperparameter: **skipping non-English tokens**. It will determine if invalid English tokens are skipped when building the Q-gram index. The validity will be determined by using the external library **PyEnchant**[1]. With this, I aim to test out if the detection of erroneous token would get better if only valid English words are included in the Q-gram index. This hyperparameter, however, would also eliminate many words from the training datasets on its own, as many of the correct tokens have *punctuation marks* at the end of them (e.g., *"bower,"*). The hyperparameter is also impervious to **named entities**, such as names of people or business, as they are not valid English words themselves. Overall, I hypothesize that this hyperparameter would **decrease** the performance on *both* error correction and detection, chiefly because of the difficulties that lie in the field of Post-OCR correction when dealing with punctuation and named entities.

It is also important to notice that I will be evaluating the Q-gram indices on **case-insensitive** data only, meaning that all words will be cast to lowercase before being processed. This eliminates the risk of known words in the vocabulary of the index to be misclassified based on different casing (e.g., *"the"* and *"The"*). Additionally, I will be setting the max distance to be **3**, as the data statistics from chapter 9 have shown that the majority of mistakes have an edit distance ≤ 3.

[1]Homepage of the library is: `https://pyenchant.github.io/pyenchant/`

A note on error detection evaluation

As hinted at above, the baseline Q-gram indices will also be evaluated on error **detection**. However, the evaluation procedure for error detection, as described in subsection 10.1.2, computes the evaluation results on an **entity-level**. This means that it is very important whether a multi-token entity is marked as a *whole*, or in parts (e.g., *"<TGT>c@r p3t<TGT>"* and *"<TGT>c@r<TGT> <TGTp3t<TGT>"*. This will *heavily* impact the evaluation results of the baseline algorithm on error detection, as it is **impossible** for the model to detect *word boundary* errors (i.e., *run-on* errors and *incorrect split* errors, as shown above). This is due to the very nature of how the baseline algorithm works, in that it first splits text sequences into its **individual** tokens, before doing operations on them. This problem with word boundary errors is **known** with classical approaches, as is described in chapter 2. Because of this, I will still be carrying out the evaluation for the sake of completeness.

10.3 External Baselines

As mentioned in chapter 2, I will be using two additional external baselines, with which to compare the performances of my baseline and deep learning approaches — the external **natas** library, and **Google**. I will not be performing any additional experiments for the two external baselines, and rather use them out-of-the-box as they are intended. Additionally, I will not be including the spelling correction test datasets of *"Matthias artificial"* and *"Matthias realistic"*, instead only focusing on the Post-OCR correction datasets.

For *natas*, I will be using the **isolated** error correction samples to test the Post-OCR correction abilities of the library, due to the library not supporting **context** correction. Additionally, I will be only taking the **top** correction candidate with the highest confidence (by default, natas returns multiple candidates).

For the error detection task, I will be using the same error detection datasets that I will employ for evaluating the BERT detection model. I, however, expect natas to have very deteriorated results on error detection, as it suffers from the same problem as the baseline algorithm, explained in subsection 10.2.2.

In order to evaluate **Google** as a Post-OCR correction engine, I will take a random sample of **25** error *detection* samples from each of the Post-OCR correction datasets (i.e., ICDAR and "ACL benchmark" datasets) — amounting to **100** error detection samples overall. I use error detection samples, as they have indications for all erroneous tokens in an arbitrary sequence, which can then be corrected as many times as Google can to reach a correction prediction.

10.4 Error Correction Experiments

I will be evaluating two types of encoder-decoder networks for Post-OCR correction — an **LSTM** encoder-decoder network and a **Transformer** model (inherently sequence-to-sequence). I will first run experiments with different configurations of both models, in order to determine the best hyperparameters to train them with (e.g., which attention mechanism works best for LSTM encoder-decoders). These experiments will be trained on the *base* OCR correction datasets — all **ICDAR** and the "**Pure OCR Errors**" datasets. They will then be evaluated on their respective **validation** sets, together with the validation sets of the **ACL** and "**Matthias benchmark**" datasets. The information will then be used to create a **final** model that should perform as good as it can, when it comes to hyperparameter configurations. The final models will then be evaluated on the **test** sets, which will supply the final evaluation results. The final model will also additionally be trained on **artificially** created data from *200,000* erroneous token pairs. Additional statistics for that dataset are given in subsection 11.4.1 in the following chapter.

The experiments of the error correction models suggest a perfect error detection model — i.e., the error correction models only see samples, in which the erroneous token is correctly marked. Because of this, the % improvement rates on some datasets might very well be higher than reported in the papers, introduced in the "Related Work" chapter (e.g., ICDAR2019).
After training the final model, I will also combine the best error detection model with the best LSTM encoder-decoder and the best Transformer correction models respectively and evaluate their joint 'pipeline' performance. The results from the pipeline evaluation would then be directly comparable to the reported metrics from 'Related Work', as the pipeline would take samples in their entirety, find the errors in them, and lastly — correct them.

As far as *LSTM* encoder-decoder models are concerned, I will be carrying out experiments with hyperparameters in the list immediately below. The model presets were manually chosen based on observations during work on what performs best.

- **Attention type** - none, dot, general, and concat (refer to [LPM15])l the number of epochs each model is trained for 75 epochs)
- **Case sensitivity** - all lowercased and mixed-case
- **Context size** of the correction samples
  - (1,1) (trained for **50 epochs** and maximum sequence length of **64**)
  - (3,3) (trained for **50 epochs** and maximum sequence length of **100**)
  - (5,5) (trained for **50 epochs** and maximum sequence length of **128**)
  - (5,1) (trained for **50 epochs** and maximum sequence length of **100**)
- **Model presets** - base and big

- Base:
    - Character embedding size: 128
    - Input dropout: 0.1
    - Hidden state size: 256
    - Number of LSTM layers in encoder: 2
    - Number of LSTM layers in decoder: 1
    - Bidirectional encoder layers: True
    - Encoder recurrent dropout: 0.2
    - Decoder recurrent dropout: 0.1
- Big:
    - Character embedding size: 128
    - Input dropout: 0.2
    - Hidden state size: 512
    - Number of LSTM layers in encoder: 4
    - Number of LSTM layers in decoder: 2
    - Bidirectional encoder layers: True
    - Encoder recurrent dropout: 0.3

        * Decoder recurrent dropout: 0.1

The following hyperparameters values were used as **default**, when they were not overwritten by the appropriate hyperparameters in given experiments:

- Context size: **(1, 1)**
- Case sensitivity: **mixed-case**
- Maximum sequence length: **64**
- Batch size: **256**
- Learning rate: **0.0002**
- Model preset: **base**
- Attention type: **dot**

With the **Transformer** model, the list of variable hyperparameters can be seen in the list immediately below. The *base* model preset was taken from the original Transformer paper [VSP+17], while the other two are meant to represent gradually smaller models.

- **Case sensitivity** - all lowercased and mixed-case
- **Context size** of the correction samples:
  - (1,1) (trained for **50 epochs**)
  - (3,3) (trained for **50 epochs** and maximum sequence length of **100**)
  - (5,5) (trained for **50 epochs** and maximum sequence length of **128**)

  - (5,1) (trained for **50 epochs** and maximum sequence length of **100**)

- **Model presets** - base and big
  - Small:
    * Character embedding size: 256
    * Feed-forward multiplier: 1
    * Number of attention heads: 8
    * Input dropout: 0.1
    * Number of encoder blocks: 2
    * Number of decoder blocks: 2
    * Encoder recurrent dropout: 0.1
    * Decoder recurrent dropout: 0.1
  - Medium (trained for **50 epochs**):
    * Character embedding size: 256
    * Feed-forward multiplier: 4
    * Number of attention heads: 8
    * Input dropout: 0.1
    * Number of encoder blocks: 4

    - Number of decoder blocks: 4
    - Encoder recurrent dropout: 0.1
    - Decoder recurrent dropout: 0.1

The following hyperparameters values were used as **default**, when they were not overwritten by the appropriate hyperparameters in given experiments:

- Context size: **(1, 1)**
- Case sensitivity: **mixed-case**
- Maximum sequence length: **64**
- Batch size: **128**
- Learning rate: **0.0005**
- Epochs: **100**
- Number of warm-up steps: **4000**
- Model preset: **small**

Additionally, for both types of error correction models, **greedy** decoding was used. This means that at every time step of the prediction, the vocabulary character with the maximum probability was chosen. For example, if we have a character-level vocabulary with 'a': 1, 'b': 2, 'c': 3, and a dummy prediction for a certain time step $y_t = [0.5, 0.3, 0.2]$, then the predicted character at time step t is 'a'.

I will also offer the results of another experiment, using a standard Transformer model with *(3, 3)* context size. Namely, I will be testing out how the performance of the

correction model changes when trained on different "*mixes*" of datasets. In particular, this means that the Transformer model will be trained on different combinations of datasets, starting with only the ICDAR2017 **monograph** dataset, and going up to all OCR correction datasets — i.e., all ICDAR and the "Pure OCR Errors" datasets, together with an artificial dataset with *200,000* erroneous token pairs. The idea of this experiment will be to evaluate the **domain specificity** of the Post-OCR correction task, as multiple papers from the "Related Work" chapter (see 2) have suggested that using extra datasets does *not* always lead to better results.

10.5 Error Detection Experiments

The process for evaluating the BERT detection model is the same as the encoder-decoder correction models from the last section 10.4. In particular, I will first run several experiments regarding different configurations of the model, in order to determine what configuration should result in the highest performance. The experiments will be evaluated on the *validation* datasets. At the end, a *final* model will be trained and evaluated on the *test* datasets.
The final model will additionally be trained on an **artificially** generated dataset, which had a limit of **200,000** token pairs. Additional statistics for that dataset are given in subsection 11.4.1 in the following chapter.

The experiments for the BERT detection model include the hyperparameters listed immediately below. The preset experiments only add linear layers **on top** of the BERT model and do not change the architecture of BERT in any way. As such, the architecture of the **base** BERT model (the one used in this paper) stays the same stays the same.

- **Case sensitivity** - all lowercased and mixed-case
- **Fine-tuning** - different combinations of freezing the layers of the BERT model

  - Do not freeze neither the embedding layer, nor any BERT layers
  - Do not freeze the embedding layer, but freeze first **9** layers of BERT
  - Freeze both the embedding layer, and the first **9** layers of BERT
  - Do not freeze the embedding layer, but freeze all BERT layer but the last one
  - Freeze both the embedding layer, and all BERT layers but the last one
  - Freeze both the embedding layer, and all BERT layers
- **Model presets** - base and big
  - Small:
    * Dropout on top of BERT: 0.1
    * Linear layers after dropout: None
  - Base:
    * Dropout on top of BERT: 0.1
    * Linear layers after dropout: [1024, 512]
  - Big:
    * Dropout on top of BERT: 0.1
    * Linear layers after dropout: [4096, 2048, 1024, 512]

The following hyperparameters values were used as **default**, when they were not overwritten by the appropriate hyperparameters in given experiments:

- Case sensitivity: **mixed-case**
- Model preset: **base**
- Freeze embeddings: **False**
- Freeze BERT layers: **None**
- Batch size: **64**
- Maximum sequence length: **128**
- Learning rate: **2e-5**
- Weight decay: **1e-2**

Additionally, I will use a **classification threshold** of *0.98* (explained in 8.4). The value was manually determined after carrying out a few evaluation tests on the validation data with different values from 0.9-0.99. The best-performing classification threshold, which struck the best balance between precision and recall for most datasets, was *0.98*.

11 Results

In this chapter, I will present and discuss the results of the experiments from the previous chapter 10. In particular, I will start by looking at the results of the baseline experiments in section 11.1. Then, I will explore the results for the *two* error correction models in section 11.2, and the BERT detection model results in section 11.3. The experiments from the first three sections were evaluated on the **validation** datasets. The only exception to this are the experiments of the baseline Q-gram index models, as they were directly evaluated on the **test** datasets. Moreover, the results for error correction and detection are provided **independent** of one another (i.e., not as a pipeline).
Section 11.4 then shows the result of combining the best-performing error detection and correction models as a *pipeline*. They will be evaluated on both tasks, and their results compared with all baselines and ICDAR competition models. The chapter finishes off with section 11.5, where I outline the flaws of the final detection and correction models.

