Master's Thesis

Automatic Correction of Misaligned Spaces and Typos Using Deep Learning

---

Mostafa M. Mohamed

Examiners: Prof. Dr. Hannah Bast &
Prof. Dr. Joschka Bödecker
Adviser: Prof. Dr. Hannah Bast

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures
June 14th, 2018

Writing period

15. 12. 2017 – 14. 06. 2018

Examiners

Prof. Dr. Hannah Bast

Prof. Dr. Joschka Bödecker

Adviser

Prof. Dr. Hannah Bast

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

Large text corpora could get corrupted for a variety of reasons, such corruptions could be tokenization corruptions, like removing correct spaces between words, adding wrong spaces in-between words, splitting words across lines or garbling some characters in words. These mistakes could make it hard for systems, that process large text corpora, to identify the words within a text correctly, and as a result, lose a lot of information about the text. This thesis developed a model which utilizes character-based language models, that are trained as recurrent neural networks via deep learning. The model automatically detects the tokenization mistakes in a given text and fixes them accordingly. The model can successfully classify, and hence fix, the tokenization errors with a mean F_1-score that, depending on the model's specifications, training settings, and evaluation settings, varies between to 81% to 96%. Furthermore, two non-learning dictionary-based approaches are presented, using a baseline greedy strategy and dynamic programming, in addition to an end-to-end deep learning approach and a baseline learning approach using 3-Gram Markov models. The models are trained and evaluated using two datasets: Simple-Wikipedia and Reuters-21578. The presented approaches are empirically evaluated to demonstrate the superior performance of the presented model. Additionally, it is experimented using different RNN architectures, RNN history length, model size, beam size in Beam-search, training data size, tokenization errors and some internal features.

Zusammenfassung

Korpora, die viel Text enthalten könnten durch verschiedene Gründe korrumpiert werden. Solche Korruptionen können Tokenisierungskorruption sein, wie z.B. die Entfernung korrekter Leerzeichen zwischen Wörtern, das Hinzufügen falscher Leerzeichen zwischen Wörtern, die Aufteilung Wörtern auf verschiedene Zeilen oder die Korruption von Zeichen innerhalb von Wörtern. Solche Fehler können für Systeme, die große Korpora verarbeiten, dazu führen, dass es für die Systeme schwer wird Wörter korrekt zu identifizieren, was zu Informationsverlust über den Text führt. In dieser Arbeit wurde ein Modell entwickelt, das Zeichen-basierte Sprachmodelle verwendet, die durch tiefes Lernen mittels rekurrenten neuronalen Netzwerken trainiert wurden. Dieses Modell findet Tokenisierungsfehler innerhalb eines Textes und repariert diese Tokenisierungsfehler mit einem durchschnittlichen F_1-Score, der zwischen 81% und 96% variiert, erfolgreich. Außerdem wurden zwei Wörterbuch-basierte Ansätze vorgestellt, die ein Greedy-Verfahren und dynamische Programmierung verwenden. Zusätzlich wurde ein End-zu-End-Lernverfahren und, als Baseline, ein 3-Gram Markov-Modell entwickelt. Alle vorgestellten Modelle wurden auf zwei Datensätzen, Simple-Wikipedia und Reuters-21578, trainiert und evaluiert. Die vorgestellten Verfahren wurden empirisch evaluiert und gegenübergestellt um deren Leistung zu vergleichen. Außerdem wurden verschiedene RNN Architekturen mit Variationen der Hyperparametern wie der Sequenzlänge, der Modelgröße, der Suchtiefe der Beamsuche, der Größe des Trainingsdatensatzes, verschiedene Tokenisierungsfehler und anderen internen Merkmalen untersucht.

Acknowledgments

First and foremost, I would like to thank each of ...

- Prof. Dr. Hannah Bast for choosing such an interesting topic, for giving me the opportunity to work with her, for the guidance and crucial advice throughout my thesis, and making sufficient resources at my disposal when needed.
- MSc. Markus Näther for his continuous support and help on a theoretical level regarding deep learning, and on a technical level regarding tips and debugging tools, and for maintaining RiseML which I used to run all the GPU training and for proofreading my thesis.
- Frank Dal-Ri for responsiveness regarding the clusters I used to evaluate the results of my experiments, and all related technical issues.
- MSc. Maged Shalaby for his suggestions about datasets and proofreading my thesis.
- My family for their love, support and encouragement, that lead me where I am today.
- My girlfriend Emese Pálffy for her support and patience with me throughout my thesis.
- My friends Omar Kassem and Mina Nessiem for listening to my achievements throughout my thesis and their proofread of my thesis.
- My friend Hatem Elshatlawy for being a supportive friend since I came Freiburg.

Contents

1 Introduction 1

2 Related work 3

3 Background 5
- 3.1 Preliminaries and notations . . . 5
- 3.2 Dynamic programming review . . . 7
  - 3.2.1 Edit distance . . . 8
- 3.3 Trie dictionary . . . 18
- 3.4 Neural network review . . . 19
  - 3.4.1 Multi-class classification . . . 22
  - 3.4.2 Class sampling . . . 23
  - 3.4.3 Dropouts . . . 23
- 3.5 Recurrent neural networks review . . . 23
  - 3.5.1 RNN cells: Simple, LSTM, GRU . . . 24
  - 3.5.2 Character-based language model . . . 26

4 Datasets 31
- 4.1 Simple-Wikipedia dataset . . . 31
- 4.2 Reuters-21578 news dataset . . . 33
- 4.3 Corruptions . . . 33

5 Problem definition 37
- 5.1 Fixing evaluation definition . . . 37

6 Dictionary-based approaches 41
- 6.1 Greedy based approach . . . 41
- 6.2 Dynamic programming based approach . . . 44
  - 6.2.1 Token scoring . . . 44
  - 6.2.2 Retokenization . . . 46
  - 6.2.3 Grouping . . . 47

7 Learning-based approaches 53
7.1 Maximum likelihood sequence estimation . . . 53
7.1.1 Beam search . . . 55
7.1.2 State space . . . 56
7.1.3 State updates . . . 57
7.2 Input processing . . . 59
7.2.1 Input format for RNN models . . . 59
7.2.2 Input perturbation . . . 60
7.2.3 Edit alignments . . . 61
7.3 Bicontext model . . . 62
7.3.1 Long looking and occurrence functions . . . 62
7.3.2 Fixing operations . . . 65
7.3.3 Decisions tuner . . . 67
7.3.4 Look-forward in additions . . . 70
7.3.5 Wrapping up . . . 72
7.4 Baseline 3-Gram Markov model . . . 74
7.5 End-to-end model . . . 74
7.6 Utility . . . 75
7.6.1 Caching . . . 75

8 Experiments 79
8.1 Experiments specifications . . . 79
8.2 Character-based language models evaluation . . . 82
8.3 Fixing evaluations . . . 84

9 Conclusion and future work 91
9.1 Summary and conclusions . . . 91
9.2 Future work . . . 93

Bibliography 94

List of Figures

1 Trie data-structure . . . 19
2 Neural network . . . 20
3 Unrolled many-to-one RNN . . . 29
4 Corruption and fixing operations diagram . . . 39
5 Fixing actions updates . . . 58
6 Edit alignment example . . . 61

List of Tables

1 Edit distance table example 14

2 Datasets summary 32

3 Tokens Scoring 46
4 Retokenization values 47
5 DP approach example 50
6 DP approach example, tables 52

7 Bicontext approach snapshots 73

8 Simple-Wikipedia train/test losses 83
9 Reuters-21578 dataset train/test losses 88
10 Simple-Wikipedia fixing evaluations 89
11 Reuters-21578 fixing evaluations 89
12 Different features evaluation 90

List of Algorithms

1 Edit distance Tabular method . . . 10
2 Edit distance Tabular method . . . 11
3 Edit distance Tabular method, with dimension compression . . . 12
4 Edit distance traceback construction . . . 13
5 Trie word insertion . . . 20
6 Trie modified search . . . 21

7 Corruptor . . . 35

8 Greedy Trie traversal . . . 42
9 Greedy approach . . . 43
10 Retokenization . . . 48
11 Dynamic Programming approach . . . 51

12 Beam search . . . 54
13 Update(S) function in the beam search algorithm . . . 59
14 Bicontext tuned transition function $\delta(S)$ . . . 71
15 Bicontext approach . . . 72
16 Caching data structure . . . 77

1 Introduction

One of the main mediums used for communication is texts, there are gigantic datasets of text like the World Wide Web and Wikipedia. Texts exist in a variety of digital forms, it can be in text files, documents, portable document format (PDF) files, web pages and pictures. Having the text stored in some editable form gives more freedom for using text, searching through it, copying it, correcting mistakes, adding comments and making improvements. Consequently, there are methods for transformation of a variety of sources into editable text files, and other methods are used to preprocess text corpora and other methods for searching over text. Such methods could face difficulties when dealing with corrupt texts, like in our case, texts with badly tokenized words, or texts with spelling mistakes. Such issues could occur with text sources that don't have well-formulated text, for example, an image containing text could be blurry and hence having unclear characters that get parsed wrongly, leading to a corrupt version of the actual text. Also, a PDF parser/viewer could render the spaces wrongly, and as a result, the parsed words could have misallocated spaces and hence misalignment of words [1].

One class of errors that will be addressed in this thesis is mainly tokenization errors in word-based languages (like English or French, but not Chinese). A variety of systems and algorithms depend that a given text consists of individual tokens where each token is a correct word in the language. Based on that a variety of processing methods can be used on the data depending on the targeted application, for example, word search and part-of-speech tagging depend on this [2]. Tokenization errors are classified as one of four types:

1. Missing spaces (like 'HelloWorld')
2. Adding wrong spaces (like 'Hello Wor ld')
3. Line break hyphenation (like 'Hello Wo-[newline]rld')
4. Garbled characters (like 'Hello Warld')

The main issue addressed in this thesis is automatically fixing such tokenization mistakes in a given text corpus. With the rise of machine learning and specifically

deep learning, there are a variety of models that could gain knowledge about texts and predict a variety of features about them, one of the famous such examples is word2vec [3], which generates n-dimensional vectors that correspond to languages' words. This drives a motivation for using such models to fix corrupt texts according to language models, which could get better results than other traditional methodologies.

In deep learning, there are character-based and word-based language models. The significance of each model strongly depends on the application, for example, the word-based language model is used in neural machine translation as in [4]. However, the word-based model already assumes that the words are well-defined entities, which is not the case in the given problem. On the other hand, the character-based models get an intuition about the characters distribution in a corpus and hence can generate sample text that "looks" very similar to texts in the training corpus [5]. The knowledge of the character-based model regarding the characters distribution within text derived the motivation to solve the addressed problem using character-based models. The main presented approach by this thesis is a deep learning approach that utilizes character-based language models to solve the tokenization errors. Additionally, in order to have an insight about the performance of the proposed approach, which is based on deep learning, two other learning approaches are presented, using end-to-end deep learning and 3-Gram Markov models. Furthermore, two non-learning dictionary-based approaches, using greedy and dynamic programming strategies, are also proposed in order to compare the learning approaches against them.

The thesis is outlined as chapters: 'Related work' where I briefly talk about other work that solve a similar problem or using similar techniques to solve other problems, 'Background' where I list the notations that will be used throughout the thesis and review some common background of algorithms and models like dynamic programming and neural networks. After that, I elaborate on the datasets I used and how I preprocessed them in the 'Datasets' chapter, followed by a short chapter 'Problem definition' which contains a formal definition of the problem and how it will be evaluated. Followingly, the dictionary-based approaches are presented in the 'Dictionary-based approaches' chapter. The learning approaches, including the main approach (bicontext deep learning approach) are then presented in the chapter 'Learning-based approaches', followed by an empirical evaluation chapter 'experiments'. Finally, the last chapter is 'Conclusion and Future work' where I summarize the work of the thesis, the results and talk about ideas for later improvements and shed the light on some derivative problems from the presented problem, that might be solved using the presented approach.

2 Related work

The work done by Déjean et. al. (2006) [1] builds a system that transform a given PDF file into an XML representation. They encountered tokenization problems while parsing PDFs, more particularly because of spaces, which is addressed by the first two types of tokenization errors mentioned in the introduction chapter. One of the reasons for the occurrence of these errors was the difference in glyphs' (character-representation) widths and the spacings between them. They attempted to solve this problem by using a dictionary-based method and applying Viterbi algorithm. By trying all possible retokenizations of the text, in attempt to match all the words.

Wen et. al. (2003) [6] developed an Hidden Markov Model (HMM) model which identifies a token depending on the type of information it contains, like name, email address, location, etc. Their model, however, uses a word-based language model, in order to label tokens depending on the labels of neighboring tokens. This problem would be even harder if the given text is not correctly tokenized due to the presented mistakes.

Graves (2013) [5] developed character-based language models using RNNs, and how to use it to generate fake text, which is a text that looks like a real text but doesn't have a particular meaning. This motivated deep learning to solve a variety of text related problems. Some of the relevant work are solving syntactical or spelling mistakes in a given text using deep learning. However, they are solving different types of errors and not tokenization errors. Gupta et. al. (2017) [7] developed DeepFix which tries to solve syntactical errors in C programs, using an end-to-end model based on sequence-to-sequence with attention. Ghosh et. al. (2017) [8] developed a sophisticated deep learning model, which combines convolutional layers, character-based layers and word-based embedding layers in order to fix spelling mistakes that are likely to happen during writing on a keyboard. Sakaguchi et. al. (2017) [9] developed a deep learning model which is hybrid between characters and words, which identifies words with reshuffled characters (except the first and last characters) and fixes them.

Chollampatt et. al. (2016) [10, 11] developed two models for fixing grammatical

errors in a given text. Xie et. al. (2016) [12] developed a character-based language model with attention that suggests better expressions for new language learners; these are another forms of mistakes or corruptions that are being fixed using deep learning models.

The approaches that are presented in this thesis are either based on dictionary methods like [1] or deep learning like the other mentioned work. The main presented approach was motivated by the character-based language model presented by [5], with an enhancement of another character model that generates characters in a backward direction.

3 Background

This chapter contains the definitions and notations that will be used throughout the thesis, and it will review some topics, on which I will be building my models and algorithms later on. The topics include dynamic programming, Trie data structure, neural networks, recurrent neural networks (RNNs) and fake text generation using RNNs, in addition to some topics/terminologies used in deep learning.

3.1 Preliminaries and notations

In order to be able to define the problem formally, a set of definitions and notations is proposed, which will be used throughout the thesis.

- Partitioned alphabet $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$: An ordered set, which is a union of disjoint sets of characters that are used to form the texts. Mainly, a union of digits set, English alphabet set and special characters set.
- Delimiters Γ: A set of special characters that are used as delimiters or separators to tokenize texts, such that $\Sigma \cap \Gamma = \phi$. Mainly, spaces and newlines.
- Padding character $\dot{\$}$: A special character that is used to fill in padding/empty parts of strings, such that $\dot{\$} \notin \Sigma \cup \Gamma$.
- Unknown character Ψ: A special character denoting an unknown character, such that $\Psi \notin \Sigma \cup \Gamma \cup \{\dot{\$}\}$.
- Character-set Λ: The universal alphabet set of all characters that will be used in our models. In other words, $\Lambda = \Sigma \cup \Gamma \cup \{\Psi, \dot{\$}\}$.
- Text T: A sequence of characters from Λ, or in regular expression terms $T \in \Lambda^*$.
- Empty text ε: An empty sequence of characters.
- Substring $T_{i \to j}$: A string that consists of the characters $T_i, \cdots, T_{j-1}$, additionally $T_{i \twoheadrightarrow j}$ is the string that consists of $T_i, \cdots, T_j$.

- Prefix $T_{\rightarrow i}$: A string or sequence that consists of the elements $T_1, \cdots, T_{i-1}$, also the prefix $T_{\twoheadrightarrow i}$ is the sequence that consists of $T_1, \cdots, T_i$.
- Suffix $T_{i\rightarrow}$: A string or sequence that consists of the characters $T_i, \cdots, T_{|T|}$; the suffix will also be denoted by $T_{-i\rightarrow}$ which is the sequence $T_{|T|-i+1}, \cdots, T_{|T|}$.
- Edit operation: An operation that can be applied on a string T that changes the string. Such as:
  1. (ADD, i, s): adds the character s before the i-th character in string T.
  2. (DEL, i, T_i): deletes the i-th character in string T.
  3. (CHG, i, s): changes the i-th character in string T to s.
- Copy (No-change) operation: (NOP, i, T_i) an ineffective edit operation that copies the i-th character in the string T as it is. This might also be referred to as (CPY, i, T_i) depending on the context. It is called ineffective, because it doesn't change the text.
- Corruption operation: An edit operation that is used to transform a correct text into a corrupt one.
- Fixing operation: An edit operation that attempts to fix a corrupt text back into a correct text.
- Token: A pair (w, sp) of a word $w \in \Sigma_s^+$ and delimiters split $sp \in \Gamma^*$. Σ_s is one of the partitions of Σ.
- Concatenation $\circ$: A binary operator between two strings or sequences A and B such that $A \circ B = A_1, A_2, \cdots, A_{|A|}, B_1, B_2, \cdots, B_{|B|}$
- Tokenization of a text T: A sequence of tokens $Q_1, \cdots, Q_r$ such that the words and splits are maximals, and the concatenation $Q_1.w \circ\ Q_1.sp\ \circ Q_2.w\ \circ \cdots \circ\ Q_r.sp = T$.
- One-hot vector $\{c = 1\}_S$: A unit vector of length $|S|$ over an ordered set S. If c has an index i in the set S, then i-th element of the vector is 1 and the remaining are zeros.
- Smoothing ϵ: A small decimal number $\approx 10^{-30}$ or 10^{-10} that is used to smooth formulas that could have undefined computations. Like, $\frac{a}{b} \mapsto \frac{a}{b+\epsilon}$, $\frac{a}{b} \mapsto \frac{a+\epsilon}{b+\epsilon}$ or $\log\{x\} \mapsto \log\{x + \epsilon\}$.

- Discrete probability distribution $\mathbf{p}_S$: A vector of size $|S|$, corresponding to an ordered set S. Such that for all i $0 \leq \mathbf{p}_i \leq 1$ and $\sum_{i=1}^{n} \mathbf{p}_i = 1$. The value $\mathbf{p}_i$ corresponds to the probability of the i-th element to participate in some event.

- Tensor $\mathbf{Q}$: n-dimensional array of numbers, often integers or real numbers.

- Boolean-to-integer notation $[x]$: An integer value, which is equal to 1 if the boolean condition x is true, and 0 otherwise. This is also applicable on any tensor by using this operator on all the elements point-wisely. For example, given a matrix R, then $[R > 3]$ is a matrix M where $M_{i,j} = [R_{i,j} > 3]$ for all i, j.

- Enumerator or Generator: A function that returns a collection of items on demand, the items are returned using a **yield** statement, which returns an object and pauses; upon the request of the next object, the function continues until it yields the next element and pauses. This goes on until all the elements are yielded.

- Element-wise product $\odot$: An operation between two tensors, that multiplies each element of the two given tensors point-wisely. For example, given two matrices A and B, then $(A \odot B)_{ij} = A_{ij}B_{ij}$ for all i, j.

3.2 Dynamic programming review

In this section, I will review dynamic programming because it will be essential to construct one of the dictionary-based approaches. The most suitable example for this is the edit distance algorithm because it will be used later on for the evaluation of the performance of my approaches, and also it will be used to construct training data for some of the approaches.

Dynamic programming is a well-known technique for solving a variety of problems like optimization problems and combinatorial counting problems. It mainly depends on the divide-and-conquer strategy that breaks down a given problem into a set of subproblems that are simpler yet have the same underlying structure, then solves each of the subproblems recursively and combine their solutions to solve the given problem. Furthermore, dynamic programming makes use of overlapping subproblems by caching the already known solutions of subproblems in order to avoid recomputing the solutions for the same subproblems many times, which results in optimizing a variety of algorithms from an exponential time complexity into a polynomial time

complexity. The general structure of dynamic programming solutions, similar to how it is defined in [13], consists of the following generic steps:

1. Define any subproblem with a simple state, this also includes the main given problem.
2. Divide and conquer the problem, by defining a recursive relationship between the problem's state and subproblems' states.
3. Formulate a recursion that solves the complex problem by combining the solutions of the simpler subproblems.
4. Compute the recursive formula while caching the computed results, in order not to recompute overlapping subproblems.
5. Compute the solution from the computed information.

The caching in step 4 is done in one of two ways, either having a table and iterating over it in a bottom-up approach (i.e. from the simple subproblems to the more complex ones), or having a lookup table and recursively enumerate through the problems in a top-down approach (i.e. from the complex problem to the simpler subproblems) while keeping track if a subproblem is already computed or not in order to avoid recomputation. These methods will be demonstrated in the edit distance algorithm [14].

3.2.1 Edit distance

Given two strings T and S, we are asked to transform T to S by using a minimal sequence of edit operations. For example, the string 'heloworxld' can be changed into 'hello world' by using 3 edit operations $(ADD, 3, \text{`l'})$, $(ADD, 5, \text{` '})$ and $(DEL, 8, \text{`x'})$.

The edit distance algorithm finds such minimal sequence that transforms T to S. An additional note, as seen in the example, the closer the edit distance to 0, the more similar the strings T and S are.

Any instance of a subproblem is formulated by a pair (i, j) which denotes the prefixes $T_{1 \twoheadrightarrow i}, S_{1 \twoheadrightarrow j}$, where $0 \leq i \leq |T|$ and $0 \leq j \leq |S|$. The instance (i, j) is related to the subproblems $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$ by the operations DEL, ADD and NOP/CHG respectively. Then we have a recursive formula that solves the

subproblem (i,j) by using the solutions of the 3 subproblems using the formula:

$$edit[i,j] = \min \begin{cases} edit[i-1,j]+1 & i>0\text{, by deleting } T_i \\ edit[i,j-1]+1 & j>0\text{, by adding } S_j \text{ between } T_i, T_{i+1} \\ edit[i-1,j-1]+1 & i,j>0 \text{ and } T_i \neq S_j\text{, by changing } T_i \rightarrow S_j \\ edit[i-1,j-1] & i,j>0 \text{ and } T_i = S_j\text{, by copying } T_i \\ 0 & i=j=0 \end{cases} \quad (1)$$

Algorithm 1 shows how to compute the edit distance, according to equation 1, in a tabular bottom-up fashion. The execution goes row-by-row from the smaller rows to the higher rows, ensuring that when $edit[i,j]$ is being computed, all the 3 needed subproblems are already computed. The other approach for computation is using recursion with memoization [13] in a top-down approach. The recursive function in the top-down approach checks at the beginning if the global memoization lookup table already has the computed solution for the given subproblem. Consequently, it just returns the cached value if it exists, otherwise it computes the value recursively and stores it in the lookup table then returns it. This is depicted in algorithm 2.

Both algorithms have the same time and space complexities given by $\mathcal{O}(|T||S|)$, because they both have a state space of that size and each element in it is computed only one time by making $\mathcal{O}(1)$ steps, by checking values of a constant number of subproblems. However, the top-down approach in practice can be slightly slower by some small factor, because the recursion execution could have an overhead of how it is internally computed. One of the advantages of the top-down approach is that, it is more intuitive to understand, and we don't need to care about the order of execution because it's automatically handled by the recursion. It simply uses recursion and augments it with a caching structure. This idea will be useful in the main approach in section 7.3.

Algorithm 3 shows how to compute the edit distance in a bottom-up approach like algorithm 1, but with making use of the dimension compression technique [15], which reduces the space complexity of the lookup table from $\mathcal{O}(|T||S|)$ to $\mathcal{O}(|S|)$. This technique operates on the key observation that the bottom-up approach is using only 2 rows at any given time (the current row and previous row), therefore it compresses the dimension of the rows to only two rows, and uses them interchangeably. This is achieved by using the $\bmod 2$ operation, when accessing the table using index i, because $i \bmod 2$ points at the current row, and $(i-1) \bmod 2$ (which is equal to $(i$

Algorithm 1 Edit distance Tabular method

```
function EDIT-DISTANCE(T, S)
    edit := Table((|T| + 1, |S| + 1), ∞)
    operation := Table((|T| + 1, |S| + 1), nil)
    foreach i ∈ {0, 1, ··· , |T|} do
        foreach j ∈ {0, 1, ··· , |S|} do
            if i = j = 0 then
                edit[i, j] := 0
            end if
            if i, j > 0 and edit[i, j] > edit[i − 1, j − 1] and T_i = S_j then
                edit[i, j] := edit[i − 1, j − 1]
                operation[i, j] := NOP
            end if
                        ▷ Remove this block if CHG operations are not allowed.
            if i, j > 0 and edit[i, j] > edit[i − 1, j − 1] + 1 and T_i ≠ S_j then
                edit[i, j] := edit[i − 1, j − 1] + 1
                operation[i, j] := CHG
            end if
            if j > 0 and edit[i, j] > edit[i, j − 1] + 1 then
                edit[i, j] := edit[i, j − 1] + 1
                operation[i, j] := ADD
            end if
            if i > 0 and edit[i, j] > edit[i − 1, j] + 1 then
                edit[i, j] := edit[i − 1, j] + 1
                operation[i, j] := DEL
            end if
        end for
    end for
    return operation
end function
```