11.1 Baseline Experiment Results

The results from the baseline experiments are supplied in two tables. Table 6 shows the results for error *correction*, while table 7 — for error *detection*.

		ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Matthias artificial	Matthias realistic
Training Q-index	Plain	+7.8%	+16.33%	+1.62%	-4.46%	+18.45%	+22.53%
	Skip NE	+7.34%	+14.42%	-1.56%	-11.92%	+17.62%	+22.67%
ArXiv Q-index	Plain	+9.48%	+14.26%	-1.88%	-11.99%	+20.11%	+15.14%
	Skip NE	+8.26%	+12.75%	-4.38%	-14.56%	+20.43%	+23.62%

Table 6: Error correction results for different experiments with baseline Q-index model, evaluated on the **test** datasets
the results are given as % improvement (marked with +) or worsening (marked with -) on the sum of edit distances
olive marks best-performing for *dataset*; **red** marks worst-performing

		ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Matthias artificial	Matthias realistic
Training Q-index	Plain	28.53%	49.2%	35.04%	44.71%	56.74%	29.4%
		28.27%	48.64%	33.55%	43%	55.45%	29.37%
	Skip NE	27.93%	48.13%	35.36%	43.26%	58.52%	33.29%
		27.57%	47.52%	33.76%	41.48%	57.14%	33.22%
ArXiv Q-index	Plain	34.49%	46.58%	32.23%	42.14%	59.69%	29.4%
		34.25%	45.93%	30.81%	40.5%	58.65%	29.36%
	Skip NE	30.17%	47.71%	35.1%	43.45%	63.57%	38.26%
		29.91%	47.1%	33.5%	41.65%	62.32%	38.19%

Table 7: Error detection results for different experiments with baseline Q-index model, evaluated on the **test** datasets
the results are given with *token-level* and *entity-level* F1 scores **in that order**
olive marks best-performing for *dataset*; **red** marks worst-performing, based on **entity-level** F1 score

The experiment results for both tasks show that using a Q-gram index, trained on the correct tokens of the OCR detection datasets, outperforms the alternative — training it on words from the clean arXiv document dataset, when it comes to the ICDAR datasets. On the other hand, the latter technique is better when used on the *spelling correction* datasets and the *ACL benchmark*. This is easily explained — the arXiv document collection consists of proper English research texts. Moreover, the words there are **modern**, which additionally clashes with the archaic words, which are contained in the ICDAR datasets. As a final remark, the arXiv document collection also contains a lot of **technical jargon**, as well as formulas and mathematic symbols, which are in no way present in the ICDAR datasets. This can be used to explain why the arXiv Q-index gram performs better on the "ACL Benchmark" dataset — as mentioned in section 9.4, those are exactly the type of the most common errors in it.

As hypothesized in section 10.2 from the previous "Experiments" chapter, skipping non-English words while training the Q-gram index models does **not** improve their performance. This can be explained with the fact that the OCR correction datasets — especially the ICDAR datasets — often include corrections (i.e., target tokens), which are not valid English words by modern standards (refer to chapter 9). The only notable exception to this are, again, the *spelling correction* datasets, where having only modern English words better fits the expected targets for the task.

11.2 Correction Experiment Results

This section will offer results for the error correction experiments, described in section 10.4 from the previous "Experiments" chapter. I will split the section into three: subsection 11.2.1 will discuss the results for the LSTM encoder-decoder model, subsection 11.2.2 — the results for the Transformer model, and subsection 11.2.3 — the results from the "mixed dataset" experiment.

		ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Pure OCR Errors	Matthias artificial	Matthias realistic
Case sensitivity	All lower (5:39:53)	-8.48% (2% missed)	+32.59% (2% missed)	+27.74% (1% missed)	+15.51% (2% missed)	+34.63% (2% missed)	-21.1% (3% missed)	-17.19% (3% missed)
	Mixed (5:39:04)	-6.21% (2% missed)	+33.54% (2% missed)	+30.59% (0.7% missed)	+20.22% (2% missed)	+37.46% (2% missed)	-19.1% (2% missed)	-15.7% (2% missed)
Context size	1, 1 (4:01:43)	-9.66% (4% missed)	+37.23% (3% missed)	+31.65% (2% missed)	+18% (2% missed)	+39.47% (3% missed)	-18.43% (3% missed)	-16.28% (4% missed)
	3, 3 (5:20:00)	-7.96% (3% missed)	+32.87% (4% missed)	+30.84% (2% missed)	+19.65% (3% missed)	+36.04% (5% missed)	-21.28% (4% missed)	-20.56% (4% missed)
	5, 5 (8:39:29)	-8.52% (2% missed)	+36.43% (2% missed)	+29.27% (1% missed)	+21.62% (0.9% missed)	+37.66% (2% missed)	-24.3% (2% missed)	-16.8% (3% missed)
	5, 1 (5:20:48)	-7.17% (2% missed)	+35.75% (3% missed)	+31.48% (2% missed)	+20.53% (2% missed)	+40.96% (2% missed)	-18.6% (3% missed)	-24.16% (2% missed)
Presets	Base (6:01:40)	-8.03% (2% missed)	+34.85% (2% missed)	+28.98% (2% missed)	+19.69% (2% missed)	+38.7% (2% missed)	-19% (3% missed)	-17.78% (4% missed)
	Big (6:00:18)	-9.17% (1% missed)	+38% (2% missed)	+33.5% (1% missed)	+29.43% (1% missed)	+49.36% (2% missed)	-17.72% (2% missed)	-17.7% (2% missed)
Attention Type	None (5:20:02)	-34.45% (1% missed)	+14.88% (2% missed)	+0.63% (1% missed)	-7.16% (2% missed)	+24.78% (2% missed)	-35.09% (2% missed)	-17.19% (2% missed)
	Dot (5:42:16)	-6.48% (2% missed)	+32.53% (2% missed)	+30.22% (1% missed)	+20.12% (2% missed)	+36.82% (2% missed)	-19,77% (2% missed)	-14.47% (1% missed)
	General (5:36:18)	-8.41% (0.8% missed)	+35.92% (2% missed)	+31.83% (0.6% missed)	+17.51% (1% missed)	+39.27% (2% missed)	-17.84% (1% missed)	-13.85% (1% missed)
	Concat (8:08:41)	-5.14% (0.3% missed)	+35.64% (1% missed)	+30.98% (0,6% missed)	+16.23% (0.5% missed)	+39% (0.75% missed)	-17.17% (1% missed)	-11.84% (1% missed)

Table 8: Results for different experiments with an LSTM encoder-decoder correction model, evaluated on the **validation** datasets; the results are given as % improvement (marked with +) or worsening (marked with **-**) on the sum of edit distances; "*% missed*" indicates what percent of model predictions did not have a focus token, properly encased in $<TGT>$ meta tokens; **olive** marks best-performing in *experiment group*; **red** marks worst-performing (averaged on the five Post-OCR correction datasets); timestamps below experiment names show training time

11.2.1 LSTM Experiment Results

Table 8 visualizes the results for the LSTM encoder-decoder experiments, based on percent change of the sum of Levenshtein distance. I will look more closely into the results of 3 of the 4 experiment groups, namely "*case sensitivity*", "*context size*" and "*attention type*". The result from the *preset* experiment is straight-forward: a bigger LSTM encoder-decoder trains better on the task, suggesting that a bigger model capacity is helpful for Post-OCR correction.

Case sensitivity

At first thought, using *mixed-case* data intuitively looks like a better approach. Indeed, Post-OCR correction often deals with mistakes that transform letters into symbol combination, e.g., *D* to *|)*, or *T* to *l-*. However, a sequence-to-sequence model should theoretically not have a problem with mapping symbol combinations, that are originally for uppercase letters, to their lowercase representations (e.g., *|)* should be mapped directly to *d*).

Box 11.1 shows a random subset of correction samples, which were corrected properly **only** by the *cased* LSTM model. Looking at sample pairs #1, #3, #4, #7, #8, using mixed cased does not seem to provide extra information, which would make a cased model better than an uncased one. Instead, it seems that having mixed cases helps the model *train more easily*, which leads to a better performance.

Sample pairs #2, #5, #6, on the other hand, hint that a cased model seems to learn to work with **named entities** better (e.g., Roman, Swedish, Americas, etc.). Indeed, having all data be lowercased removes the telltale sign of named entities — namely, that they start with uppercase letters. If brought down to all lowercase letters, named entities "blend in" with the rest of the English words, which might confuse the model further.

Cased: (iii) <TGT>Cpmnunications<TGT> among (iii) → (iii) <TGT>Communications<TGT> among
Uncased: (iii) <TGT>cpmnunications<TGT> among → (iii) <TGT>coomunications<TGT> among

Cased: are <TGT>Americas- North<TGT> America → are <TGT>Americas-@North<TGT> America
Uncased: are <TGT>americas- north<TGT> america are → are <TGT>americaa–north<TGT> america

Cased: <TGT>Never<TGT> mind → <TGT>'Never<TGT> mind
Uncased: <TGT>never<TGT> mind → <TGT>'nevee<TGT> mind

Cased: a <TGT>publSshing<TGT> association → a <TGT>publishing<TGT> association
Uncased: a <TGT>publsshing<TGT> association → a <TGT>publissing<TGT> association

Cased: a <TGT>R oman<TGT> Caholic → a <TGT>R@oman<TGT> Caholic
Uncased: a <TGT>r oman<TGT> caholic → a <TGT>rroman<TGT> caholic

Cased: the <TGT>S wedish<TGT> Police → the <TGT>S@wedish<TGT> Police
Uncased: the <TGT>s wedish<TGT> police → the <TGT>sewedish<TGT> police

Cased: design <TGT>an<TGT> construction, → design <TGT>and<TGT> construction,
Uncased: design <TGT>an<TGT> construction, → design <TGT>an<TGT> construction,

Cased: fat <TGT>gentle-man,<TGT> who → fat <TGT>gentle@man,<TGT> who
Uncased: fat <TGT>gentle-man,<TGT> who → fat <TGT>gentle@mmn,<TGT> who

Box 11.1: Examples of correction samples, which were **exclusively** corrected from the *cased* model; the first line of every pair shows the *cased* input erroneous and *output* target samples; the second line shows the *uncased* input erroneous sample and the uncased model *prediction*

Box 11.2, on the other hand, showcase a random subset of the correction samples, which are only properly corrected by the **uncased** model. The sample is smaller, as the same conclusions can be drawn from these samples, as with the previous one. Indeed, using mixed-case samples seems to make the correction model better when dealing with *named entities* (refer to previous paragraph), but also carries the pitfall of recognizing **false positives**. Sample pair #1 from the box can be used to visualize this concept — the lowercased word *nhether* is easy to map back to the correct token *whether*. At the same time, however, *Nhether* could have been recognized by the model as a named entity and thus left uncorrected.

Other than sample pair #1, the other examples from the box do not carry any significant information. It looks like the choice between the two hyperparameters does not have an objectively right answer. Rather, case sensitivity impacts the learning process of the models differently, and using *mixed-case* samples has a slight edge in that regard.

Uncased: <TGT>nhether<TGT> the → <TGT>whether<TGT> the
Cased: <TGT>Nhether<TGT> the → <TGT>Nhether<TGT> the

Uncased: was <TGT>origi,mlly<TGT> introduced → was <TGT>originally<TGT> introduced
Cased: was <TGT>origi,mlly<TGT> introduced → was <TGT>origiaally<TGT> introduced

Uncased: raiders <TGT>(nxw<TGT> skyhawks) → raiders <TGT>(now<TGT> skyhawks)
Cased: Raiders <TGT>(nXw<TGT> Skyhawks) → Raiders <TGT>(aow<TGT> Skyhawks)

Uncased: this <TGT>repl sentation<TGT> was → this <TGT>representation<TGT> was
Cased: This <TGT>repl sentation<TGT> was → This <TGT>replesentation<TGT> was

Uncased: <TGT>for-give<TGT> me → <TGT>for@give<TGT> me
Cased: <TGT>For-give<TGT> me → <TGT>For-give<TGT> me

Box 11.2: Examples of correction samples, which were **exclusively** corrected from the *uncased* model; the first line of every pair shows the *uncased* input erroneous and *output* target samples; the second line shows the *cased* input erroneous sample and the cased model *prediction*

Context size

The context size experiment shows a preference towards **preceding** tokens. The results of the experiments seem consistent with the experiments of the **CLAM** model from the ICDAR competitions from 2017 (refer to [CDCM17]) and 2019 (refer to [RDCM19]). Indeed, the context size of **(5, 1)** – the best-performing one in the experiment group — was directly inspired by the description of the CLAM model in the 2017 iteration. The authors of the model there found that a context size of *(4, 1)* or *(6, 1)* works best. Similarly, the **(5, 5)** context size works best for the ICDAR2019

dataset, again mirroring the results of the CLAM model, where the authors used a *(10, 10)* context size.
As far as the other datasets are concerned, the "**ACL Benchmark**" dataset also seems to profit from additional context, especially from preceding tokens (compare *(5, 1)* versus *(3, 3)*). A similar tendency is also observed with the "**Pure OCR Errors**" dataset. There, using context sizes *(3, 3)* and *(5, 5)*, which feature more *succeeding* tokens, actually **decreases** the performance of the model.