Algorithm 2 Edit distance Tabular method

```
function EDIT-DISTANCE(T, S)
    edit := Table((|T| + 1, |S| + 1), ∞)
    operation := Table((|T| + 1, |S| + 1), nil)
    procedure EDITDISTFUNC(i, j)
        if i = j = 0 then
            edit[i, j] := 0
            return edit[i, j]
        end if
        if operation[i, j] ≠ nil then
            return edit[i, j]                    ▷ The result was computed earlier
        end if
        edit[i, j] := ∞
        if i, j > 0 and edit[i, j] > EDITDISTFUNC(i − 1, j − 1) and T_i = S_j then
            edit[i, j] := EDITDISTFUNC(i − 1, j − 1)
            operation[i, j] := NOP
        end if
                              ▷ Remove this block if CHG operations are not allowed.
        if i, j > 0 and edit[i, j] > EDITDISTFUNC(i − 1, j − 1) + 1 and T_i ≠ S_j then
            edit[i, j] := EDITDISTFUNC(i − 1, j − 1) + 1
            operation[i, j] := CHG
        end if
        if j > 0 and edit[i, j] > EDITDISTFUNC(i, j − 1) + 1 then
            edit[i, j] := EDITDISTFUNC(i, j − 1) + 1
            operation[i, j] := ADD
        end if
        if i > 0 and edit[i, j] > EDITDISTFUNC(i − 1, j) + 1 then
            edit[i, j] := EDITDISTFUNC(i − 1, j) + 1
            operation[i, j] := DEL
        end if
        return edit[i, j]
    end procedure
    EDITDISTFUNC(|T|, |S|)         ▷ Calling the procedure, to fill the lookup table
    return operation
end function
```

Algorithm 3 Edit distance Tabular method, with dimension compression

```
function EDIT-DISTANCE(T, S)
    edit := Table((2, |S| + 1), ∞)
    operation := Table((|T| + 1, |S| + 1), nil)
    foreach i ∈ {0, 1, ··· , |T|} do
        i_2 := i  mod 2
        foreach j ∈ {0, 1, ··· , |S|} do
            edit[i_2, j] := ∞
            if i = j = 0 then
                edit[i_2, j] := 0
            end if
            if i, j > 0 and edit[i_2, j] > edit[i_2 ⊕ 1, j − 1] and T_i = S_j then
                edit[i_2, j] := edit[i_2 ⊕ 1, j − 1]
                operation[i, j] := NOP
            end if
                                    ▷ Remove this block if CHG operations are not allowed.
            if i, j > 0 and edit[i_2, j] > edit[i_2 ⊕ 1, j − 1] + 1 and T_i ≠ S_j then
                edit[i_2, j] := edit[i_2 ⊕ 1, j − 1] + 1
                operation[i, j] := CHG
            end if
            if j > 0 and edit[i_2, j] > edit[i_2, j − 1] + 1 then
                edit[i_2, j] := edit[i_2, j − 1] + 1
                operation[i, j] := ADD
            end if
            if i > 0 and edit[i_2, j] > edit[i_2 ⊕ 1, j] + 1 then
                edit[i_2, j] := edit[i_2 ⊕ 1, j] + 1
                operation[i, j] := DEL
            end if
        end for
    end for
    return operation
end function
```

Algorithm 4 Edit distance traceback construction

```
function EditOperations(T, S, detailed = True)
    operation := Edit-distance(T, S)
    s_0 := (|T|, |S|)
    ops := [ ]
    t := 0
    while s_t ≠ (0, 0) do
        t := t − 1
        i, j := s_{t+1}
        if operation[i, j] = NOP then                                   ▷ T_i = S_j
            if detailed then
                ops.push-front((i, j, (NOP, i, T_i)))                   ▷ R_t pushed
            end if
            s_t := (i − 1, j − 1)
        end if
        if operation[i, j] = CHG then
            if detailed then
                ops.push-front((i, j, (CHG, i, S_j)))                   ▷ R_t pushed
            else
                ops.push-front((CHG, i, S_j))
            end if
            s_t := (i − 1, j − 1)
        end if
        if operation[i, j] = ADD then
            if detailed then
                ops.push-front((i + 1, j, (ADD, i + 1, S_j)))           ▷ R_t pushed
            else
                ops.push-front((ADD, i + 1, S_j))
            end if
            s_t := (i, j − 1)
        end if
        if operation[i, j] = DEL then
            if detailed then
                ops.push-front((i, j + 1, (DEL, i, T_i)))               ▷ R_t pushed
            else
                ops.push-front((DEL, i, T_i))
            end if
            s_t := (i − 1, j)
        end if
    end while
    return ops                                  ▷ ops = R_{−m}, · · · , R_{−2}, R_{−1}
end function
```

mod 2) $\oplus$ 1, where $\oplus$ is the bit-wise xor operation) points at the previous row, and then they are automatically interchanged. In practice, the table operation can be stored using a smaller data type (1 byte instead of 4), because it has 4 possible values. As a result, the memory complexity is still the same (because of the *operation* table), but there is a factor of saved memory, from $8k|T||S|$ to $k|T||S| + 8k|S|$ bytes (nearly saving a factor of 8), for some constant number k. This technique is only useable with the bottom-up approach, which gives the bottom-up approach an advantage. This saved factor was helpful during my evaluations, especially when they were ran using multiple processes (between 4 and 24) simultaneously with a shared RAM memory.

All the algorithms 1, 2 or 3 have tie breaking rules which prefer NOP, CHG, ADD then DEL respectively, by checking if the subproblem's solution value is strictly less then the current $edit[i][j]$. An operation is only assumed as optimal if it makes additional improvement over the operation before it. Additionally, we can make the CHG operations not allowed by removing the corresponding block from the algorithm as shown by the comments, because in the evaluation metrics, the CHG operations will not be used.

Algorithm 4 shows how to compute the actual edit operations from string T to string S given the *operation* table computed in the algorithms 1, 2 or 3. The output can be undetailed, which outputs a set of edit operations from T to S, that are not ineffective (so mainly ADD, CHG or DEL), which shows the main differences between the two texts. The output can also be detailed which is explained next.

	j	0	1	2	3	4	5
i		ε	a	b	x	c	h
0	ε	0	$\leftarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5
1	a	$\uparrow$ 1	$\nwarrow$ 0	$\leftarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4
2	x	$\uparrow$ 2	$\uparrow$ 1	$\leftarrow$ 2	$\nwarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3
3	b	$\uparrow$ 3	$\uparrow$ 2	$\nwarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4
4	c	$\uparrow$ 4	$\uparrow$ 3	$\uparrow$ 2	$\leftarrow$ 3	$\nwarrow$ 2	$\leftarrow$ 3

Table 1: Table of the values of $edit[i][j]$, and the arrows show which subproblems are used to compute the given values. The arrows are determined by the values $operation[i][j]$.

An example of the computed tables is demonstrated in table 1, where T = 'abxc', S = 'axbch', $\uparrow$ denotes $operation[i][j]$ = DEL, $\nwarrow$ denotes $operation[i][j]$ = NOP, $\leftarrow$ denotes $operation[i][j]$ = ADD and CHG operations are not used.

Detailed edit operations

The detailed version of edit operations construction, has as output, a sequence of triples if executed synchronously on the string T, then they would generate the string S. The triples are on the form (i, j, op), where they denote that the first edit operation to transform the suffix $T_{i\to}$ to $S_{j\to}$ is to use the edit operation op. The claim that these triples denote that op is the first edit operation that transforms the suffix $T_{i\to}$ to suffix $S_{j\to}$, is the basis for edit alignments in subsection 7.2.3, and therefore I will prove this claim here, the proof will be done by mathematical induction, but first, few definitions need to be introduced. First, we define R_t as the element inserted at time point t in the triples list (where t enumerates through $0, -1, \cdots, -m$, and m is the length of the returned result), so the output of the algorithm is the list $R_{-m}, \cdots, R_{-1}$. We define $R_t := (x_t, y_t, op_t)$, and also $R_0 = (|T| + 1, |S| + 1, nil)$ and $R_{-m-1} = (0, 0, nil)$ for completeness, but R_0 and R_{-m-1} will not be pushed by the algorithm. Additionally, we annotate $s_t = (i_t, j_t)$ which are the indices of the traceback path (marked in grey in the example, in table 1). As observed in algorithm 4, for all t whenever $R_t = (x_t, y_t, op_t)$ is pushed in the resulting operations, s_t is set to the value $(x_t - 1, y_t - 1)$ 2-3 lines after that, thus we have for all t the equations $i_t = x_t - 1, j_t = y_t - 1$. If we consider when R_t is pushed (in one of the four marked lines in algorithm 4), we find out that the values of all pairs (x_t, y_t), for all t, are given by:

$$(x_t, y_t) = \begin{cases} (i_{t+1}, j_{t+1}) = (x_{t+1} - 1, y_{t+1} - 1) & \text{if } op_t = (\text{NOP}, x_t, S_{y_t}) \\ (i_{t+1}, j_{t+1}) = (x_{t+1} - 1, y_{t+1} - 1) & \text{if } op_t = (\text{CHG}, x_t, S_{y_t}) \\ (i_{t+1} + 1, j_{t+1}) = (x_{t+1}, y_{t+1} - 1) & \text{if } op_t = (\text{ADD}, x_t, S_{y_t}) \\ (i_{t+1}, j_{t+1} + 1) = (x_{t+1} - 1, y_{t+1}) & \text{if } op_t = (\text{DEL}, x_t, T_{x_t}) \\ (i_{t+1}, j_{t+1}) = (0, 0) & \text{if } t = -m - 1 \\ (i_0 + 1, j_0 + 1) = (|T| + 1, |S| + 1) & \text{if } t = 0 \end{cases} \quad (2)$$

By rearrangement of the previous equation, we get that:

$$(x_t, y_t) = \begin{cases} (x_{t-1}+1, y_{t-1}+1) & \text{if } op_{t-1} = (\text{NOP}, x_{t-1}, S_{y_{t-1}}) \\ (x_{t-1}+1, y_{t-1}+1) & \text{if } op_{t-1} = (\text{CHG}, x_{t-1}, S_{y_{t-1}}) \\ (x_{t-1}, y_{t-1}+1) & \text{if } op_{t-1} = (\text{ADD}, x_{t-1}, S_{y_{t-1}}) \\ (x_{t-1}+1, y_{t-1}) & \text{if } op_{t-1} = (\text{DEL}, x_{t-1}, T_{x_{t-1}}) \\ (0,0) & \text{if } t = -m-1 \\ (i_0+1, j_0+1) = (|T|+1, |S|+1) & \text{if } t = 0 \end{cases} \tag{3}$$

In order to make the proof simple, we shift the sequence by introducing $\tilde{R}_t = R_{t-m-1}$, $\tilde{x}_t = x_{t-m-1}$, $\tilde{y}_t = y_{t-m-1}$, $\tilde{op}_t = op_{t-m-1}$, after this shifting, the returned result by algorithm 4 is the list of triples $\tilde{R}_1, \cdots, \tilde{R}_m$. The equation after the shifting is given by:

$$(\tilde{x}_t, \tilde{y}_t) = \begin{cases} (\tilde{x}_{t-1}+1, \tilde{y}_{t-1}+1) & \text{if } \tilde{op}_{t-1} = (\text{NOP}, \tilde{x}_{t-1}, S_{\tilde{y}_{t-1}}) \\ (\tilde{x}_{t-1}+1, \tilde{y}_{t-1}+1) & \text{if } \tilde{op}_{t-1} = (\text{CHG}, \tilde{x}_{t-1}, S_{\tilde{y}_{t-1}}) \\ (\tilde{x}_{t-1}, \tilde{y}_{t-1}+1) & \text{if } \tilde{op}_{t-1} = (\text{ADD}, \tilde{x}_{t-1}, S_{\tilde{y}_{t-1}}) \\ (\tilde{x}_{t-1}+1, \tilde{y}_{t-1}) & \text{if } \tilde{op}_{t-1} = (\text{DEL}, \tilde{x}_{t-1}, T_{\tilde{x}_{t-1}}) \\ (0,0) & \text{if } t = 0 \\ (|T|+1, |S|+1) & \text{if } t = m+1 \end{cases} \tag{4}$$

Finally, we introduce a function *Apply*, which can be interpreted as a function that simulates applying the edit operations until it transforms the source string T to the target string S, *Apply* is defined as:

$$Apply(X, R = (x, y, op)) := \begin{cases} X \circ S_y & \text{if } op = (\text{ADD}, x, S_y) \\ X & \text{if } op = (\text{DEL}, x, T_x) \\ X \circ T_x & \text{if } op = (\text{NOP}, x, S_y), \text{ where } T_x = S_y \\ X \circ S_y & \text{if } op = (\text{CHG}, x, S_y) \end{cases} \tag{5}$$

and a sequence X which is defined recursively by: $X_k := Apply(X_{k-1}, \tilde{R}_k)$ for all $k > 0$ and $X_0 = \varepsilon$.

Proof. I will prove that X_k is constructed by transforming the prefix $T_{\twoheadrightarrow \tilde{x}_k}$ to the prefix $S_{\twoheadrightarrow \tilde{y}_k}$ using the sequence of edit operations $\tilde{op}_1, \cdots, \tilde{op}_k$. This is achieved by proving $X_k = S_{\twoheadrightarrow \tilde{y}_k}$ by consuming $T_{\twoheadrightarrow \tilde{x}_k}$ for all $k \leq m+1$, which will be done by

mathematical induction on k.

- **Induction Basis**: $k = 0 \Rightarrow X_0 = \varepsilon = S_{\twoheadrightarrow 0}$
- **Induction hypothesis**: For all $0 \leq r < k$, $X_r = S_{\twoheadrightarrow \tilde{y}_r}$ which is transformed from the prefix $T_{\twoheadrightarrow \tilde{x}_r}$ using the operations $\tilde{op}_1, \cdots, \tilde{op}_r$.
- **Induction step**: If $\tilde{R}_k = (\tilde{x}_k, \tilde{y}_k, \tilde{op}_k)$, then we have one of 4 scenarios:
  - If $\tilde{op}_k = (\text{NOP}, \tilde{x}_k, S_{\tilde{y}_k})$, then $X_k = X_{k-1} \circ S_{\tilde{y}_k}$ (by definition), then $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}} \circ S_{\tilde{y}_k}$ (by induction hypothesis), $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}} \circ S_{\tilde{y}_{k-1}+1} = S_{\twoheadrightarrow \tilde{y}_{k-1}+1} = S_{\twoheadrightarrow \tilde{y}_k}$ (because $\tilde{y}_k = \tilde{y}_{k-1} + 1$). This is acquired by transforming the prefix $T_{\twoheadrightarrow \tilde{x}_{k-1}}$ (by induction hypothesis), and by copying $T_{\tilde{x}_k}$, which is the transformation of $T_{\twoheadrightarrow \tilde{x}_k}$ (because $\tilde{x}_k = \tilde{x}_{k-1} + 1$).
  - If $\tilde{op}_k = (\text{CHG}, \tilde{x}_k, S_{\tilde{y}_k})$, then $X_k = X_{k-1} \circ S_{\tilde{y}_k}$ (by definition), then $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}} \circ S_{\tilde{y}_k}$ (by induction hypothesis), $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}} \circ S_{\tilde{y}_{k-1}+1} = S_{\twoheadrightarrow \tilde{y}_{k-1}+1} = S_{\twoheadrightarrow \tilde{y}_k}$ (because $\tilde{y}_k = \tilde{y}_{k-1} + 1$). This is acquired by transforming the prefix $T_{\twoheadrightarrow \tilde{x}_{k-1}}$ (by induction hypothesis), and by changing $T_{\tilde{x}_k} \rightarrow S_{\tilde{y}_k}$, which is the transformation of $T_{\twoheadrightarrow \tilde{x}_k}$ (because $\tilde{x}_k = \tilde{x}_{k-1} + 1$).
  - If $\tilde{op}_k = (\text{ADD}, \tilde{x}_k, S_{\tilde{y}_k})$, then $X_k = X_{k-1} \circ S_{\tilde{y}_k}$ (by definition), then $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}} \circ S_{\tilde{y}_k}$ (by induction hypothesis), $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}} \circ S_{\tilde{y}_{k-1}+1} = S_{\twoheadrightarrow \tilde{y}_{k-1}+1} = S_{\twoheadrightarrow \tilde{y}_k}$ (because $\tilde{y}_k = \tilde{y}_{k-1} + 1$). This is acquired by transforming the prefix $T_{\twoheadrightarrow \tilde{x}_{k-1}}$ (by induction hypothesis), which is the same as transforming $T_{\twoheadrightarrow \tilde{x}_k}$ (because $\tilde{x}_k = \tilde{x}_{k-1}$), then $S_{\tilde{y}_k}$ is added after this prefix.
  - If $\tilde{op}_k = (\text{DEL}, \tilde{x}_k, T_{\tilde{x}_k})$, then $X_k = X_{k-1}$ (by definition), then $X_k = S_{\twoheadrightarrow \tilde{y}_{k-1}}$ (by induction hypothesis), $X_k = S_{\twoheadrightarrow \tilde{y}_k}$ (because $\tilde{y}_k = \tilde{y}_{k-1}$). This is acquired by transforming the prefix $T_{\twoheadrightarrow \tilde{x}_{k-1}}$ (by induction hypothesis) and deleting $T_{\tilde{x}_k}$, which is the transformation of $T_{\twoheadrightarrow \tilde{x}_k}$ (because $\tilde{x}_k = \tilde{x}_{k-1} + 1$).

In conclusion, we proved that applying the first t returned operations, namely $\tilde{op}_1, \cdots, \tilde{op}_t$, will transform the prefix $T_{\twoheadrightarrow \tilde{x}_t}$ to the prefix $S_{\twoheadrightarrow \tilde{y}_t}$, hereby we can notice that all the operations before $\tilde{R}_t$ will not affect both prefixes $T_{\twoheadrightarrow x_t}$ and $S_{\twoheadrightarrow y_t}$ (it might affect only one though), because for all t we have $\tilde{x}_{t-1} < \tilde{x}_t$ or $\tilde{y}_{t-1} < \tilde{y}_t$ (according to the update equation 4), and therefore $\tilde{R}_k$ is the first operation applied on both of the suffixes $T_{\tilde{x}_t \rightarrow}$ and $S_{\tilde{y}_t \rightarrow}$ simultaneously, since $T_{\tilde{x}_t}$ and $S_{\tilde{y}_t}$ form the only intersection between both suffixes $T_{\tilde{x}_t \rightarrow}$, $S_{\tilde{y}_t \rightarrow}$ and their corresponding prefixes $T_{\twoheadrightarrow \tilde{x}_t}$, $S_{\twoheadrightarrow \tilde{y}_t}$.

□

In the example shown earlier in table 1, the cells marked in grey are the cells that algorithm 4 passes by during the edit operations construction, namely the values s_t, and the corresponding detailed output is given by the list of triples: $(1, 1, (\text{NOP}, 1, a))$, $(2, 2, (\text{DEL}, 2, x))$, $(3, 2, (\text{NOP}, 3, b))$, $(4, 3, (\text{ADD}, 4, x))$, $(4, 4, (\text{NOP}, 4, c))$, and finally $(5, 5, (\text{ADD}, 5, h))$.

3.3 Trie dictionary

Trie is a data structure that is used to store a set or a dictionary of words and to search efficiently on words using word's prefixes [16]. It is built as a rooted directed tree stucture, with labeled edges and nodes. In figure 1, an example of a Trie is depicted that contains 6 strings a, abc, axc, axy, bxy and $abcy$. Each node in the Trie's tree represents a prefix of a certain word, and the highlighted nodes (marked in grey) mark that this is a full prefix, i.e. it is one of the dictionary words that we are storing. The edges of the Trie are labeled by the characters from a given alphabet Σ.

The construction of this structure is done by inserting all words into the Trie. The insertion of a word is done by traversing the labeled edges of the Trie by using the edges with labels from the word's characters while creating necessary nodes on the traversal path until the end of the path is reached (when the word is traversed), then the terminal node is marked. This is depicted in algorithm 5. Given this structure, we can easily search if a word exists or not be traversing through the edges of the tree using the characters of the query word, and at the end, we answer the query depending if the traversal reaches a marked node or not. Furthermore, the search function can be modified in order to search if a query word exists in the dictionary or to find words that are similar to the query word, where the similarity is defined as the edit distance. The modification is done during traversal of the search query; if the traversal is standing at a node u after traversing $q_{1\twoheadrightarrow i}$, we can either try traversing using q_{i+2} as if q_{i+1} is deleted, or traverse all edges that use a character $c \neq q_{i+1}$ then traverse starting from q_{i+2} after as if we changed q_{i+1} to c, or lastly we can traverse using an arbitrary character c and then traverse starting from q_{i+1} as if we added a character c before q_{i+1}, meanwhile we can keep track of how many edit traversals were done and deduct a 'matching' score of the query word accordingly. The deduction will be by a factor $0 \leq \varphi \leq 1$, for each edit operation. The words that are found at the end of the traversal are returned with some matching score, which is equal to 1 or some function on the relative frequency of the matched word. The modified search function is shown by algorithm 6.

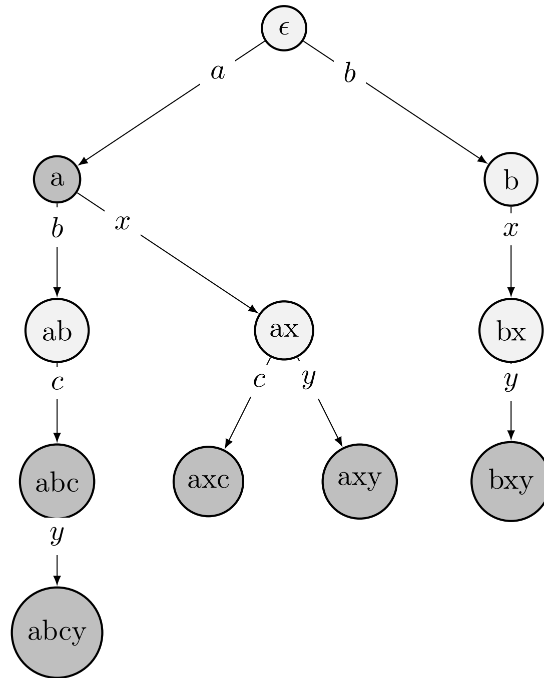

Figure 1: Trie datastructure sample, with 6 inserted strings: 'a', 'abc', 'axc', 'axy', 'bxy', 'abcy'

3.4 Neural network review

Neural networks is one of the machine learning topics, also referred to in some contexts as deep learning. The model consists of nodes/neurons that are partitioned into layers where the layers (and the neurons) are connected in a directed acyclic graph (often a chain). The connections between neurons have weights, that can be tuned in order to tune the output of each neuron and hence the output of the whole network. The architecture of a network consists of input neurons that are grouped in an input layer, then they are connected to hidden neurons (partitioned in hidden layers), then connected to the output neurons which are grouped in an output layer. This structure is depicted in figure 2. The number of layers in the network is also referred to as the network's depth, hence the name deep learning which refers to deep neural networks. One of the strength points of deep neural networks is their capability of approximating complicated functions without collapsing into a linear function, by making use of a number of non-linear layers [17].

Neural networks operate in two main ways, forward propagation and backpropagation. In the forward propagation, the input neurons are fed with values that

Algorithm 5 Trie word insertion

```
procedure INSERT(W, i = 1, nod = ROOT)
    if i > |W| then
        mark nod
    else
        if nod has no edge labeled W_i then
            nod.next[W_i] := new Node          ▷ Create necessary unfound nodes
        end if
        INSERT(W, i + 1, nod.next[W_i])
    end if
end procedure
```

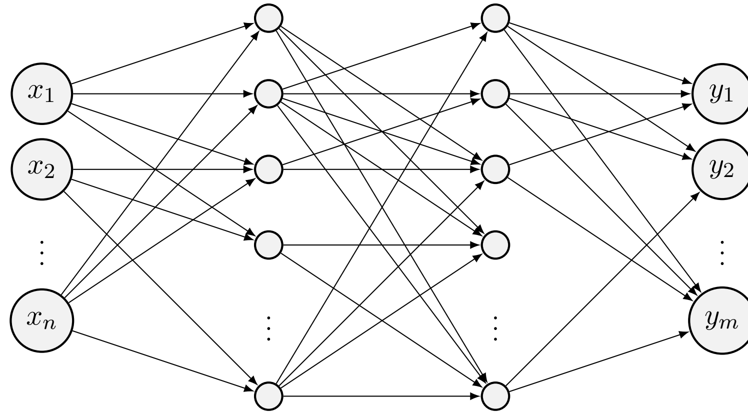

Figure 2: Neural network example, $x_1, \cdots, x_n$ are the input neurons, $y_1, \cdots, y_m$ are the output neurons. The remaining neurons are the hidden ones.

are passed to the successor neurons, then each hidden neuron computes the output depending on its input values, connections' weights and neuron's bias, then the output is fed forward as input to the successor neurons, this goes on until the output layer computes its output, which is the output of the whole network [18]. This is how a neural network can make a prediction or compute a value. This whole structure behaves like a complicated non-linear differentiable function that has the biases and weights as input parameters. On the other hand, backpropagation [18] is used to tune the parameters of the model, it has to be used after a forward propagation; backpropagation relies on the assumption that the forward propagation has cached all its computations, then at the output layer, the predicted output is compared to a ground truth output according to a differentiable cost function J, and based on this comparison there are update values that get backpropagated from the output layer

Algorithm 6 Trie modified search

```
procedure SEARCH(W, i = 1, nod =ROOT, R=‘’, e = 2)    ▷ Word W, index i
                ▷ allowed edits e, R is accumulated result, φ is damping factor
    if i > |W| then
        return (R, int(nod is marked))                ▷ matched word score
    end if
    W_res, s_res := (W, 0.0)
    if nod has labeled edge W_i then
        nxt = nod.next[W_i]
        W_nop, s_nop := SEARCH(W, i + 1, nxt, R ∘ W_i, e)
        if s_nop ≥ s_res then
            W_res, s_res := W_nop, s_nop
        end if
    end if
    if s ≈ 1.0 then
        return (W_r, s)                               ▷ If the word is found, return it
    end if
    W_del, s_del := SEARCH(W, i + 1, nod, R, e − 1)
    if s_del ≥ s_res then
        W_res, s_res := W_del, φ · s_del              ▷ Try to delete W_i
    end if
    for next c of nod do
        W_add, s_add := SEARCH(W, i, nod.next[c], R ∘ c, e − 1)
        if s_add ≥ s_res then
            W_res, s_res := W_add, φ · s_add          ▷ Try to add c before W_i
        end if
        W_chg, s_chg := SEARCH(W, i + 1, nod.next[c], R ∘ c, e − 1)
        if s_chg ≥ s_res then
            W_res, s_res := W_chg, φ · s_chg          ▷ Try to change W_i → c
        end if
    end for
    return (W_res, s_res)
end procedure
```

until the input layer; the updates of a weight between two neurons depend on the backpropagated values as well as the cached values (from the forward propagation) of the output neuron.