Curiously, the *spelling correction* datasets show **worse** results when using more context. I argue that this lies in the fact that using more context makes the task *more difficult*. Indeed, if we consider correction samples with the default *(1, 1)* context size, there is a higher probability of a target token being a part of a sample with *similar* context. For example, if we consider the most common English word *the*, there is a high probability that the surrounding context around the word often has the form "*in <TGT>the<TGT> ...*", "*of <TGT>the<TGT> ...*", "*at <TGT>the<TGT> ...*" and so on. When the context is increased, the probability of having similar context **drops**. Thus, the error correction model will not be as confident with its correction prediction.
This argument can also be used to explain the bad performances on the OCR correction dataset as well. Indeed, one could make the argument that the *(5, 1)* context size achieves the best results, as the surrounding context *keeps* the knowledge of the **immediate** preceding context, and then uses the extra preceding tokens to further refine its predictions. Moreover, it does not change the correction samples when it comes to the immediate **succeeding** context.

Attention type

The experiment group for the attention type of the LSTM encoder-decoder model shows the most significant results. Indeed, using even the most *basic* form of attention — Luong's **dot attention** — boosts the performance on the Post-OCR correction task

significantly. Moreover, using a deeper form of attention increases the performance of LSTM encoder-decoder models further, but with *diminishing returns* — the **concat** attention mechanism is only very slightly better than **general** on average.

In order to further inspect the discrepancy between **not** using attention and the alternative, I offer a random subset of correction samples, which the *concat* model properly corrected, when compared to the model *without* attention. It can be inspected in box 11.3. Sample pairs #4 through #7 suggest that the "attentionless" LSTM encoder-decoder model seems to struggle with **semantic** and **punctuation** errors. In particular, the last two sample pairs show how the attentionless model *did* predict proper English words as corrections to the erroneous sample — they were, however, not correct in their surrounding context. The rest of the samples show unspecific mistakes, likely due to sole exclusion of an attention mechanism.

W/ att.: straightforward <TGT>appfication<TGT> of → straightforward <TGT>application<TGT> of
W/o att.: straightoorward <TGT>applifation<TGT> of

W/ att.: <TGT>ha !<TGT> → <TGT>ha@!<TGT>
W/o att.: <TGT>ha!!<TGT>

W/ att.: his <TGT>fore-head.<TGT> → his <TGT>fore@head.<TGT>
W/o att.: his <TGT>fore-head.<TGT>

W/ att.: lady <TGT>read,<TGT> which → lady <TGT>read@<TGT> which
W/o att.: lady <TGT>read,<TGT> which

W/ att.: last <TGT>lingerlng,<TGT> tinge → last <TGT>lingering@<TGT> tinge
W/o att.: last <TGT>lingering,<TGT> tinge

W/ att.: after <TGT>com-mitting<TGT> the → after <TGT>com@mitting<TGT> the
W/o att.: after <TGT>com@mutting<TGT> the

W/ att.: freedom <TGT>o f<TGT> speech → freedom <TGT>o@f<TGT> speech
W/o att.: freedom <TGT>off<TGT> speech

Box 11.3: Examples of correction samples, which were **exclusively** corrected from the *concat attention* model; the first line of every pair shows the input erroneous and *output* target samples; the second line shows the incorrect predictions, made by the model **without** attention

"ACL Benchmark" performance

In this subsection, I would like to take a brief look at the results of the experiments on the "**ACL Benchmark**" dataset. The keen reader would have immediately noticed that it is the only OCR detection dataset, which the LSTM encoder-decoder models do not succeed to improve. To that extent, I manually extracted a subset of the error correction samples, which could not be improved by the **concat** LSTM encoder-decoder model, from the "attention type" experiment. They are listed in box 11.4.

of <TGT>GmeptWizatiofls<TGT> Urderlying → of <TGT>Gmeptrization<UNK>s<TGT> Urderlying → of <TGT>Conceptualizatio@ns<TGT> Urderlying

be <TGT>awit r _mmchaw<TGT> → be <TGT>awiterohmmchaa.<TGT> → be <TGT>switched somehow.<TGT>

or <TGT> ixsemglL .<TGT> → or <TGT>AixsemglLs.<TGT> → or <TGT>polysem@ous.<TGT>

of <TGT>'IIYXEght<TGT> and → of <TGT>@@IYXEght<TGT> and → of <TGT>@@Thought<TGT> and

Francism, <TGT>Qlifomia.<TGT> → Francism, <TGT>Q@i<UNK>omia.<TGT> → Francism, <TGT>California.<TGT>

J., <TGT>Gutllet,<TGT> A. → J., <TGT>GusLLET,<TGT> A. → J., <TGT>Guillet,<TGT> A.

4-3-11 <TGT>T keda,<TGT> Kofu → 4-3-11 <TGT>Tikeda,<TGT> Kofu → 4-3-11 <TGT>Takeda,<TGT> Kofu

i: <TGT>f (X,y)<TGT> = → i: <TGT>feee, y<TGT> = → i: <TGT>fi(X,Yy)<TGT> =

<TGT>= <TGT> ('–*OR → <TGT>On<TGT> ('–*OR → <TGT>=><TGT> ('–*OR

by: <TGT>(T∼)-1,<TGT> if → by: <TGT>(Til-1,<TGT> if → by: <TGT>(Tiˆm)ˆ-1,<TGT> if

<TGT>P∼idagogischer<TGT> Verlag → <TGT>Panlaaogischer<TGT> Verlag → <TGT>Päidagogischer<TGT> Verlag

einem <TGT>Gener\]erungssystem<TGT> fHr → einem <TGT>Generakerunnssyystem<TGT> fHr → einem <TGT>Gener@ierungssystem<TGT> fHr

Box 11.4: Examples of correction samples, which the an LSTM encoder-decoder with *concat* attention was not able to correct; each sample triple contains the input erroneous sample, followed by the model prediction and ending with the expected target

I have grouped the main types of erroneous predictions on the "ACL Benchmark" in four. The first main group contains sample pairs #1 through #4. As we can see, the erroneous tokens in this group are either **too dirty** when compared to their respective target tokens (e.g., sample pairs #1 and #4), or are flat-out **mismatched**

(e.g., sample pairs #2 and #3). These sample pairs have an uncharacteristically large Levenshtein distance, when compared with the error distribution statistics from subsection 9.1.2. Even if we give the benefit of the doubt to the correction model, and assume that it has learned to correct errors with a large Levenshtein distance as well, I would argue that those samples would pose a challenge even for *humans* to correct.

The second main group concerns **named entities**. This group is composed of samples #5 through #7. In them, the expected target token is a correction of a named entity (e.g., *Gutllet* to *Guillet*), which the correction model would not know how to correct if it has not seen it often enough. Post-OCR correction has historically had problems with named entities (refer to chapter 2), given that they are properly-spelled, but **invalid** English words. As such, error correction models expectedly often overcorrect unknown named entities, or try to guess what the correct target might be (as is the case with samples #5 through #7).

The third big group consists of corrections to **mathematical equations** or **symbols**. Consider sample pairs #8, #9, and #10. This type of incorrect predictions is expected — the other OCR correction datasets do not feature similar kinds of mistakes, as they are comprised of scans of *literature*. The only other source of training data, which also deals with these kinds of errors, is the artificial datasets, based on the arXiv document collection. This fact will come back up again in the upcoming subsection 11.2.3, where the addition of the artificial dataset will be further inspected.

The last mistake group comprises the last two samples in box 11.4 — correction of **German** words. I intentionally left in the evaluation data, in order to gauge the performance of the correction models on these types of errors. As expected, however, training the correction models on purely English datasets renders this task borderline impossible.

		ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Pure OCR Errors	Matthias artificial	Matthias realistic
Case sensitivity	All lower (5:23:20)	-13.2%	+41.24%	+27.76%	+12.59%	+41.64%	-27%	-26%
	Mixed (5:27:16)	-14.98%	+39.29%	+27.4%	+16.44%	+42.86%	-26.36%	-28.92%
Context size	1, 1 (5:25:38)	-14.98%	+39.29%	+27.4%	+16.44%	+41.37%	-26.36%	-28.92%
	3, 3 (8:35:54)	-15.19%	+39.27%	+27.91%	+13.46%	+39.23%	-26.17%	-25.95%
	5, 5 (13:37:10)	-18.33%	+37.31%	+25.84%	+13.68%	+34.23%	-27.54%	-28.83%
	5, 1 (8:30:33)	-16.35%	+37.19%	+27.22%	+14.18%	+38.93%	-28.34%	-28.33%
Presets	Small (5:27:40)	-14.98%	+39.29%	+27.4%	+16.44%	+42.86%	-26.36%	-28.92%
	Medium (9:22:42)	-14.76%	+40.49%	+29.31%	+18.41%	+46.25%	-25.67%	-26.46%

Table 9: Results for different experiments with a Transformer correction model, evaluated on the **validation** datasets;
the results are given as % improvement (marked with +) or worsening (marked with -) on the sum of edit distances;
olive marks best-performing in *experiment group*; **red** marks worst-performing (averaged on the four Post-OCR correction datasets);
timestamps below experiment names show training time

11.2.2 Transformer Experiment Results

Table 9 showcases the results for the Transformer experiments, again based on percent change of the sum of Levenshtein distance. The results of the "*presets*" and "*case sensitivity*" experiment groups are unsurprising. For a deeper introspection of the two experiment groups, I refer the reader to the previous subsection 11.2.1, where they were explained in the case of using an LSTM encoder-decoder network.

The most surprising part of the experiments with the Transformer model is the **context size**. Looking at table 9, the results for the default **(1, 1)** context size trumps every other combination by a significant margin.
I started inspecting this by first looking at the training process of the *(1, 1)* and *(5, 5)* models. I suspected that the *maximum sequence length* I put was causing too much of the training data being **dropped**, leading to worse performance. This was not the case, as the size of the training set for the (1, 1) model was 82,745, while the other one was 81,537 — a negligible difference.
Then, I picked out a subset of 10 random samples, which the *(1, 1)* model got right, but the *(5, 5* did not. A manual inspection did not reveal anything suspicious, as with the check about the case sensitivity from the previous subsection 11.2.1.
I also hypothesized that the length of the *(5, 5)* samples might have been messing with the attention of the model, so I ran the randomly picked 10 samples through their respective models and plotted the attention scores from the last cross-attention module. A tiny subset of 3 samples was randomly picked to be visualized in figure 20. As seen in the figure, the differences in attention weights for the **focus token** between the two models is also negligible. While it is possible that my random subset landed on exactly such samples. where both models had similar attention weights, I find that to be unlikely.