The input and output neurons are stacked in the vectors $\mathbf{x}$ and $\mathbf{y}$ respectively. The l-th layer's output is referred to as the vector $\mathbf{h}_l$, where $\mathbf{x} = \mathbf{h}_0$, $\mathbf{y} = \mathbf{h}_L$ and L is the network's depth. The output of the most basic layer (known as fully-connected layer) is computed by $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}\mathbf{W}_l + \mathbf{b}_l)$, where $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weights matrix and bias vector connecting layer $l-1$ and l, and f_l is differentiable function, referred to as activation function. Usually, the activation function is one four common functions: hyperbolic tangent $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$, rectified linear unit $relu(z) = \max(z, 0)$ and the identity function $I(z) = z$; the $relu(z)$ function is not differentiable at $z = 0$, but regardless of that, it is used in practice [19]. During the backpropagation, a parameter $\mathbf{u}$ is updated depending on the gradient of the cost function J, the update value is given by $\Delta\mathbf{u} = \frac{\partial J}{\partial \mathbf{u}} = \nabla J|_{\mathbf{u}}$; the parameter $\mathbf{u}$ is updated with a learning rate α and the value $\Delta\mathbf{u}$ using one of the optimization algorithms like Adam optimization algorithm [20]. One forward propagation followed by backpropagation is a training step. Additionally, the training epochs is defined as the number of times that the model trained on the whole dataset.

3.4.1 Multi-class classification

Neural networks can be used to classify a given input $\mathbf{x}$ as one of k classes, this can be achieved by setting exactly k neurons in the output layer, then augmenting this output layer by Softmax as the activation function [18]. The Softmax is computed by the equation:

$$\text{Softmax}(y_1, y_2, \cdots, y_k) = \left(\frac{e^{y_1}}{\sum_i e^{y_i}}, \frac{e^{y_2}}{\sum_i e^{y_i}}, \cdots, \frac{e^{y_k}}{\sum_i e^{y_i}}\right) \tag{6}$$

The sum of this output vector is 1, as a result, we get an output which is a probability distribution over k possible values corresponding to class 1, $\cdots$ to class k respectively, which is interpreted as the probability of some class to be true. This output is usually evaluated with the cost function *categorical cross-entropy* [18] which is defined by the equation:

$$J(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{k} \hat{\mathbf{y}}_i \log\{\mathbf{y}_i\} \tag{7}$$

where $\hat{\mathbf{y}}$ is the ground truth output vector which is a one-hot vector $\{s = 1\}_{\mathcal{C}}$, where s is the corresponding true class. The value of this cost function will be minimal

(zero) if the probability of the class s that should be predicted is 1, because $\log \mathbf{y}_s = 0$ and for all $q \neq s$ we have $\hat{\mathbf{y}}_q = 0$.

3.4.2 Class sampling

Given a probability distribution vector $\mathbf{p}$ over a set $\mathcal{C}$ of k classes, we can accordingly sample one of the classes by distilling [21] the probability distribution. This is achieved by the function $f_\tau(\mathbf{p}_i) = \frac{e^{\frac{\log \mathbf{p}_i}{\tau}}}{\sum_j e^{\frac{\log \mathbf{p}_j}{\tau}}}$ for some temperature parameter τ, then we can use this new distribution to sample a class i, by uniformly sampling a random number $r \in [0, 1)$ and i as the smallest number such that $\sum_{j=1}^{i} \mathbf{p}_j > r$.

The temperature parameter τ controls how diverse or conservative the sampling is; as $\tau \to 0$ all the values of $f_\tau(\mathbf{p})$ approach 0 except for the class of maximum probability will approach 1, which is the one-hot vector of the most likely predicted class, in this scenario the sampling is strongly sticking to the probabilities predicted in $\mathbf{p}$, which can get stuck on some patterns predicted by $\mathbf{p}$. On the other hand, as $\tau \to \infty$, all the values approach the value $\frac{1}{k}$ which is a uniform distribution, consequently in this scenario, the model is less caring about the probabilities $\mathbf{p}$, which generates more diverse outputs and also with more mistakes.

3.4.3 Dropouts

Dropout [22] is a technique used while training a neural network, in order to enhance the capability of the neural network to generalize over output and be less prune to overfitting. It generally operates by choosing a drop-rate p, and used on some input tensor $\mathbf{Q}$ (n-dimensional array), and it drops randomly an expected ratio of p elements from this tensor, so these will neither be used to compute an output nor be updated; this prevents that the output is dependent on a specific neuron. A simple equation that can be used in practice to compute the dropout, is by sampling $\mathcal{R} \sim \mathcal{U}(0, 1)$ (where $\mathcal{R}$ has the same shape as $\mathbf{Q}$), then using $\mathbf{P} = \frac{1}{1-p}[\mathcal{R} > p]$ and accordingly, the dropped out $\tilde{\mathbf{W}}$ is given by $\mathbf{W} \odot \mathbf{P}$. The term $\frac{1}{1-p}$ is a correction term, because the expected output was reduced by a factor of $1 - p$ because there is a ratio of p dropped out neurons.

3.5 Recurrent neural networks review

Recurrent neural networks (RNN) is a special type of neural networks, where neurons can be connected with self-loops, and thus the output of a neuron is recursive

depending on the output and its own earlier output. This structure is suitable for data that consists of sequences like $x_1, \cdots, x_T$. Given the input sequence $x_1, \cdots, x_T$ and an initial activation state a_0, then the abstract procedure of the RNN is to compute an activation state $a_t = f(a_{t-1}, x_t)$ and an output $y_t = g(a_t)$ for each element x_t in the sequence.

3.5.1 RNN cells: Simple, LSTM, GRU

One of the most basic implementations of RNN is the Vanilla RNN [23]. A Vanilla RNN cell is implemented by the equations:

$$
\begin{aligned}
\mathbf{a}_t &= \sigma(\mathbf{W}^a[\mathbf{a}_{t-1}, \mathbf{x}_t]) \\
\mathbf{h}_t &= \mathbf{O}\mathbf{a}_t
\end{aligned} \tag{8}
$$

The notation $[\mathbf{a}_{t-1}, \mathbf{x}_t]$ denotes a stacking of the vectors $\mathbf{a}_{t-1}$ and $\mathbf{x}_t$. $\mathbf{W}^a, \mathbf{O}$ are the training parameters. However, this RNN has a difficulty to keep a memory of long-term dependencies between the sequence elements. In addition to suffering from vanishing or exploding gradients during backpropagation [24], making the RNN hard to train. The vanishing gradient problem occurs when the gradients have very small (near zero) magnitude such that they are too slow to make any progress in tuning the parameters. On the other hand, the exploding gradient happens when the gradients have very large magnitudes, and as a consequence, the parameters diverge instead of converging to some optimal value, which makes the output of the network undefined. As a consequence, LSTM was introduced in order to solve these issues, and later on, GRU was introduced to simplify the architecture of the LSTM. Both were shown to have comparable performance [25] and superior results in gated RNNs, such that no known gated RNN to date overperforms those two [19].

LSTM

Long short-term memory (LSTM) [26] is a sophisticated RNN that maintains a memory cell $\mathbf{c}$ and gates to decide to forget or remember earlier terms from the given sequence. This solves the issues that the basic RNNs suffered from. A LSTM cell is given by the equations:

$$
\begin{aligned}
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}^c[\mathbf{a}_{t-1}, \mathbf{x}_t] + \mathbf{b}^c) \\
\mathbf{u}_t &= \sigma(\mathbf{W}^u[\mathbf{a}_{t-1}, \mathbf{x}_t] + \mathbf{b}^u) \\
\mathbf{o}_t &= \sigma(\mathbf{W}^o[\mathbf{a}_{t-1}, \mathbf{x}_t] + \mathbf{b}^o) \\
\mathbf{f}_t &= \sigma(\mathbf{W}^f[\mathbf{a}_{t-1}, \mathbf{x}_t] + \mathbf{b}^f) \\
\mathbf{c}_t &= \mathbf{u}_t \odot \tilde{\mathbf{c}}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1} \\
\mathbf{a}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
\tag{9}
$$

Where $\tilde{\mathbf{c}}_t$ is a candidate to replace the previous memory cell $\mathbf{c}_{t-1}$; $\mathbf{u}$, $\mathbf{o}$ and $\mathbf{f}$ are update, output and forget gates respectively. Using the forget $\mathbf{f}$ and update $\mathbf{u}$ gates decide what to forget from the previous memory cell and what to update in it, then the output gate $\mathbf{o}$ decides what is used from the memory cell to output the activation state $\mathbf{a}_t$. $\mathbf{W}^c, \mathbf{W}^u, \mathbf{W}^o, \mathbf{W}^f, \mathbf{b}^c, \mathbf{b}^u, \mathbf{b}^o, \mathbf{b}^f$ are the training parameters of the LSTM cell.

GRU

GRU [27] (Gated Recurrent unit) was introduced as a simplification of the LSTM, that still solves the same issues (long-term dependencies and vanishing/exploding gradients). It also maintains a memory cell $\mathbf{c}_t$, which is also used as the activation state $\mathbf{a}_t$. A GRU cell is given by the equations:

$$
\begin{aligned}
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}^c[\mathbf{r}_t \odot \mathbf{c}_{t-1}, \mathbf{x}_t] + \mathbf{b}^c) \\
\mathbf{u}_t &= \sigma(\mathbf{W}^u[\mathbf{c}_{t-1}, \mathbf{x}_t] + \mathbf{b}^u) \\
\mathbf{r}_t &= \sigma(\mathbf{W}^r[\mathbf{c}_{t-1}, \mathbf{x}_t] + \mathbf{b}^r) \\
\mathbf{c}_t &= \mathbf{u}_t \odot \tilde{\mathbf{c}}_t + (1 - \mathbf{u}_t) \odot \mathbf{c}_{t-1} \\
\mathbf{a}_t &= \mathbf{c}_t
\end{aligned}
\tag{10}
$$

Where $\tilde{\mathbf{c}}_t$ is a candidate the replaces the previous memory cell $\mathbf{c}_{t-1}$; $\mathbf{u}$ and $\mathbf{r}$ are the update and reset gates respectively. The reset $\mathbf{r}$ gate decides what to reset from the previous memory cell. The update gate $\mathbf{u}$ decides what to update in the new memory cell. $\mathbf{W}^c, \mathbf{W}^u, \mathbf{W}^r, \mathbf{b}^c, \mathbf{b}^u, \mathbf{b}^r$ are the training parameters of the GRU cell.

Bidirectional RNN

Bidirectional RNN [19] is a recurrent neural network that processes the input sequence twice, once in a forward direction and secondly in a backward direction using the reversed input sequence. The outputs of both passes are then combined to give the final activation value at some time step. One bidirectional RNN layer is comparable to two unidirectional layers, with the difference that the input of the second layer is fed from the reversed input sequence instead of the output of the first layer. The main advantage of this architecture is providing information from both sides of the sequence at each time step, instead of getting information only from the earlier terms of the sequence. Bidirectional LSTM will be used in some of the presented experiments.