The only other reasonable explanation about the discrepancy between the two models is **overfitting**. Indeed, inspecting figure 21 shows that both models overfit to the data

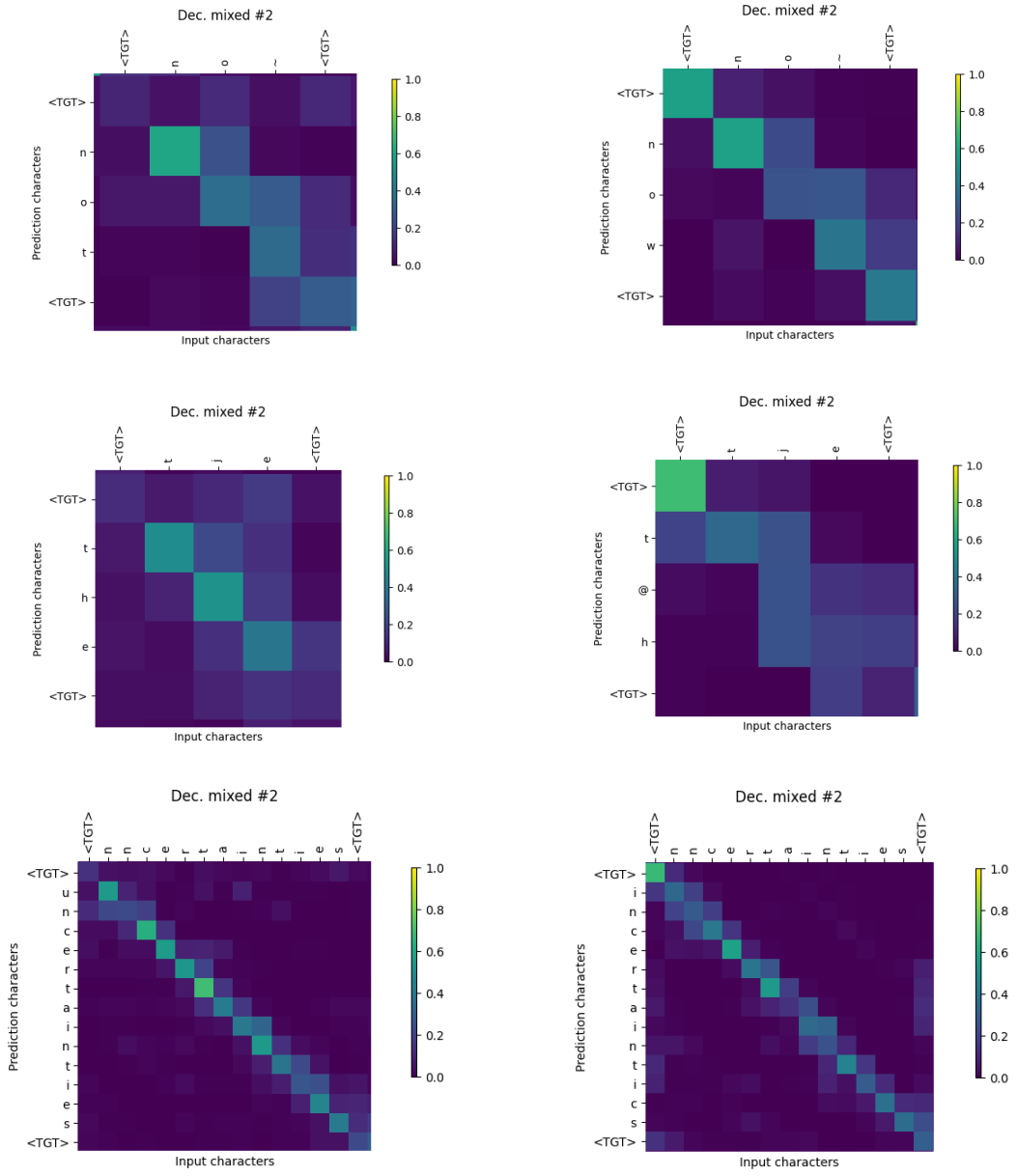


Figure 20: Attention visualization for the focus words on 4 randomly-picked samples from the error correction dataset; the let column shows the attention of an *(1, 1)* context size Transformer, while the right one — a *(5, 5)*

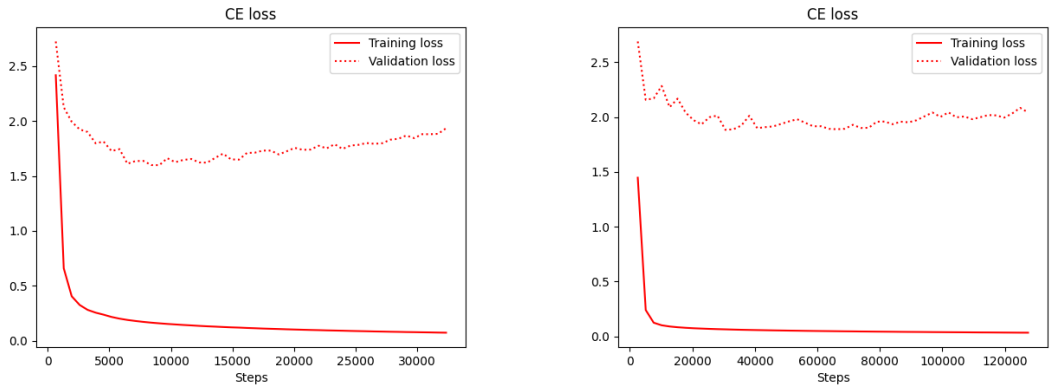


Figure 21: Training and validation loss values throughout the training process of a *(1, 1)* (left) and *(5, 5)* Transformer models

very early into the training process. However, the overfitting was way bigger for the *(5, 5)* model. As explained in the previous subsection 11.2.1, increasing the context size around the target token increases the probability of it being only in *extremely specific* cases. Intuitively explained, having less **variance** in the surrounding context, due to its length (and the amount of tokens included), could lead to a deep model like the Transformer to **fixate** on the target tokens only in exact cases. Inversely, having a *smaller* context size increases the probability of a target token being surrounded by *different* combinations of surrounding tokens, which would stop the Transformer from "memorizing" and force it to learn how the target token *fits* in the language.

11.2.3 Mixed Dataset Experiment Results

Table 10 visualizes the results from the "mixed dataset" experiment. The dataset shows a common theme — using bigger "mixes" of datasets slightly brings the performance on the ICDAR datasets down, depending on the dataset, while the opposite is true for the "ACL Benchmark", "Pure OCR Errors" and the spelling correction datasets. These results mirror the empirical results from [AC18]. There, the authors carried out a similar experiment, where they tried different "levels of mixing", and found that each higher level lead to a worse performance. This **domain-specificity** of

	ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Pure OCR Errors	Matthias artificial	Matthias realistic
Only ICDAR2017 monograph (4:15:30)	-40.09%	+34.21%				-34.79%	-33.39%
Both ICDAR2017 datasets (4:29:38)	-32.43%	+32.69%	+25.46%			-28%	-31.57%
Both ICDAR2017 + ICDAR2019 (4:28:50)	-28.76%	+33.37%	+23.97%	+5.09%		-27.94%	-33.96%
All ICDAR + Pure OCR Errors (4:59:52)	-14.64%	+33.64%	+25.56%	+5.64%	+32.18%	-25.81%	-26.46%
All ICDAR + Pure OCR Errors + Artificial 200k (6:09:54)	+3.79%	+31.11%	+24.02%	+4.27%	+41.99%	-0.62%	-12.13%

Table 10: Results for running a *(3,3)* context Transformer model on different "mixes" of datasets, evaluated on the **validation** datasets; the results are given as % improvement (marked with +) or worsening (marked with -) on the sum of edit distances; **olive** marks best-performing for *dataset*; **red** marks worst-performing; timestamps below experiment names show training time

the Post-OCR correction task was also discussed in the "Related Work" chapter 2. I expected this aspect to be less severe when combining *English-only* datasets, but this does not seem to be the case.

Surprisingly, adding the "Pure OCR Errors" dataset seems to have a positive influence to the performance of the ICDAR2017 periodical and the ICDAR2019 datasets. This suggests that the two datasets contain OCR errors, which are shared with the ones from the ACL Anthology Corpus, from which the "Pure OCR Errors" stems from. Inspecting the table reveals that including an artificial dataset slightly brings down the performance on all ICDAR Post-OCR correction datasets. At the same time, however, it **boosts** the performance on the "ACL Benchmark" and "Pure OCR Errors" datasets **significantly**. One reasonable explanation I can offer for this discrepancy is that the artificial generation procedure *favored* and recreated such errors, which did not match the distribution of the errors in the ICDAR datasets, but rather the ones from the ACL Anthology Reference corpus. While this is highly unlikely — a

quick glimpse in the table 3 with error correction statistics shows that the sum of all correction samples from the ICDAR datasets is almost equal with the ones from the "Pure OCR Errors" dataset — the random nature of the generation procedure does not eliminate the possibility.
Another explanation would be **class imbalance**. The 200,000 artificially-generated erroneous tokens outnumber the number of samples from all other OCR correction datasets. If a model is left to overfit, it might focus on correcting the artificial tokens, leading to a worse performance on the other datasets.

11.3 Detection Experiment Results

The results from the experiments with the BERT detection models are offered in table 11. Unlike the experiments with the error correction models, the results in this case are *not* surprising. Similar to the correction models, using detection samples with mixed-case letters looks to make the training of the model easier, and stacking more linear layers on top of BERT also leads to a better performance.
The "**fine-tuning**" experiment also shows definitive results. Indeed, leaving the whole BERT model **unfrozen** leads to the best metrics, while freezing the whole of it is worst. This is expected, as the Post-OCR correction differs from the traditional NLP tasks, on which BERT is pre-trained.

11.4 Final Results

In this section, I will present the evaluation results of the *final* versions of my proposed models. Moreover, the error correction evaluation will be presented as a *pipeline*, and not solely on the target correction samples with already marked focus entities. The results will be compared to the baseline and external benchmarks, as well as a subset of the models from the ICDAR competitions.

		ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Pure OCR Errors	Matthias artificial	Matthias realistic
Case sensitivity	All lower	43.25%	61.7%	52.82%	52.16%	88%	69.11%	66.28%
	(8:15:52)	38.7%	53%	47.85%	45.18%	80.9%	64.74%	60.56%
	Mixed	55.68%	65.6%	59.15%	61%	89.66%	78.68%	73.59%
	(9:55:49)	46.1%	55.88%	51.74%	50.87%	83.81%	73.65%	67.69%
Presets	Small	55.92%	65.98%	62%	59.67%	90.1%	81.24%	75.34%
	(8:08:14)	44.98%	57.08%	55%	51.1%	84.61%	77.39%	70.34%
	Base	55.68%	65.6%	59.15%	61%	89.66%	78.68%	73.59%
	(8:14:39)	46.1%	55.88%	51.74%	50.87%	83.81%	73.65%	67.69%
	Big	57.89%	68.41%	65.66%	64.33%	90.36%	83.48%	78.61%
	(9:04:27)	49.93%	59.93%	60.14%	56.6%	85.23%	81.24%	73.8%
Fine tuning	Unfr. emb. + no fr. BERT layers	60.87%	68.64%	63.51%	65.34%	88.91%	84.53%	77.26%
	(8:46:42)	50%	60.77%	58.19%	56.43%	83.58%	81.12%	72.71%
	Unfr. emb. + fr. nine layers	57.24%	67%	62.03%	60.93%	90.17%	82.41%	78.1%
	(7:16:08)	43.7%	54.73%	50.61%	48.36%	83.21%	75.33%	70%
	Unfr. emb. + fr. eleven layers	55.12%	64.27%	60.58%	59,26%	88.8%	83.16%	78.16%
	(7:00:54)	36.67%	48.55%	46.35%	41.52%	79.36%	74.25%	68.28%
	Fr. emb. + fr. nine layers	58.25%	66.67%	62.46%	60.54%	89.13%	82.1%	75.1%
	(5:39:42)	41.37%	51.84%	48.64%	45.54%	80.89%	72.83%	66.13%
	Fr. emb. + fr. eleven layers	56.24%	62.83%	59.3%	53.92%	87.1%	80.98%	76.81%
	(5:18:52)	34.59%	43.1%	40.2%	32.57%	73.84%	67.21%	63.48%
	Fr. emb. + fr. all layers	34.1%	42.75%	41.14%	33.19%	58.37%	63.5%	65.57%
	(4:44:51)	15.86%	20.42%	21.39%	13.36%	23.35%	42.76%	48.16%

Table 11: Results for different experiments with a BERT detection model, evaluated on the **validation** datasets with classification threshold *0.98*; the results are given with *token-level* and *entity-level* F1 scores **in that order**
olive marks best-performing for *dataset*; **red** marks worst-performing, based on **entity-level** F1 score;
timestamps below experiment names show training time

In particular, I will start the section of with subsection 11.4.1, which will briefly go over the statistics of the artificial dataset, which I also used when training the final models. Then, subsection 11.4.2 will cover the final error *detection* results. Finally, subsection 11.4.3 will look at the final error *correction* results.

11.4.1 Artificial Data Statistics

For the training of the final sequence-to-sequence correction and BERT detection models, I used artificially-generated datasets comprised of **200,000** target token pairs. I also tried using datasets with a limit of *1,000,000* target token pairs, but the results with the smaller artificial datasets were better. My hypothesis is that having an artificial dataset with way more artificial samples (compare 1,000,000 and 128,588 overall training samples from OCR correction datasets) caused the model to largely focus on the information from the former, leading to worse performance.
The datasets were generated according to the procedure, outlined in section 9.7. Setting the maximum token pair amount to 200,000 resulted in the datasets with the following sizes:

- an *isolated* error correction dataset with 200,000 samples
- a *context* error correction dataset with 200,00 samples
- an error detection dataset with 140,480 samples

I used a fraction of 0.1 to control the number of target entities per error detection sample, as explained in subsection 9.7.1. At the end, the error detection samples ended up having 22,653,909 tokens in total, with 1,616,633 of them being erroneous. This puts the percentage of erroneous tokens at **7.14%**, which is close to the training set average of *8.6%* (refer to 9.1.2). Furthermore, there were 999,020 target *entities*, which averages out to *1.62%* tokens per entity — again close to the training set average.

As far as error correction statistics are concerned, the artificial dataset features 182,483 insertions (3.53%), 362,278 deletions (7%), and 921,238 substitutions (17.82%). Additionally, the distribution of errors by Levenshtein distance was as follows: 654,366 (65.44%) were single-mistake samples, 264,233 (26.42%) were double-mistake samples, 53,266 (5.33%) were triple-mistake samples, and 28119 (2.81%) — multi-mistakes samples (i.e., four mistakes or more). As far as word boundary errors were concerned, the procedure also generated 31,030 (3.1%) run-on and 44,534 (4.45%) incorrect split errors.

All-in-all, the artificially generated datasets fulfilled all average statistics, presented in section 9.1. Although the results from the mixed dataset experiment from subsection 11.2.3 did not show a positive correlation between using an artificial dataset and increase in performance on the OCR correction datasets, I nonetheless decided to include the artificial datasets when training the final models. My reasoning for this is that the experiments from the last section showed better results, when **larger** models were used (both for error correction and detection). Thus, I hypothesized that including the artificial datasets would be beneficial for the large models, as they would have the capacity to learn additional patterns from them.

11.4.2 Final Detection Results

Final model hyperparameters

I used the following hyperparameter configuration to train the final version of the BERT detection model:

- **Model parameters**:
  - Case sensitivity: mixed (upper- and lowercase)

  - Fine-tuning: do not freeze neither the embedding layer, nor any BERT layers
  - Linear layers on top of BERT: [4096, 2048, 1024, 512]
  - Maximum sequence length: 128

- **Training parameters:**
  - Batch size: 64
  - Learning rate: 2e-5
  - Weight decay: 1e-2

The model was trained for **30 epochs**, clocking up to **21 hours** of training time. Figure 22 shows a collage of different statistics for the training process of the model. In particular, subfigure 22a shows the training and *validation* losses of the model. We can see there that the detection model overfits *early* – on the *second* epoch of training. At the second epoch, the model had an average validation loss over all OCR correction datasets (i.e., excluding the spelling correction datasets) of *0.157*, while at the end of the training process — *0.408*. However, the average information retrieval statistics of the model were *better* at the end than after the second epoch. In particular, the model achieved a *precision* value of 82%, a *recall* value of 66%, and an *F1 score* of 73.05%. In comparison, the model had a *precision* value of 80%, a *recall* value of 68%, and an *F1 score* of 73.37% at the end of training. The reason for this can be seen in subfigure 22b — the F1 scores on the **ICDAR** datasets continued getting better beyond the "overfitting threshold". I decided to use the model at the **end** of the training process for that very reason, even though the model was overfit. Subfigure 22c shows a confusion matrix from the final validation step in the training process. As we can see from the matrix, the model was able to predict a 100% of the

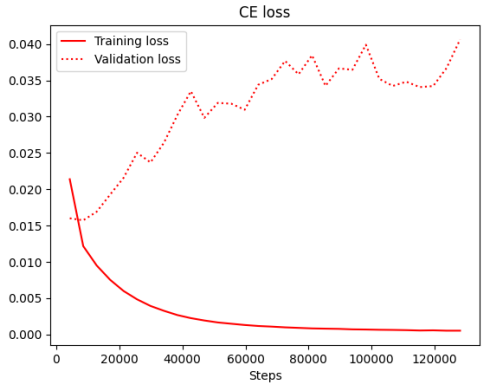
	ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Matthias artificial	Matthias realistic
Plain	28.53%	49.2%	35.04%	44.71%	56.74%	29.39%
Training Q-Index	28.27%	48.63%	33.55%	43%	55.45%	29.37%
w/ max. dist. 3						
Char-SMT/NMT	x	67% -	64% -	x	x	x
WFST-PostOCR	x	73% -	68% -	x	x	x
CLAM	x	67% -	x	45% -	x	x
CCC	x	x	x	67% -	x	x
Nguyen et al.	x	72% -	74% -	68% -	x	x
Google Autocorrect			36.93% 39.04%		x	x
NATAS	10.05%	27.53%	23.54%	28.42%	42.04%	27.81%
	9.81%	26.94%	22.1%	26.92%	40.51%	27.77%
Big Unfrozen BERT	51.61%	57.13%	52.37%	42%	81.29%	76.54%
	46.43%	57.87%	48.49%	32.2%	79.05%	74.35%

Table 12: Final results for OCR error detection with classification threshold *0.98* on the **testing** datasets, given in **F1 score**;
'x' indicates that the model was not evaluated on the given dataset when it was published;
'-' indicates that the authors only supplied *token-level* evaluation
olive indicates the best-performing model on a specific dataset, based on *token-level* evaluation

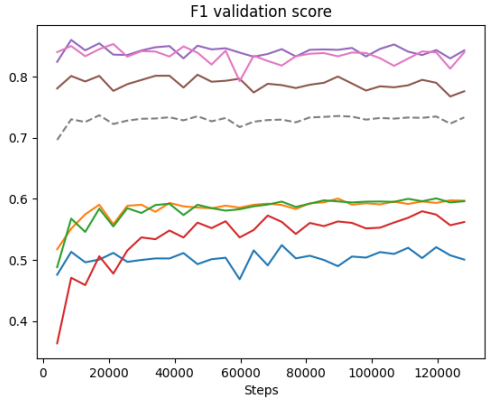
miscellaneous tokens (i.e., BERT meta tokens), $\sim 99\%$ of the non-erroneous tokens, $\sim 70\%$ of the "start erroneous" tokens, and $\sim 77\%$ of the erroneous continuation tokens. The model seems to have properly learned to distinguish between the two types of erroneous tokens — there is not much overlapping mismatches between the two.

Comparison table w/ baselines and SOTA

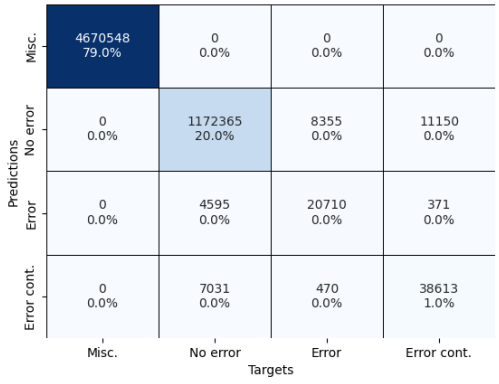
Table 12 offers the evaluation results of the final BERT detection model on all test datasets. As clearly seen in the table, the model gets better results than the benchmark models (i.e., Q-gram Index, NATAS and Google on average), but is underwhelming when it comes to the current state-of-the-art models.



(a) Training and validation loss while training the final detection model



(b) F1 score evolution on the **validation** datasets while training the final detection model blue stands for "ACL Benchmark"; orange stands for ICDAR2017 monograph; green stands for ICDAR2017 periodical; red stands for ICDAR2019; purple stands for "Matthias artificial"; brown stands for "Matthias realistic"; pink stands for "Pure OCR Errors"; the **dotted** line stands for the **average** over all OCR correction datasets



(c) Confusion matrix of the final detection model on the **validation** datasets

Figure 22: Training statistics for the final BERT detection model

		ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Matthias artificial	Matthias realistic
Plain Training Q-Index w/ max. dist. 3	Matched	16.45%	31.62%	19.45%	26.34%	36.93%	17.19%
	Missed	4.82%	15.86%	25.64%	25.86%	16.04%	16.75%
	Superfluous	78.73%	52.52%	55%	47.8%	47%	66.06%
Google Autocorrect	Matched	22.44%				x	x
	Missed	54.15%				x	x
	Superfluous	23.41%				x	x
NATAS	Matched	5.2%	15.33%	11.92%	14.91%	4.27%	3.46%
	Missed	4.77%	8.92%	12.36%	13.82%	31.4%	19.62%
	Superfluous	90%	75.92%	75.72%	71.25%	64.34%	76.92%
BERT + (3,3)	Matched	28.63%	37.8%	30%	17.47%	62.67%	59.64%
	Missed	48.66%	42.1%	47.47%	63.53%	30.13%	33.95%
	Superfluous	22.71%	20.1%	22.58%	19%	7.2%	6.4%
BERT + Isolated	Matched	29.01%	38.51%	30.42%	18.29%	62.88%	59.38%
	Missed	48.66%	42.83%	47.38%	63.42%	30.1%	34.29%
	Superfluous	22.33%	18.66%	22.2%	18.29%	7.1%	6.33%

Table 13: Fractions of **detection-generated** samples across different datasets into *matched* (i.e., properly generated), *missed* (i.e., skipped over), and *superfluous* (i.e., improperly generated); the BERT model combinations use a **classification** threshold of 0.98

Distribution of predicted samples

The performance of the error detection model is crucial to the overall performance of the two-step approach. As explained in the paper previously, the two-step approach hinges on the detection model marking as many **truly erroneous** tokens as possible, without also generating excess superfluous samples. To that extent, table 13 showcases the distribution of **detector-generated** correction samples, split into the three distinct groups from subsection 10.1.3. The distribution is also given for the Q-index and external baselines.

Note that there are different results when running a *context* correction error versus an *isolated* one. This is due to the very nature of the "**matching process**", which joins detector-generated correction samples with expected ones. Indeed, the matching is done by matching the *erroneous*, or **input** parts of the samples, as the detection model does not have anything other than the erroneous test samples. In the case of

isolated error correction, however, the inputs are comprised of a single token, encased in *<TGT>* meta tokens. This means that there is a chance of having the same input sample multiple times, even though the *context*, from which it was **extracted**, might have been *different.*

The first thing that may stand out in the eyes of the reader is how big the fraction of **superfluous** — i.e., a model predicted an erroneous token where there is not one — samples is with the baseline Q-gram index and NATAS models. The value is *extraordinarily* high with the "*ACL Benchmark*" dataset, and box 11.5 visualizes why this is. As explained in section 9.4, the "ACL Benchmark" dataset contains a lot of invalid English words, like *URLs*, *named entities*, *technical jargon and mathematical symbols*, ***German*** *words*, *meta symbols* (e.g., citation symbol) and so much more. This clashes with the way error detection is implemented with a Q-gram index model, as erroneous words are just checked against a vocabulary of *known words.* The high amount of superfluous samples is then logical — each one of the aforementioned challenging groups of tokens to work with is caught by the Q-index and loaded up for correction. What is interesting is that in most cases the Q-gram index does *not* end up correcting the token, as it can not find a suitable replacement for it within the given maximum edit distance of *three.*
The similar phenomenon is also experienced with the NATAS library, which suggests that it also uses some kind of dictionary approach to classify error before correcting them.