3.5.2 Character-based language model

Character-based language model is a probabilistic model that determines how likely a given string s to appear in a language. It is mainly implemented using a recurrent neural network that uses as input a sequence of characters of fixed length H. The network solves a multi-classification over the character-set Λ, so the output of the network is a probability distribution over the character-set characters predicting the successor character. For example, given the input sequence ‘Hello frien’, and $H = 15$ we feed the input $(\{\dot{\$} = 1\}_\Lambda, \{\dot{\$} = 1\}_\Lambda, \{\dot{\$} = 1\}_\Lambda, \{\dot{\$} = 1\}_\Lambda, \{H = 1\}_\Lambda, \{e = 1\}_\Lambda, \{l = 1\}_\Lambda, \{l = 1\}_\Lambda, \{o = 1\}_\Lambda, \{\text{‘ ’} = 1\}_\Lambda, \{f = 1\}_\Lambda, \{r = 1\}_\Lambda, \{i = 1\}_\Lambda, \{e = 1\}_\Lambda, \{n = 1\}_\Lambda)$ to the language model which predicts a probability distribution over the character-set, denoting the probability of a certain character being a successor, this should predict the character ‘d’ with high probability. This model can be implemented using many-to-many RNN or many-to-one RNN. In my implementation, I used the many-to-one implementation as depicted in figure 3. The output will be denoted by $p_f(c|s_1, \cdots, s_H)$ which is interpreted as the probability that the character c will follow the context string $s_1, \cdots, s_H$. Furthermore, the same model can be implemented to predict the preceding character instead of the successor character, so it will be looking backwards before the given context. Both models will be distinguished as the backward character model and forward character model, which points to which direction the prediction is made. The backward model will be denoted by $p_b(c|s_1, \cdots, s_H)$, which is interpreted as the probability that the character c comes before the string $(s_1, \cdots, s_H)$. Both formulas will be used in the approach chapter 7.

IO specification

The character model takes as input a 3-dimensional tensor $\mathbf{X} \in \{0,1\}^{m \times H \times |\Lambda|}$ where the value $\mathbf{X}_{i,t,c}$ is 1 if the t-th element of i-th example is the c-th character in the character-set Λ, or 0 otherwise. It is a representation of m examples of context strings of length H, represented as one-hot vectors over the character-set. The output is a corresponding 2-dimensional tensor $\mathbf{y} \in [0,1]^{m \times H}$, where $\mathbf{y}_{i,c}$ is the probability that the successor character of the i-th example is the c-th character, namely $p_f(c|s_{i,1}, \cdots, s_{i,H}) = \mathbf{y}_{i,c}$ for all i, c.

Additionally, if the context string is longer than H, than the last H characters are taken, in case of the forward model, and the first H characters are taken in case of the backward model. However, in case the context string is shorter than H, it is padded from the beginning with $\dot{\$}$ characters, in case of the forward model, and it is padded from the end with $\dot{\$}$ characters. This ensures that the input string will always have the length H. Using of the padding characters was shown in the example in the character-based language model definition.

Fake text generation

One of the applications of the character-based language model is the fake text generation. It operates simply by maintaining a context string, of fixed history length, and based on it, we predict the next character using the model. The newly predicted character is appended to the result string and to the maintained context. This keeps going on until the result string reaches a maximum length or some distinguished character like an 'end of text' character. The sampled text yielded some surprisingly good results [5], for example when the model is trained on Wikipedia articles, the language model generates articles that look like Wikipedia articles with similar vocabulary. [1] One of the models that I trained on the Simple-Wikipedia dataset, using character class sampling as described in subsection 3.4.2, with diversity temperature $\tau = 0.5$, generated the text:

> Island is a city of India. It is also colleged by the season of the Mangandand went to the part of the more many of the part of the based of the band of the and a player and head for the area of the police, and in order and made in see that an area in a lot of the plants and a bened in children services and the character that has been a second and his movie, and a

[1] There are more examples mentioned in the blog post `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

> second former encourage and the main faction of the company state of the television of the series. It is also have been another characters. The movie probably for the some of the same of the music in the movie shopper and has a little of a commune. It is found in the second of the changes and parts of the second of the lands and can be offered in the United States. In 1964, the many was called a population of the change of the comeon particles and particent. He was also the first most individual and and interest and singing the family in the common Constantery in 1997.

And using temperature $\tau = 0.1$, generates the more conservative (repetitive) text:

> Island is a commune. It is found in the United States. It is a population of the province of the second of the programs and the second of the second of the state of the second of the second company in the United States. It is a commune. It is found in the United States. It is a province of the second of the south of the Constitution of the United States. It is a province of the part of the second of the province of the state of the second of the south of the United States. It is a province of the second of the state of the state of the south of France. The company of the second of the second of the state of the country in the United States. It is a province of the second of the second of the second company in the country of the second are a state of the second of the second of the second of the second of the state of the second of the second production of the state of the second of the state of the second of the second of the $\cdots$

Using temperature $\tau = 3.0$ generates a total random string:
" PeoplePSefosaCjiofhCntwas"mhefeonkshefro4cPunordend "

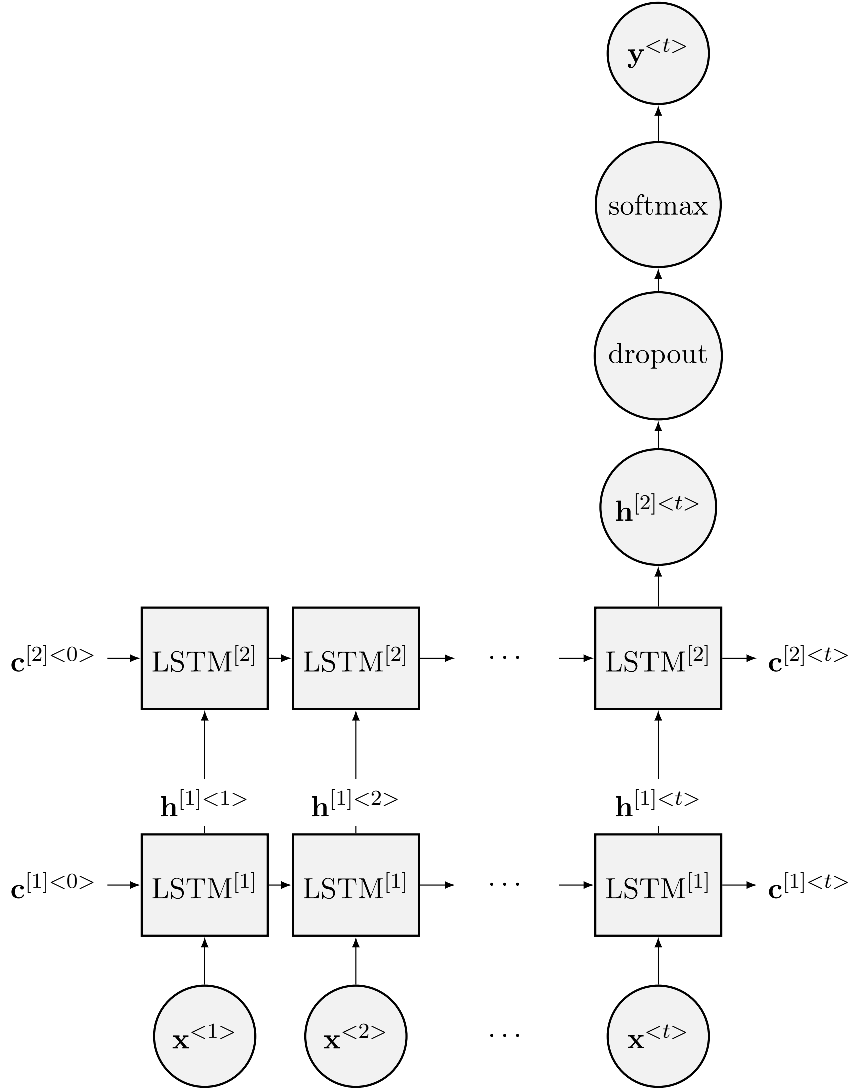

Figure 3: Many-to-one two-layered unrolled LSTM recurrent neural network, $\mathbf{x}^{<t>}$ is the t-th element in the input sequence, $\mathbf{c}^{[l]<t>}$ is the t-th activation value at the l-th layer. The output of the network $\mathbf{h}^{[l]<t>}$ is followed by a dropout layer then by a softmax layer, in order to classify the successor element in the sequence.

4 Datasets

The given problem needs a dataset that consists of texts and corrupt texts, and since it preferable to have a big amount of data for training[28], datasets from large corpus were used. In the training and evaluations, two main datasets were used, a collection of texts from Simple-Wikipedia, and the other dataset consists of news from Reuters-21578. A summary of some statistics about the datasets is shown in table 2.

The datasets are available as compressed files, which are uncompressed into a bunch of raw files, each of them containing a set of articles in some format. The format for each dataset will be mentioned in the corresponding sections 4.1 and 4.2. After that, the articles' texts are extracted from the raw files into text files. Additionally, corrupted versions of the texts are constructed, as well as, files are computed, which are storing the edit operations between the correct and corrupt texts.

During training or evaluation, the articles were loaded in a sorted lexicographically order by file name, then shuffled in order to easily use a random sample that is representative of the whole dataset, additionally, to ensure the replication of data throughout the experiments, the files were always shuffled with a fixed seed (which was 41). Furthermore, since the datasets only consist of correct text, a corruption mechanism had to be made for synthesizing the correct texts, and generating corresponding corrupt text files, the corruption algorithm is explained in details in section 4.3 and the construction of edit operations is done as explained by subsection 3.2.1 using algorithms 4 and 3.

4.1 Simple-Wikipedia dataset

Simple-Wikipedia is a corpus that contains articles like the English Wikipedia with similar format, however, the articles are simpler in terms of length and complexity of English language, yet still the articles contained a variety of topics from diverse domains, which makes it a dataset with wide scope of topics, making the dataset convenient to represent a large corpus with a diversity of domains. For example,

Feature	Reuters-21578 dataset	Simple-Wikipedia dataset
# of articles	19,043	131,566
Avg. # of characters	800	800
Vocabulary size	49,847	348,924
Vocabulary size (freq ≥ 3)	24,986	130,191
Domain	Economics	Various
Raw format	SGM	XML

Table 2: Summary of the datasets

the dataset contains articles about 'Donald Trump'(politics), 'Evolution'(biology), 'Ahmadiyya'(religious sect), 'Catholicism'(religious sect), 'China'(country), 'Plato' (ancient philosopher), 'One Formula' (entertainment) , 'Rings' (abstract-algebra), 'The Matrix' (movie), 'Angela Merkel'(politics), 'DNA' (biology), 'Benzene' (organic chemistry) 'Bonsai' (tree), 'Freiburg im Breisgau' (city) and 'Egyptian Pyramids' (tourism). The dataset contains nearly 130,000 articles, where the article on average consists of 800 characters. Furthermore, the vocabulary of the dataset consists of 348,924 distinct words (from which 130,191 had frequency > 2) which is another indicator of the diversity of the dataset, in comparison with the Reuters-21578 dataset.

The version of the dataset that I used throughout my experiments is the simple Wikipedia dump 20180201 [1], which is published on the 1st of February 2018. The raw format of the dataset is stored in a compressed XML in bz2 compressed format. I extracted it into JSON format using Attardi's script: WikiExtractor. Attardi's script is a pure Python script that extracts and cleans Wikipedia articles from XML compressed files. [2] The articles are extracted into a list of files, each contains a list of lines, where each line is a JSON description of an article, including the title, article id and text. I parsed the JSON descriptions into text files, and processed the text by removing empty lines from the text.

[1] The dataset is available in the link `https://dumps.wikimedia.org/simplewiki/20180201/simplewiki-20180201-pages-meta-current.xml.bz2`

[2] Attardi's WikiExtractor script is available on the link `https://github.com/attardi/wikiextractor`

4.2 Reuters-21578 news dataset

The Reuters-21578 ("Reuters-21578, Distribution 1.0") dataset was collected from news articles from Reuters news-wire [3]; the dataset has mostly short news over a period of 10 months in 1987. Some of the articles in this dataset were annotated with some topics. The top topics included: earn, acq(acquisition), money-fx, crude, grain, trade, interest, wheat, ship, corn, dlr (dollar), oilseed, money-supply and sugar; which points out that the articles in this dataset has a narrower scope diversity, that is more specific to economics. The dataset contained 19,043 articles, with an average that consists of 800 characters per article. Additionally, the vocabulary of the Reuters-21578 dataset contained 49,487 distinct words (from which 24,986 had frequency > 2), which confirms the specificity of the domain of the dataset.

The raw format of the dataset is a list of SGM files (XML is a derivative of SGM); it is formatted as a tree of tags containing a variety of information. Each article is contained in a REUTERS tag, which contains some metadata and a TEXT tag which contains the main information about the text, like a TITLE tag for article's title and a BODY tag which contains the actual text of the article. I extracted the texts in the BODY tags into text files, and processed the texts by removing tabs (4 spaces) at the beginning of each paragraph.