"*<TGT>http://www.icsi.berkeley.edu/~framenet<TGT>*", "*<TGT>"hat",<TGT>*",
"*<TGT>(as<TGT>*", "*<TGT>clarity<TGT>*", "*<TGT>inflammation<TGT>*",
"*<TGT>c(i)<TGT>*", "*<TGT>31,777<TGT>*", "*<TGT>1986<TGT>*, '*<TGT>georgetown<TGT>*',
'*<TGT>prominenzbasierte<TGT>*', '*<TGT>prosodieanalyse<TGT>*', '*<TGT>75%,<TGT>*',
'*<TGT>[4]<TGT>*'"

Box 11.5: A random subset of **superfluous** correction samples, generated by the final Q-gram index from the "ACL Benchmark" dataset; each line represents one detector-generated correction sample

Opposite of the two aforementioned models is the Google Autocorrect engine, as well as the BERT detection model. Indeed, the Google Autocorrect engine shows an incline to correct words with a low Levenshtein distance and with mistakes that would more-so fit *spelling correction*, rather than Post-OCR correction. For example, it properly managed to correct "*Orimea,*" to "*Crimea,*", "*marvadous*" to "*marvelous*", "*unthrifti nesse*" to "*unthriftiness*". However, it also showed a tendency to normalize historical words, with corrections like "*HISTORIE*" to "*HISTORY*" and "*almightie*" to "*almighty*", which were **not** erroneous in the case of the evaluation. The Google Autocorrect engine would also try to make *larger* "corrections" (e.g., "*he was sittir was*" to "*he was*", as well as punctuation fixes (e.g., "*an swered, that*" to "*answered that*").

All of these tendencies are reminiscent of, well, an autocorrect engine on a phone, which helps stop mistakes that **humans** make while typing (i.e., *spelling correction*). As such, the Google Autocorrect engine is **not** a good Post-OCR correction engine, and a shoddy OCR detection one. Indeed, it skips almost all cases, where there is a typical Post-OCR correction mistake, which does not even look that much harder than spelling correction examples. As an example, the following sentence does not feature a **single** correction from the Autocorrect engine, even though more than a couple are needed: "*\<TGT>[n<TGT> some <TGT>(:rises,<TGT> the <TGT>he:gallon<TGT> of \[A is ingr <TGT>It\]<TGT> suggests the <TGT>ingredi(mcc<TGT> of the object A to another type <TGT>(J,<TGT> such that there exists a type <TGT>1)<TGT> which is greater than <TGT>13<TGT> and <TGT>(7<TGT> in the lattice of types: The spoke wheel is not a part of a <TGT>ear<TGT> (it <TGT>Zs<TGT> part of a <TGT>bike').<TGT>*"

With the **BERT** detection model, the distribution of the sample is more or less *expected*. Indeed, the idea behind the model (as explained all the way back in chapter 8) is to **only** catch and relay samples with a *guaranteed* error in them, as the correction models were *not* trained to handle false positive samples (i.e., a correct word is incorrectly passed to the correction model). On inspection of the distribution

statistics from table 13, one can see how this technique is very **flawed**, when it comes to the Post-OCR correction field. Curiously, this strategy seems to work wonders for **spelling correction**, as around **60%** of the erroneous tokens there are properly caught, without **ever** training on a single sample from a spelling correction dataset. Coming back to Post-OCR correction, however — the design of the two-step approach, together with the classification threshold of **0.98**, lead to a *massive* amount of missed erroneous tokens. These missed samples prove detrimental to the overall performance of the pipeline, due to how the pipeline evaluation process works — the Levenshtein distance from every missed sample pair is added to the overall sum of Levenshtein distance of the **predicted** texts, because the error stays in-tact. The following subsection on error *correction* results will delve deeper into how much the missed samples hurt the end results, but the fractions promise it to be a lot.

I also briefly want to discuss and visualize the different classes of samples, which the BERT detection model generates. As the result is most striking on the **ICDAR2019** dataset, I will be focusing my exploration there — specifically, "BERT + (3, 3)". For this, I'm going to do a quick exercise of picking 20 samples from the group of **missed** samples. By doing this, I want to find out if it is really the model that is at fault with how the results turned out, or is the data just too dirty.
The randomly-picked missed samples are displayed in box 11.6. Some samples suggest that the test set of the ICDAR2019 dataset is pretty dirty — there is unnecessary addition in the sample pair "*Meet*" → "*Meet me*", unnecessary deletion in "*Pariars. , 3*" → "*Pariars.*", random correction of punctuation marks (e.g., "*jade!*" → "*jade !*" and "*withall,*" → "*withall ,*"), and so on. A telltale sign about how dirty the testing set actually is just inspecting the start-of-the-art — the current SOTA on the ICDAR2019 dataset is **CCC**, with a mere **11%** Levenshtein distance sum improvement. Moreover, I mentioned in the "Related Work" chapter that the ICDAR evaluation script was hard to set up with the two-step approach I'm using, so I'm not sure if the intended evaluation strips away more or less tokens than I did.

“<TGT>foul<TGT>” → “<TGT>soul<TGT>”, “reply3d” → “reply’d”, “y” → “)”, “A H!” → “AH !”, “Meet” → “Meet me”, “jade!” → “jade !”, “os” → “of”, “withall,” → “withall ,”, “REARING” → “BEARING”, “LTSTON,” → “LISTON,”, “moft” → “most”, “Pariars. , 3” → “Pariars.”, “8” → “B”, “own” → “ovvn”, “en” → “on”, “xcviii.” → “XCVIII.”, “Shil” → “Shil3”, “Scius.” → “Seius”, “Eroxena,” → “Eroxena.”, “PALACE 3” → “PALACE”

Box 11.6: A random subset of 20 missed ICDAR2019 samples

11.4.3 Final Correction Results

For this subsection, I will first introduce the tables with the final correction statistics for the paper. The tables will feature four variations of the *two-step* approach:

- Two-step approach with BERT and **(3, 3)** context LSTM encoder-decoder
- Two-step approach with BERT and **isolated** LSTM encoder-decoder
- Two-step approach with BERT and **(3, 3)** context Transformer
- Two-step approach with BERT and **isolated** Transformer

After inspecting the tables, I will go into more details in the best ones, sharing training statistics and doing error analysis. The keen reader might question why I am using context size *(3, 3)*, when the experiments from the first section 11.2 showed that it is not optimal for either of the correction models — I refer the reader to last section 11.5 for an explanation.

Without further ado, table 14 shows the overall results for Post-OCR correction by using the aforementioned four two-step approaches. The metrics are given in *% improvement on the sum of Levenshtein distances* of all prediction tests, as explained in subsection 10.1.3 from the previous chapter.
As with error detection, the two-step approaches achieve a large improvement over the baseline Q-gram index and external baselines, but pales in comparison to the

	ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Matthias artificial	Matthias realistic
Plain Training Q-Index w/ max. dist. 3	-76.52%	-52.1%	-52.33%	-46.28%	-46.72%	-68.58%
Char-SMT/NMT	x	+43%	+37%	x	x	x
WFST-PostOCR	x	+28%	x	x	x	x
CLAM	x	+29%	+22%	+0.4%	x	x
CCC	x	x	x	+11%	x	x
Nguyen et al.	x	+36%	+27%	+4%	x	x
Google Autocorrect	-21%				x	x
NATAS	-92.5%	-81.84%	-81.16%	-75%	-74%	-84.48%
2-step w/ (3,3) LSTM	-8.1%	+2.38%	-7.3%	-9.72%	+8.36%	+8.38%
2-step w/ isolated LSTM	-8.45%	+1.56%	-9.31%	-10.61%	+3.22%	+5.34%
2-step w/ (3,3) Transf.	-12.99%	-4.3%	-11.95%	-14.31%	+3.43%	+2.02%
2-step w/ isolated Transf.	-8.2%	+2.2%	-9.99%	-10.67%	+2.39%	-2.29%

Table 14: Final results for OCR error correction on the **testing** datasets, given in **% improvement of the sum of Levenshtein distances**;
'x' indicates that the model was not evaluated on the given dataset when it was published;
olive indicates the best-performing model on a specific dataset

	ACL	ICDAR2017 monograph	ICDAR2017 periodical	ICDAR2019	Matthias artificial	Matthias realistic
Plain Training Q-Index w/ max. dist. 3	+9.33%	+19.78%	-1.7%	-8.45%	+17.38%	+41.65%
Google Autocorrect	+23.88%				x	x
NATAS	-46.67%	-26.73%	-44.76%	-43.43%	-39.18%	-34.36%
(3,3) LSTM	+35.57%	+52.72%	+41.97%	+34.17%	+22.1%	+22.94%
Isolated LSTM	+37.65%	+49.3%	+33.72%	+26%	+14.86%	+18.6%
(3,3) Transf.	+31.23%	+52.76%	+38.54%	+26.25%	+19.59%	+16%
Isolated Transf.	+43.92%	+53.29%	+37.17%	+30%	+14.99%	+7.8%

Table 15: Performance of the different models from the paper exclusively on the **matched** group of correction samples (i.e., which the detection model recognized properly)
olive indicates the best-performing model on a specific dataset

results, achieved from the models from the ICDAR competitions. As hinted at in the previous subsection with the results for error detection, this is mostly due to the large amount of **non-matched** samples. Indeed, table 15 shows another perspective of the final correction results — this time looking at the percent improvement of **only** the samples from the *matched* dataset. As we can see there, the sequence-to-sequence error correction models actually achieve very nice results when they receive the proper erroneous tokens from the error detection model. Two of the models stand-out in particular — the **(3,3) LSTM** and the **isolated Transformer** models. The presence of the latter model is a very interesting result, as it essentially re-affirms the hypothesis from the Transformer "context size" experiment, where the model with the **least** context worked the best. It seems that this also extends to using *no context at all*. It is also curious to observe that this does *not* apply to the spelling correction datasets, where the opposite is true — the models with context size *(3, 3)* have a huge lead over their counterparts.

Coming back to table 14 with the overall % improvement, I wanted to highlight how big of an influence the error detection model has on the overall performance of the pipeline. First, we will look at the baseline Q-gram index model, which achieves a healthy +19.78% increase on the matched samples from the *ICDAR2017 monograph* dataset. If speaking in the sums of Levenshtein distances themselves, the baseline model brings the sum down from 11,038 on the original erroneous texts to **8,855** on the predicted texts. However, the leftover subset of *missed* samples, which counts up to *4156* samples, adds another **5315** on top of the sum of Levenshtein distances. These two sums, however, mean absolutely nothing when added to the sum of Levenshtein distances, which the Q-gram index accumulated "correcting" the **13,759** superfluous samples — **19,954**. Therefore, the sum of the Levenshtein distances are the original $11,038 + 5,315 = 16,353$, while that of the predicted texts becomes $8,855 + 5,315 + 19,954 = 34,124$, or -52.1% worsening.

The same observation can also be made with the four *two-step* approaches. If we once again take the best-performing two-step model on the *ICDAR2017 monograph*

— with the **isolated Transformer** model — we can see that the correction model achieves a staggering result of **+53.29%** on the matched samples (as seen in table 15). Speaking in real numbers, the sum of Levenshtein distances goes from 7,985 on the original texts to **3,730** on the predicted ones. This achievement, however, is overshadowed by the sheer amount of *missed* and *superfluous* samples, which add another **8,858** and **3,884** to the sum of Levenshtein distances accordingly. Still, even with the addition of the *non-matched* samples, the two-step approach with an isolated transformer still manages to achieve a net *+2.2%* improvement on the ICDAR2017 dataset. While that result is dreadful when compared with the ones from the competitions above, I find it promising that the model achieved a positive result in the presence of all the negative influence from the non-matched samples.

Final LSTM encoder-decoder model

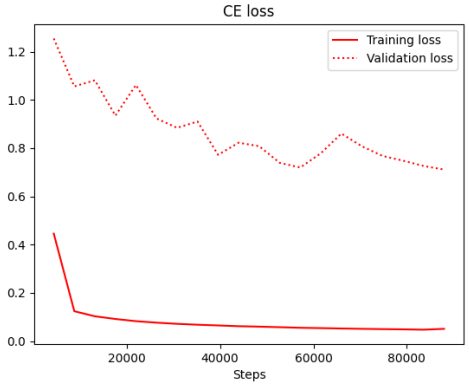
As determined in the last subsection, the **(3,3)** context LSTM model performs better on the Post-OCR correction datasets than the *isolated* one. This is not a surprise, as it is echoed by many papers in the "Related Work" chapter 2. What is more of a surprise is that the context LSTM model is the **best-performing** model out of all four final correction models — meaning that it also beat out the **isolated Transformer** model by a *very slight* margin. While this does not yet confirm that Transformers are better than LSTM encoder-decoder networks (as is the case with many other NLP tasks) on Post-OCR correction, it shows that Transformer models can also be employed for the task. This was one of my biggest questions when starting to work on this project, as even state-of-the-art *two-step* approaches still stuck to using LSTM encoder-decoder networks. This paper can be used as a stepping stone and motivation to explore Transformers for Post-OCR correction further.