4.3 Corruptions

The construction of a large corpus with good and corrupt texts is necessary for a rich dataset that can be used for deep learning, since deep learning often gets better with a large amount of training data [28]. Therefore, in order to have the freedom of the corruptions' distribution, I constructed a corruption generation algorithm that synthesizes the correct text files, and generates corresponding corrupt files. The corruptions are constructed to mimic the tokenization errors mentioned in the introduction chapter 1. The algorithm tokenizes the good text into a list of tokens, and randomly (with probability p, I chose $p = 0.5$) corrupts a token by applying a corruption operation on this word. The corruption operation either merges a token with the token after it, splits a token into 2 by adding a space, splits a token into two lines using '-[newline]' or finally introduces an edit operation on the token. The 4 corruptions are distributed with the ratios $4 : 4 : 1 : 2$ respectively. This corruption procedure is shown in algorithm 7.

[3] The dataset is available on `https://archive.ics.uci.edu/ml/datasets/reuters-21578+text+categorization+collection`

The algorithm can be demonstrated on the example text:

'This[2] text is a[1] really[1] good text[1]
to demonstrate[3] the[4] corruption example[1] and
how it[4] works, because[1] it is long[2] enough[2] to[4] show how[1] all the various
corruptions[3] apply. Additionally[2] I am putting[2] here[1] some[2] more
extra[2] text just to[2] make[4] it[1] long enough.'

The underlined words are the tokens chosen with expected 50% chance to be corrupted, there are 44 words in this paragraph, and 22 of them are underlined. According to the ratio 4 : 4 : 1 : 2 mentioned, we expect to find 8 tokens with token merge with the previous token (marked with 1), 8 tokens with word splits (marked with 2), 2 tokens with split using line-hyphenation (marked with 3) and 4 tokens with typos(marked with 4). As a result, it is corrupted to the following text:

'Th is text is areallygood textto
demons-
trate ther corruption exampleand
how if works, becauseit is lon g eno ugh go show howall the various
corrupt-
ions apply. Additional ly I am puttin g he resome more
ext ra text just t o fake itlong enough.'

Algorithm 7 Corruptor

```
function CORRUPT(S, p = 0.5)
    T := TOKENIZE(S)
    R := []                                          ▷ result corrupt tokens
    u := nil                                         ▷ Last processed token
    foreach t in T do
        if u is not nil then
            t := Token(u.word + t.word, t.split)
            u := nil
        end if
        mode := RANDOM-SAMPLE(merge-next : 4, split : 4, line-split : 1, edit : 2)
        if mode = merge-next then
            u := t
        end if
        if RANDOM() ≤ p and |w| ≥ 2 then
            w = t.word
            sp = t.split
            i = RANDOM-INT(1, |w| − 1)
            if mode = split then
                R.APPEND(Token(w_{→i}, ' '))
                R.APPEND(Token(w_{i→}, sp))
            end if
            if mode = split-line then
                R.APPEND(Token(w_{→i}, '-[newline]'))
                R.APPEND(Token(w_{i→}, sp))
            end if
            if mode = edit-operation then
                q := SAMPLE-EDIT(w, Σ)
                R.APPEND(Token(q, sp))
            end if
        else
            R.APPEND(t)
        end if
    end for
    return R
end function
function SAMPLE-EDIT(w, Σ)
    op := RANDOM-SAMPLE(ADD, DEL, CHG)
    c := RANDOM-SAMPLE(Σ)
    if op = ADD then
        i := RANDOM-INT(1, |w| + 1)
        return w_{→i} ∘ c ∘ w_{i→}
    end if
    if op = DEL then
        i := RANDOM-INT(1, |w|)
        return w_{→i} ∘ w_{i+1→}
    end if
    if op = CHG then
        i := RANDOM-INT(1, |w|)
        return w_{→i} ∘ c ∘ w_{i+1→}
    end if
end function
```

5 Problem definition

The given problem is formally defined as: Given a dataset that consists of pairs (G, C) where G is a correct string, C is a string which is a corrupt version of G, with tokenization corruptions (C can be constructed by algorithm 7), then we need to construct a procedure *'fix'* that fixes C into a string $F := fix(C)$, where F is as close as possible to G. The performance of the procedure 'fix' is evaluated as a binary classification task where the procedure should classify edit operations as *positive* if and only if the operation corrupted $G \to C$, and hence fix it. The closeness of F to G is measured by the F_1-score of this binary classification task. The definition of being "close" is detailed in section 5.1. At the end of the chapter, there is a full rigorous definition of the given problem.

5.1 Fixing evaluation definition

In order to quantify how 'good' a fixing is, we need to define a measure of how "close" is the fixed text to the actual correct text. This alone is not sufficient to quantify the performance of a fixing, because we also need to take into consideration how corrupted the corrupt text was, and also we need to consider if the fixer actually introduced bad fixings that corrupted the good parts. In order to measure how much improvement the fixing made, a triple comparison between the 3 texts G, F, C will be introduced, to quantify the quality of the fixing.

If we define $\mathcal{C} := EditOperations(G, C)$ and $\mathcal{F} := EditOperations(G, F)$, where *EditOperations* is not detailed (by setting detailed flag in algorithm 4 to false), then $\mathcal{C}$ gets the operations that transform the correct text to the corrupt text, which are exactly the corruption operations. Similarly, $\mathcal{F}$ gets the operations that transform the correct text to the fixed text, which are the parts that were not fixed properly by the procedure 'fix'. Therefore, $\overline{\mathcal{F}}$ is the set fixing/copy operations predicted by the procedure.

The set $\mathcal{C} \setminus \mathcal{F}$ is the set of corruption operations that are not in the non-fixed operations, so it is the set of corruptions $G \to C$ that are actually fixed. The set

$\mathcal{C} \cap \mathcal{F}$ is the set of corruption operations that are not fixed; the set $\mathcal{F} \setminus \mathcal{C}$ is the set of non-fixed operations that were not corruption operations, in other words, it is the set of wrongly introduced fixings by the procedure 'fix'. Finally, the set $\overline{\mathcal{C} \cup \mathcal{F}}$ is the set of copy operations that are shared between the corruption text C and fixed text F which points to the characters that were unaffected neither by corruption nor by fixing. A further note, the set $\mathcal{F} \cap \mathcal{C}$ has to be computed by longest common subsequence LCS($\mathcal{F}$,$\mathcal{C}$) algorithm [13], because the intersection has to take the order of operations in consideration, especially because the addition operations can be ambiguous. For example, Adding 'a' then 'b' at position 15, is not the same as 'b' and 'a' at position 15. The edit operations in algorithm 4 will be returned in the proper order suitable for LCS. Additionally, we can't use the detailed edit operations because they include alignments between G and each of C and F, and these alignments are unrelated and thus hard to to use it to evaluate.

According to the binary classification task defined earlier, $\mathcal{C}$ is the set of actual positives. The sets of predicted positives and negatives are a bit tricky to identify. The operations predicted as positive are the operations where fixings were made either correctly fixed as in $\mathcal{C} \setminus \mathcal{F}$ or wrongly fixed as in $\mathcal{F} \setminus \mathcal{C}$. The operations predicted as negatives are the non-fixed operations, either by replicating the corruptions as in $\mathcal{F} \cap \mathcal{C}$ or by copying operations as in $\overline{\mathcal{F} \cup \mathcal{C}}$. Based on this, we can deduce that the true-positives are given by $|\mathcal{C} \cap \overline{\mathcal{F}}| = |\mathcal{C} \setminus \mathcal{F}|$, the false positives are given by $|\mathcal{F} \cap \overline{\mathcal{C}}| = |\mathcal{F} \setminus \mathcal{C}|$, and the false negatives are given by $|\mathcal{F} \cap \mathcal{C}|$. Then we can use the classification measures recall and precision, in addition to the F_1-score which is defined as the harmonic mean between the recall and precision [29].

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN}, F_1 = \frac{2PR}{P + R} \tag{11}$$

These formulas are expanded to:

$$P = \frac{|\mathcal{C} \setminus \mathcal{F}|}{|\mathcal{C} \setminus \mathcal{F}| + |\mathcal{F} \setminus \mathcal{C}|} = \frac{1}{1 + \frac{|\mathcal{F} \setminus \mathcal{C}|}{|\mathcal{C} \setminus \mathcal{F}|}}$$

$$R = \frac{|\mathcal{C} \setminus \mathcal{F}|}{|\mathcal{C} \setminus \mathcal{F}| + |\mathcal{F} \cap \mathcal{C}|} = \frac{1}{1 + \frac{|\mathcal{F} \cap \mathcal{C}|}{|\mathcal{C} \setminus \mathcal{F}|}}$$

$$F_1 = \frac{2|\mathcal{C} \setminus \mathcal{F}|}{2|\mathcal{C} \setminus \mathcal{F}| + |\mathcal{F}|} = \frac{1}{1 + \frac{|\mathcal{F} \setminus \mathcal{C}| + |\mathcal{F} \cap \mathcal{C}|}{2|\mathcal{C} \setminus \mathcal{F}|}} = \frac{1}{1 + \frac{|\mathcal{F}|}{2|\mathcal{C} \setminus \mathcal{F}|}}$$

All the 3 values get higher if there are more corruptions are fixed. However, the

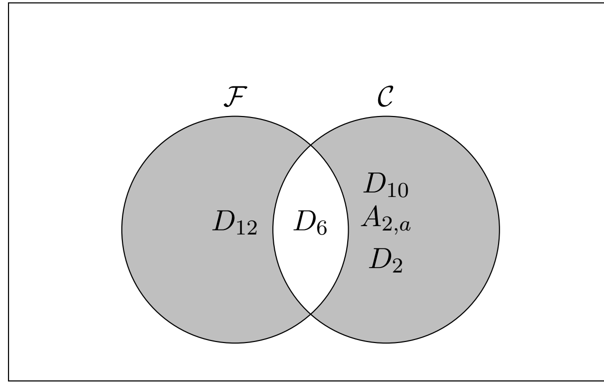

Figure 4: Simple Venn diagram of the two sets $\mathcal{F}$ and $\mathcal{C}$. Deletion and addition operations have the symbols D and A, respectively. G = 'Hello world', F = 'Helloworld!' and C = 'Halloword'

precision P gets lower if the text is fixed destructively, which is corrupting correct parts of the given corrupt text; the recall R gets lower if the there are less fixed corruptions, so the recall points out to how much progress the fixer made on the corrupted parts. Eventually, the F_1-score is the harmonic mean of P and R, so it will get higher if more corruptions are fixed and will get lower if destructive fixings are introduced, therefore it measures the overall progress of the fixer. Additionally from the last equation, we can deduce that F_1-score will get less if the edit distance $|\mathcal{F}|$ between F, G gets higher (hence less-similar), and it will get more if more corruptions are fixed, therefore it measures in a sense how much correct text is retrieved from the corrupt text, relative to how much corruptions there are in the corrupt text, which solves the issues addressed at the beginning of the section. These 3 metrics lie between $[0, 1]$, where 1 denotes the perfect fixing.

A further note on the last equations, in case $|\mathcal{C} \setminus \mathcal{F}| = 0$, then only the first forms of the equations are valid because they don't have a division by 0. Additionally, their numerators and denominators can be smoothed by adding a smoothing $\epsilon \approx 10^{-10}$ in order to avoid division by 0 in case the corrupt text is identical to the correct text, which results in $|\mathcal{C}| = |\mathcal{C} \setminus \mathcal{F}| = |\mathcal{C} \cap \mathcal{F}| = 0$.

The effects of these formulas can be demonstrated with an example; considering $G =$ 'Hello world', $C =$ 'Halloword' and $F = \text{fix}(C) =$ 'Helloworld!'. The sets of edit/fixing operations, depicted in figure 4, are given by:

$$\begin{aligned}
\mathcal{F} &= \{(\text{DEL}, 6, \text{` '}), (\text{ADD}, 12, \text{`!'})\} \\
\mathcal{C} &= \{(\text{DEL}, 2, \text{`e'}), (\text{ADD}, 2, \text{`a'}), (\text{DEL}, 6, \text{` '}), (\text{DEL}, 10, \text{`l'})\} \\
\mathcal{C} \setminus \mathcal{F} &= \{(\text{DEL}, 2, \text{`e'}), (\text{ADD}, 2, \text{`a'}), (\text{DEL}, 10, \text{`l'})\} \\
\mathcal{F} \setminus \mathcal{C} &= \{(\text{ADD}, 12, \text{`!'})\} \\
\mathcal{C} \cap \mathcal{F} &= \{(\text{DEL}, 6, \text{` '})\}
\end{aligned}$$

Which gives the values $P = R = F_1 = \frac{3}{4}$. Instead, if $F =$ 'Hello world!' with fixings of all corruptions, then $\mathcal{F} = \{(\text{ADD}, 12, \text{`!'})\}$, and $|\mathcal{F}| = 1, |\mathcal{C} \cap \mathcal{F}| = 0$, and $P = \frac{4}{5}, R = 1, F_1 = \frac{8}{9}$. Additionally, if instead, $F =$ 'Helloworld' with no destructive fixings introduced, then $\mathcal{F} = \{(\text{DEL}, 6, \text{` '})\}$, $|\mathcal{F}| = 1, |\mathcal{C} \cap \mathcal{F}| = 1$ and $P = 1, R = \frac{3}{4}, F_1 = \frac{6}{7}$.