Back to the *(3, 3)* LSTM model, its full list of hyperparameters is as follows:

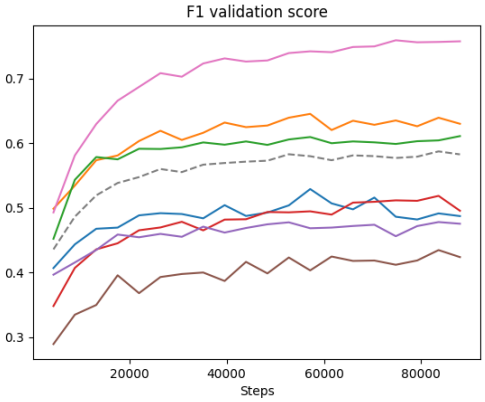
- **Model parameters**:

  - Case sensitivity: mixed (upper- and lowercase)
  - Character embedding size: 128
  - Input dropout: 0.2
  - Hidden state size: 512
  - # LSTM layers in encoder: 4
  - Bidirectional encoder layers: True
  - Encoder dropout: 0.3
  - # LSTM layers in decoder: 2
  - Decoder dropout: 0.2
  - Attention type: dot
  - Maximum sequence length: 100

- **Training parameters**:
  - Teacher-forcing probability: 0.5
  - Context size: (3, 3)
  - Batch size: 64
  - Learning rate: 1e-3



(a) Training and validation loss while training the final LSTM correction model



(b) F1 score evolution on the **validation** datasets while training the final LSTM correction model

blue stands for "ACL Benchmark"; orange stands for ICDAR2017 monograph; green stands for ICDAR2017 periodical; red stands for ICDAR2019; purple stands for "Matthias artificial"; brown stands for "Matthias realistic"; pink stands for "Pure OCR Errors"; the **dotted** line stands for the **average** over all OCR correction datasets

Figure 23: Training statistics for the final LSTM encoder-decoder correction model

The model trained for **20 epochs**, running for **9 hours** in total. Figure 23 shows a collage of training statistic for the model during the training process. Compared to the BERT detection model, the loss values in subfigure 23a do *not* indicate overfitting from the start. However, they visualize a divergence between the training and validation loss since the very beginning, which can be taken as a hint that the two types of sets have different *distributions*. I can sadly give no further ideas for why this happens, as the dataset overview section 9.1 showed that all correction datasets follow roughly the same error statistics. Nonetheless, the model trained improved its validation loss up until the last epoch, reaching the final validation loss of 0.71. The best average information retrieval metrics were also achieved in the last step (as can be seen in subfigure 23b), reaching a **precision** value of 56%, a **recall** value of 61%, and an **F1 score** of 58% over all OCR correction datasets (i.e., excluding the spelling correction datasets). I purposely trained the model for 20 epochs, as previous tests showed that it begins to overfit after hitting around 60% average F1 score.

Final Transformer model

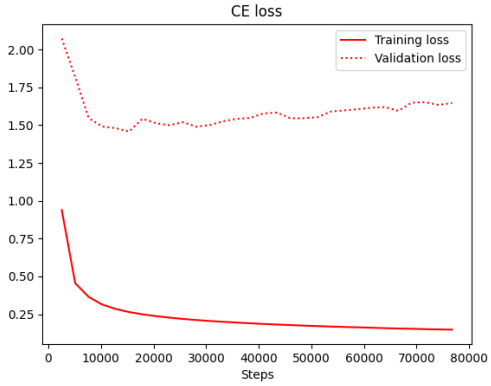
As for one of the most surprising results from this paper, the **isolated** Transformer model achieved a better overall performance on the test OCR correction datasets than its context counter-part. Its full list of hyperparameters is offered immediately below:

- **Model parameters**:
  - Case sensitivity: mixed (upper- and lowercase)
  - Character embedding size: 256
  - Feed-forward multiplier: 4
  - # of attention heads: 8

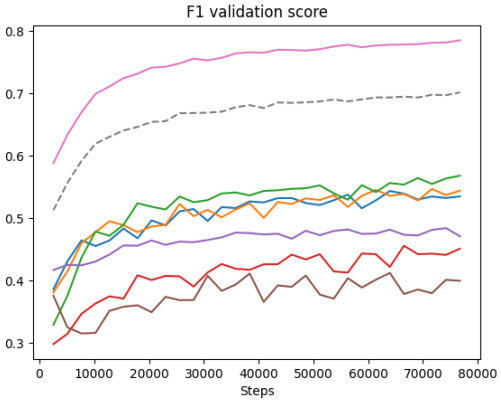
  - Input dropout: 0.1
  - # of Transformer blocks in encoder: 4
  - Encoder dropout: 0.1
  - # of Transformer blocks in decoder: 4
  - Decoder dropout: 0.1
  - Maximum sequence length: 64

- **Training parameters**:
  - Context size: (0, 0) (i.e., isolated correction)
  - Batch size: 128
  - Learning rate: 5e-4
  - Warm-up steps 4000

The model trained for **20 epochs**, running for **12 hours** in total. Figure 24 shows a collage of training statistic for the model during the training process. As with the LSTM encoder-decoder model, the two losses diverge from the get go (refer to subfigure 24a), but this time quickly overfitting at epoch 5. The validation loss then goes from 1.46 from epoch 5 to 1.65 at epoch 20. As with the BERT detection model, however, the F1 scores on the validation sets kept getting bigger even after the overfitting point. As such, they ended at a **precision** value of 68%, a **recall** value of 71%, and an **F1 score** of 70% (as seen in subfigure 24b). Because of this, I also opted to use the version of the model at its final epoch, rather than at epoch 5.



(a) Training and validation loss while training the final Transformer correction model



(b) F1 score evolution on the **validation** datasets while training the final Transformer correction model
blue stands for "ACL Benchmark"; orange stands for ICDAR2017 monograph; green stands for ICDAR2017 periodical; red stands for ICDAR2019; purple stands for "Matthias artificial"; brown stands for "Matthias realistic"; pink stands for "Pure OCR Errors"; the **dotted** line stands for the **average** over all OCR correction datasets

Figure 24: Training statistics for the final Transformer encoder-decoder correction model

For the isolated Transformer model, I also organized a quick exercise to research what types of errors the model still struggles with. I felt like this would give deeper insight into how the model works, as well as motivate some future direction, in which the model should be researched. To that extent, I manually went through the testing datasets and picked out cases, which seemed interesting, and then grouped them. The result of the process follows immediately below:

- **Non-natural language**
  - "*rdf:nodeID='A1'>?.*" → "*Ref@mode@@@@nd@@l.*" → "*rdf:nodeID='A1'>....*"
  - "*C)..()99*" → "*C@@@Rogs*" → "*@0.@@999*"
  - "*Pr(pln1)*" → "*Propl@am*" → "*Pr(p/n1)*"
  - "*xxxvii.*" → "*@Exavii-*" → "*LXXXVII.*"
- **Context problems**
  - "*It*" → "*If*" → "*it*"
  - "*ee*" → "*we*" → "*see*"
  - "*plan∼,*" → "*plane.*" → "*plan@,*"
- **Named entities**
  - "*Callfo n\ſa*" "*Callfo@@mia*" → "*Californ@ia*"
  - "*Perctvall,*" "*Percivall,*" "*PERCIVALL,*"
  - "*Sabah*" → "*Sabal*" → "*SARAH*"<TGT>
- **Punctuation-related errors**
  - "*infcrcncc)*" → "*@inferenc@e*" → "*(inference)*"
  - "*?innate ?language*" → "*@innate@@language*" → "*"innate" language*"

11.5 Flaws in the Results

As visualized in subsections 11.4.2 and 11.4.3, some final models appear to have overfit very early in the training process. This is most apparent with the Transformer error correction models, whose validation and training losses are shown to diverge from the start of the training process. I acknowledge this is as an error in the evaluation process. The reason for this inconsistency lies in time troubles. I assume that using *non-overfit* models would increase the performance of the overall pipeline — however, I do not expect that boost to make up the difference to the best-performing models in the final result tables (see 12 and 14).

Additionally, I used *(3, 3)* context size for both error correction models, even though it was not optimal for either one of the according to section 11.2. As in the previous problem, this happened due to time problems and unexpected problems with the university cluster (downtime), on which I was training my models.

12 Conclusion

This paper researched using a *two-step* approach to do Post-OCR correction, influenced by papers like [SN20] and [NJN+20]. I employed BERT as the error detection model, and tried out two different sequence-to-sequence models for the correction sub-task: LSTMs and Transformers. The two-step approach was meant to function in such a way, in which the error correction models only trained on **erroneous** samples, in order to be able to learn as much as they can from the underlying patterns. In order to facilitate this working on whole sequences, the BERT error detection model was used as a "*preprocessing step*", whose goal was to find and **mark** all erroneous tokens in a text sequence. After marking the target tokens, they were passed along to the correction model — with. optionally, context tokens appended to the left and right of the token — to be repaired.
This paper also studied the domain-specificity of the Post-OCR correction task further. In particular, there was an artificial data generation procedure explained, which was meant to use error statistics from wide-spread Post-OCR correction datasets to emulate the errors made in them and be able to generate an arbitrary amount of new data.

The two-step approach ended up yielding bad overall results when compared with the state-of-the-art models. The individual correction models initially showed a lot of promise, achieving a good % improvement on the sum of Levenshtein distances when ran *solely* on the error correction samples. Combined with the error detection model, however, the overall performance of the two-step approach was brought down into

the ground. The "Results" chapter explored the reasons for this, determining that the *bottleneck* of the approach to be the error detection model. Indeed, the latter model was too conservative in how it worked and thus missed a lot of erroneous targets in the test datasets, which ultimately left the error unchanged in the prediction texts, leading to a much worse performance.
Another observation that can be extracted from the paper is that Post-OCR correction is indeed **domain-specific**. Adding the artificial data to the final detection and correction models might have even *brought down* the results on the testing datasets. This proves that a more sophisticated is needed to bring on more data.

Other conclusions from the paper are that case sensitivity generally matters, especially when it comes to OCR error *detection*. Indeed, leaving the training data to contain a mix of lower- and uppercase letters helps the detection model train more easily, likely due to the fact that Post-OCR correction texts often feature **named entities**. If all letters were cast to be lowercased, the primary characteristic of named entities goes lost — their uppercase letters as a start.
Other than that, context was shown to be beneficial when using LSTM encoder-decoder networks, with most validation and test datasets showing a preference towards using more **preceding** tokens, rather than succeeding. This was, however, **not** the case with Transformer models, which were shown to work best in the *absence* of context — i.e., in **isolated** correction. The final isolated Transformer correction model just narrowly lost when it comes to average performance on the test datasets. Nonetheless, this shows that further research for Transformers for Post-OCR correction would be beneficial.

13 Future Work

Despite the underwhelming results of the two-step approach models, I believe there is still merit to them. The theory behind them is sound, even thought the implementation from this paper did not manage to showcase it well. In the rest of this chapter, I will suggest future improvement ideas, which I believe would make the two-step approach work better.

My first observation is about the data. As we first hinted at in chapter 2 about related work, and then confirmed with the final results in chapter 11, Post-OCR correction is *domain-specific*. In this paper, I tried to remedy this by creating artificial data based on error statistics from the datasets themselves, but ultimately failed short. Perhaps this technique would be more successful if the error statistics from the different datasets were not grouped together, but rather left separate. Then, the error statistics from each individual dataset could be used to create *smaller-sized* artificial datasets, which would be specialized in the errors from exactly one source. Another thing to consider is using additional external datasets, when generating artificial data. As discussed in the paper before, the "origins" of the different test datasets in these papers are radically different. Indeed, the "ACL Benchmark" dataset stems from the ACL Anthology Reference Corpus, which consists of OCR-ed texts from **research** papers. Compared to this, the ICDAR datasets feature OCR-ed texts from literature sources — books, newspapers, magazines, etc. These two groups of writing — researches and literature — differ when it comes to **jargon**, usage of **named entities**, use of **punctuation** and many more. Because of this, I think the

two-step approach (and all models for Post-OCR correction as a whole) would benefit if there were appropriate types of *clean datasets* (e.g., the arXiv document collection), with which more varied artificial data could be built.

My next suggestion would be to make use of a **subword** tokenizer for the error correction models, instead of a character-level vocabulary. This would, on one hand, make the pipeline process easier, as data used with the error detection model would be directly applicable to the correction models as well. On the other hand, I expect subword tokenization to also boost the performance of the overall pipeline — subwords maintain the flexibility of a character-level vocabulary (i.e., there is no chance of having an out-of-vocabulary error), while also bringing *context* (e.g., a correction model would be much more likely to correct *7hr* to *the* if it was a known, common word in its vocabulary).

The next idea is more of a *guideline*, rather than a suggestion. We have seen from the results chapter (refer to chapter 11) that the sequence-to-sequence correction models work relatively well when used on erroneous tokens that are actually detected as such by the detection model (see performance for *matched* samples in table 14). This leaves the error detection as the main **bottleneck**, which dictates whether the whole pipeline works well or not. Indeed, having a detection model with high *precision*, but a low *recall* value leaves too many erroneous samples in the OCR-ed texts. At the other end of the spectrum, having a low *precision* and a high *accuracy* value makes the model predict much more of the expected erroneous tokens from an OCR-ed text, at the cost of accumulating superfluous (i.e., **false positive**) samples as well. Indeed, this was the approach that was employed in [SN20], and is the approach which I would recommend to any future work on this task. Indeed, having the detection model have a high recall value could be "equalized" by having the error correction model train not exclusively on erroneous samples, but correct ones as well. Then, any superfluous predictions from the error detection model would have a lower chance of being modified by the correction model, which, as it stands, hurts the performance of

the two-step approach very much (refer to table 14). This way of work is just *safer* than the alternative, as there is no way to influence **missed** samples with the error correction model. Obviously, the golden middle of a high precision and recall values would be most preferred, but **recall** should be prioritized when a trade-off has to be made.

Amrhein et al. used an **ensemble** of correction models for their winning entry to the ICDAR2017 Post-OCR competition (refer to [AC18]). My penultimate suggestion is connected to this idea — it sounds very promising to evaluate the performance of an ensemble model, whose different models are trained on the different *types* of OCR mistakes (refer to explanation of those in 7.1.3). This would also help with making the combination with a **high-recall** detection model perform well, as one of the models in the ensemble could be trained to recognize correct, valid English words. Further, there could be different models in the ensemble based on the Levenshtein distance of mistakes, and also **word boundary** errors. The only problem with this suggestion is having to find enough data to train every model in the ensemble sufficiently well. As we saw in 9.1.1, this might not be easy to do with, as Post-OCR correction datasets tend to feature **big** imbalances between the different mistake types. This makes the suggestion of refining the artificial generation procedure even more important.

The last suggestion I have is connected to **named entities**. As discussed in numerous spots throughout the paper (refer to chapters 2 and 11), Post-OCR correction models have historically always had problems with named entities in text. At the same time, it is practically infeasible to use a NER model to detect named entities during the OCR correction process, as there is a high chance they feature errors, courtesy of the OCR system. Therefore, I think it would be very interesting to study the distribution of erroneous named entities in the Post-OCR correction task further. This could offer more insight into their general form when dealing with Post-OCR correction and potentially open up the way for a parallel model, which would reduce the amount of errors when it comes to named entities.

14 Acknowledgments

As always, I would like to first and foremost extend my warmest thank you to my fiancée. You gave me strength to keep going when it mattered, but also comfort and calmness when I needed to wind down. I truly couldn't have done this without you.
A very close second to her is my supervisor — Matthias Hertel. It was truly a pleasure working with you, and your contribution to this work can't be overstated. It is always important to have someone to share your ideas with and see if they make sense, and during our discussion it always felt like we were two co-students who were trying to figure out a problem. You created an environment, in which I could do my best, and I am very grateful for that.
Next, I want to thank my parents, who never let me forget that I have to be working on my master's degree and acted as a second supervisor, to whom I had to make weekly reports. The extra motivation and encouraging words came a long way for me to finish this work. If it weren't for my dad, the quality and writing time of this thesis might have been severely impacted on account of me almost short-circuiting my laptop by spilling milk.
I would also like to thank my work colleagues, who were gracious enough to read and give me feedback on the early drafts of this write out and also let me do mock presentations for them. The devil is in the details, and I couldn't have ironed out the wrinkles if it wasn't for them.
Finally, I would like to thank Lin-Manuel Miranda for creating the Alexander Hamilton musical, which gave me the energy to power through the late nights of writing. As

the main character with the same name, I hope I did not throw away my shot.

Bibliography

[Abr65] S. Abraham. Some questions of language theory. In *COLING 1965*, 1965.

[AC18] Chantal Amrhein and Simon Clematide. Supervised ocr error detection and correction using statistical and neural machine translation methods. *Journal for Language Technology and Computational Linguistics (JLCL)*, 33(1):49–76, 2018.

[ADT19] Yvonne Adesam, Dana Dannélls, and Nina Tahmasebi. Exploring the quality of the digital historical newspaper archive kubhist. In Costanza Navarretta, Manex Agirrezabal, and Bente Maegaard, editors, *Proceedings of the Digital Humanities in the Nordic Countries 4th Conference, Copenhagen, Denmark, March 5-8, 2019.*, volume 2364 of *CEUR Workshop Proceedings*, pages 9–17. CEUR-WS.org, 2019.

[Ass16] Mehdi Assefi. Ocr as a service: An experimental evaluation of google docs ocr, tesseract, abbyy finereader, and transym. *ISCV*, 12 2016.

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[BDD+08] Steven Bird, Robert Dale, Bonnie Dorr, Bryan Gibson, Mark Joseph, Min-Yen Kan, Dongwon Lee, Brett Powley, Dragomir Radev, and

Yee Fan Tan. The ACL Anthology reference corpus: A reference dataset for bibliographic research in computational linguistics. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, May 2008. European Language Resources Association (ELRA).

[Ben16] Eli Bendersky. The softmax function and its derivative, Oct 2016.

[BHM20] Hannah Bast, Matthias Hertel, and Mostafa M. Mohamed. Tokenization repair in the presence of spelling errors. *CoRR*, abs/2010.07878, 2020.

[BK17] Hannah Bast and Claudius Korzen. A benchmark and evaluation for text extraction from pdf. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–10, 2017.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[Bre01] Leo Breiman. *Machine Learning*, 45(1):5–32, 2001.

[Bri90] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[BVJS15] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. *CoRR*, abs/1506.03099, 2015.

[CDCM17] Guillaume Chiron, Antoine Doucet, Mickaël Coustaty, and Jean-Philippe Moreux. Icdar2017 competition on post-ocr text correction.

In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 01, pages 1423–1428, 2017.

[CMC+15] Maria Chatzou, Cedrik Magis, Jia-Ming Chang, Carsten Kemena, Giovanni Bussotti, Ionas Erb, and Cedric Notredame. Multiple sequence alignment modeling: methods and applications. *Briefings in Bioinformatics*, 17(6):1009–1023, 11 2015.

[Com10] Wikimedia Commons. A neural network with two layers., 2010.

[Com17] Wikimedia Commons. Recurrent neural network unfold, 2017.

[Cox58] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, December 1989.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, mar 1964.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[Den19] Timo Denk. Linear relationships in the transformer's positional encoding, Jan 2019.

[Edg04] Robert C Edgar. *BMC Bioinformatics*, 5(1):113, 2004.

[EP14] Paula Estrella and Pablo Paliza. Ocr correction of documents generated during argentina's national reorganization process. In *Proceedings of the First International Conference on Digital Access to Textual Cultural Heritage*, DATeCH '14, page 119–123, New York, NY, USA, 2014. Association for Computing Machinery.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

[Goo52] I. J. Good. Rational decisions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 14(1):107–114, 1952.

[HBFS01] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[Hea98] Marti A. Hearst. Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28, July 1998.

[Her19] Matthias Hertel. Neural language models for spelling correction. Master's thesis, University of Freiburg, Germany, 2019.

[HH19] Mika Hämäläinen and Simon Hengchen. From the paft to the fiiture: a fully automatic NMT and word embeddings method for OCR post-correction. *CoRR*, abs/1910.05535, 2019.

[HR78] David Harrison and Daniel L Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5(1):81–102, 1978.

[HR18] Jeremy Howard and Sebastian Ruder. Fine-tuned language models for text classification. *CoRR*, abs/1801.06146, 2018.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[HZZG19] Tianxing He, Jingzhao Zhang, Zhiming Zhou, and James R. Glass. Quantifying exposure bias for neural language generation. *CoRR*, abs/1905.10617, 2019.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KD16] Ido Kissos and Nachum Dershowitz. Ocr error correction using character correction and feature-based word classification. In *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*, pages 198–203, 2016.

[Koe05] Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of Machine Translation Summit X: Papers*, pages 79–86, Phuket, Thailand, September 13-15 2005.

[Lev66] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, feb 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[LNCPCA10] Rafael Llobet, José Navarro Cerdán, Juan-Carlos Perez-Cortes, and Joaquim Arlandis. Ocr post-processing using weighted finite-state transducers. pages 2021–2024, 08 2010.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[Mit80] Roger Mitton. Birkbeck spelling error corpus, 1980. Oxford Text Archive, `http://hdl.handle.net/20.500.12024/0643`.

[MLS13] Tomás Mikolov, Quoc V. Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *CoRR*, abs/1309.4168, 2013.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[MP17] Kendra Morgan and Merrilee Proffitt. Advancing the national digital platform: The state of digitization in us public and state libraries. `https://www.oclc.org/research/publications/2017/oclcresearch-advancing-national-digital-platform.html`, January 2017.

[MSA+11] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, null null, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin A. Nowak, and Erez Lieberman Aiden. Quantitative analysis of culture using millions of digitized books. *Science*, 331(6014):176–182, 2011.

[MV93] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, 1993.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, mar 2001.

[NJC+19] Thi-Tuyet-Hai Nguyen, Adam Jatowt, Mickael Coustaty, Nhu-Van Nguyen, and Antoine Doucet. Deep statistical analysis of ocr errors for effective post-ocr processing. In *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 29–38, 2019.

[NJCD21] Thi Tuyet Hai Nguyen, Adam Jatowt, Mickael Coustaty, and Antoine Doucet. Survey of post-ocr processing approaches. *ACM Comput. Surv.*, 54(6), jul 2021.

[NJN+20] Thi Tuyet Hai Nguyen, Adam Jatowt, Nhu-Van Nguyen, Mickael Coustaty, and Antoine Doucet. *Neural Machine Translation with BERT for Post-OCR Error Detection and Correction*, page 333–336. Association for Computing Machinery, New York, NY, USA, 2020.

[NOMC11] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, September 2011.

[Nor09] Peter Norvig. Natural language corpus data. In T. Segaran and J. Hammerbacher, editors, *Beautiful Data: The Stories Behind Elegant Data Solutions*, Theory in practice. O'Reilly Media, 2009.

[NvdHT17] Gerhard-Jan Nauta, Wietske van den Heuvel, and Stephanie Teunisse. Europeana dsi 2—access to digital resources of european heritage. `https://pro.europeana.eu/page/enumerate`, August 2017.

[NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48 3:443–53, 1970.

[Ola15] Christopher Olah. Understanding lstm networks, Aug 2015.

[PMB12] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[PNI+18] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018.

[PRS19] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. *CoRR*, abs/1903.05987, 2019.

[RDCM19] Christophe Rigaud, Antoine Doucet, Mickaël Coustaty, and Jean-Philippe Moreux. Icdar 2019 competition on post-ocr text correction. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 1588–1593, 2019.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[RN18] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[SC19] David A. Smith and Ryan Cordell. A research agenda for historical and multilingual optical character recognition. 2019.

[Sel74] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[SHB15] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015.

[She20] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404:132306, mar 2020.

[SN12] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152, 2012.

[SN20] Robin Schaefer and Clemens Neudecker. A two-step approach for automatic ocr post-correction. In *LATECHCLFL*, 2020.

[SQXH19] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune BERT for text classification? *CoRR*, abs/1905.05583, 2019.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[TE96] Xiang Tong and David Andreoff Evans. A statistical approach to automatic ocr error correction in context. In *VLC@COLING*, 1996.

[TMR09] Simon Tanner, Trevor Muñoz, and Hemy Ros. Measuring mass text digitization quality and usefulness: Lessons learned from assessing the ocr accuracy of the british library's 19th century online newspaper archive. *D-lib Magazine - DLIB*, 15, 07 2009.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[Wer90] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[WP98] Ronald Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2, 09 1998.

List of Figures

List of Tables

Listings