In conclusion, using the F_1-score, as defined, quantifies all the issues addressed at the beginning of the section, and therefore we have a clear definition of fixing performance of a procedure 'fix'. Consequently, we can give a full rigorous definition of the problem.

Formal problem definition

Given a set of texts $G_1, \cdots, G_m$ of good texts, that are corrupted into $C_1, \cdots, C_m$ respectively, using the corruption procedure in algorithm 7, that is $C_i = \text{CORRUPT}(G_i, p = 0.5)$, then we should construct a procedure *'fix'* that maps $G_i \mapsto^{\text{fix}} F_i$ in a manner that maximizes the triple comparison performance measure, namely the mean F_1-score given by the equation:

$$F_1 = \frac{1}{m} \sum_{i=1}^{m} \frac{2|\mathcal{C}_i \setminus \mathcal{F}_i| + \epsilon}{2|\mathcal{C}_i \setminus \mathcal{F}_i| + |\mathcal{F}_i| + \epsilon} \tag{12}$$

where $\epsilon = 10^{-10}$ is for smoothing, $\mathcal{C}_i = \mathit{EditOperations}(G_i, C_i, \text{detailed} = \text{false})$ and $\mathcal{F}_i = \mathit{EditOperations}(G_i, F_i, \text{detailed} = \text{false})$ as $\mathit{EditOperations}$ is computed by algorithm 4.

6 Dictionary-based approaches

In this chapter, two dictionary-based approaches will be presented, that are not based on machine learning. Both approaches are dictionary based, so they attempt to match words with a given dictionary, where the dictionary is implemented using a Trie data structure as explained in section 3.3. The first approach is a greedy-based approach, which tries to match words from beginning to end greedily according to the dictionary, this greedy approach will be the baseline approach. The second is a dynamic-programming (DP) based approach, which solves some of the issues in the greedy approach. The DP approach tries to re-split the tokens of a given text in attempt to globally match (according to the dictionary) as many correct words as possible.

6.1 Greedy based approach

The greedy approach is a simplistic approach that attempts to solve the given problem. Given a text T, it consumes a prefix of T to find all similar matching words, and from the candidate matching words, it chooses one word greedily based on its length and the minimal number of edit operations needed to match a dictionary word. It keeps doing this until it consumes all the given text T and retrieves all the correct words from the corrupt text. In order to match the first word, we traverse the Trie dictionary using the non-delimiter characters from T with a restricted number of allowed edit operations to use, in case words are partially matching. The traversal is similar to the search query of the Trie, except that it returns all candidate words and not only the first matched, this traversal is shown in algorithm 8, all the marked nodes that were reached by the traversal are marked as candidate nodes (with their corresponding candidate words), the matched word is chosen from the candidate nodes to minimize the value $\min(2, |w|) - e$, where w is the matched word, and e is the number of edits used. In case there are tokens that consist of digits or special characters only, they will be also matched as correct words. This description is shown in the algorithm 9.

Algorithm 8 Greedy Trie traversal

```
function GET-NEXTS(T, i, u, e, R, d)          ▷ Text T, index i, Trie node u
                                              ▷ edits allowed e, results R, last delimiter d
    if T_i ∈ Γ then                           ▷ Don't use delimiters in R
        yield-all GET-NEXTS(T, i + 1, u, e, R, T_i)    ▷ Change last delimiter d
    end if
    if T_i ∈ Σ_digits ∪ Σ_special then ▷ Don't traverse Trie with non-English alphabet
        yield-all GET-NEXTS(T, i + 1, u, e, R ∘ T_i, d)
    end if
    if i ≤ |T| then
        if u is v as next node with character T_i then
            v := u.next[T_i]
            yield (v, i + 1, e, R ∘ T_i, d)            ▷ Consume the text
        end if
    end if
    if e > 0 then                                ▷ Try editting the word
        if i ≤ |T| then
            yield (u, i + 1, e − 1, R, d)              ▷ Delete T_i
        end if
        foreach char c, next node v in Trie D do
            yield (v, i, e − 1, R ∘ c, d)              ▷ Add c before T_i
            if i ≤ |T| and c ≠ T_i then
                yield (v, i + 1, e − 1, R ∘ c, d)      ▷ Change T_i → c
            end if
        end for
    end if
end function
```

Algorithm 9 Greedy approach

```
function GET-WORD(T, i = 1, u = ROOT, e = 1, R = '', d = ' ')
                                              ▷ Text T, index i, Trie node u
                          ▷ edits allowed e, results R, last delimiter d
    F := []
    Q := [(u, i, e, R, d)]
    while Q  is not Empty do
        foreach q ∈ Q do
            if q_{|u} is marked or q_{|R} ∈ (Σ_{digits} ∪ Σ_{special})^+ then
                F.APPEND(q)                                   ▷ final state
            end if
        end for
        N := []
        foreach q ∈ Q do
            S := GET-NEXTS(T, q)
            N.EXTEND(S)
        end for
        Q := N
    end while
    compare := λ · q : min{2, |q_{|R}|} − q_{|e}   ▷ Lambda expression with input q
    q := FINDMAX(F, compare)
    if q is not nil then
        return (q_{|i}, q_{|R} ∘ q_{|d})                  ▷ Word extracted
    end if
    return (i + 1, '')
end function
function FIX(T)
    R := ''
    i := 1
    while i ≤ |T| do
        i, w := GET-WORD(T, i)
        R := R ∘ w                          ▷ word with appended delimiter
    end while
    return R
end function
```

6.2 Dynamic programming based approach

The dynamic programming (DP) approach has mainly three layers of abstraction, built on top of each others. The first layer of the algorithm is the scoring of a token, which decides how a token's word will be scored if it matches (fully or partially) with the dictionary. The other two layers are shown in the example below. The second layer of the approach is a divide and conquer approach; when given a sequence of tokens, it tries to re-split the characters in the tokens' words in order to maximize their overall score of matching the dictionary. The third layer is also a divide and conquer approach, that decides which consecutive tokens should be grouped together, such that the grouped tokens are retokenized (using the second layer) in order to best match the given dictionary, then the newly constructed tokens are re-merged to form the *fixed* text.

For example, given a text like:

'Hello, thisis the mostbasic ex amp le f or the algorithm."

would idealy be grouped as:

'(Hello)(,) (thisis) (the) (most)(basic) (ex amp le) (f or) (the) (algorithm)(.)"

and then the characters are retokenized as:

'(Hello)(,) (this is) (the) (most) (basic) (example) (for) (the) (algorithm)(.)"

and finally we get the fixed text:

'Hello, this is the most basic example for the algorithm."

6.2.1 Token scoring

Given a word w and a Trie dictionary D with damping factor φ for mismatches. We will define a score function that tells how good this word matches the dictionary. This scoring will be used in the 2nd layer of the algorithm, by summing up the score of the individual tokens. Therefore, the scoring that will be constructed here, has to take few issues into consideration:

1. The score should reward if more tokens are matched over the scenario that fewer tokens are matched.
2. The score should reward matches of long words over short words.

3. The score should punish a match depending on the edit distance to the closest matched word.

4. The score should take special consideration of abbreviations and special characters tokens.

The first consideration is essential in maximizing the number of matched tokens in order to fix as much as possible from the text, another role for this is to balance the effect of the second consideration. The second consideration is needed to avoid the scenario that the algorithm prefers to use up correct parts of the word as more correct tokens and leave some unusable bad parts. For example, we should avoid giving the combined score of the 3 tokens: 'a', 'fore', 'mentioned' higher score than 'aforementioned'. The third consideration is to assist fixing words with few typos, to the nearest word, with partial score, however it will prefer not to change the correct words. This might not fix words that have typos, if they happened to be in the dictionary. The fourth consideration is mainly to handle the special cases that were not properly addressed by the other considerations, this gives a fixed score for some specific words, mainly abbreviations.

After taking all of this into consideration, I proposed a quardratic scoring function:

$$\text{score}(w) = \begin{cases} \zeta & \text{if } 3 \leq |w| \leq 4 \text{ and } w == w.\text{upper}() \\ 0 & \text{if } e(w, D) > 1 \\ \varphi^{e(w,D)}(\alpha|\hat{w}|^2 + \beta|\hat{w}| + \gamma) & \text{otherwise} \end{cases}$$

where $e(w, D)$ is the smallest edit distance to a matching word $\hat{w}$ in the Trie dictionary D and φ is the dictionary's damping factor. $\alpha, \beta, \gamma, \zeta$ are used to tune the scoring. The use of quadratic terms is mainly to handle the second consideration, because of the property that $|w_1|^2 + |w_2|^2 < (|w_1| + |w_2|)^2$. The constant term γ handles the first consideration. The term $\varphi^{e(w,D)}$ handles the third consideration, and the fourth consideration is handle by the constant ζ. Furthermore, since the functions in the rest of the algorithm will just use linear combinations of this scoring function, and we are seeking the maximum answer (without particularly caring if the score is scaled), therefore we can divide all equation by γ (assuming that $\gamma \neq 0$) in order to have less

parameters to tune, so we end up having the scoring function:

$$\text{score}(w) = \begin{cases} \zeta & \text{if } 3 \leq |w| \leq 4 \text{ and } w == w.\text{upper}() \\ 0 & \text{if } e(w, D) > 1 \\ \varphi^{e(w,D)}(\alpha|\hat{w}|^2 + \beta|\hat{w}| + 1) & \text{otherwise} \end{cases} \tag{13}$$

After some experimenting (on a relatively small sample) with a variety of combinations of values, $\alpha = 1.15, \beta = 0.1, \gamma = 1, \varphi = 0.5, \zeta = 2$ were found to yield the best results throughout my experiments, measured by F_1-score. A demonstration of some queries and their scores are presented in table 3.

Query word q	matched word w	length $\|w\|$	$e(w, D)$	Score(q)	$\frac{Score(q)}{\|w\|}$
Hello	Hello	5	0	30.25	6.05
world	world	5	0	30.25	6.05
warld	world	5	1	15.125	3.025
war	war	3	0	11.65	3.88
to	to	2	0	5.8	2.9
td	to	2	1	2.9	1.45
today	today	5	0	30.25	6.05
tday	today	5	1	15.125	6.05
the	the	3	0	11.65	3.88
query	-	0	> 1	0	0
ad	-	0	> 1	0	0

Table 3: Table of tokens' scores, q is a given query word, w is the nearest match for it from the dictionary D, $e(w, D)$ is the corresponding edit distance (between w and q), Score(q) is value of the scoring function on the word q. The dictionary contains the words 'Hello', 'world', 'war', 'to', 'today' and 'the'.

6.2.2 Retokenization

The second layer of the DP approach is the retokenization, which is given a token (w, sp) then we need to re-form it into new sequence of tokens $Q_1, Q_2, \cdots, Q_s$ such that $\sum_{i=1}^{s} \text{Score}(Q_i)$ is maximal. This problem can be solved as a dynamic programming problem. The state of the dynamic programming is a single integer i where $1 \leq i \leq |w| + 1$, which encapsulates the suffix $w_{i\rightarrow}$. The subproblems of $w_{i\rightarrow}$ are shorter suffixes $w_{j\rightarrow}$ where $i < j \leq |w| + 1$. We can solve the problem $w_{i\rightarrow}$ by trying to take

the word $w_{i\to j}$ and solving the subproblem $w_{j\to}$ of the remaining characters. This is achieved by the recursive relation:

$$B_w[i] = \max_{i<j\leq|w|+1} \{B_w[j] \cdot \theta_{i,j} + (1-\theta_{i,j}) \cdot Score(w_{i\to j})\} \tag{14}$$

The function $B_w[i]$ is interpreted as the optimal retokenization score of some suffix $w_{i\to}$. It is also shown in algorithm 10 how this recursion is computed in a bottom-up approach (from smallest suffixes to bigger suffixes). The factor $\theta_{i,j} = \frac{|w|-j+1}{|w|-i+1}$ is a normalizing factor, that normalizes the score of a suffix $B_w[i]$ by the suffix's length, which results in stronger consideration of the word $w_{i\to j}$, because without this factor the value of $Score(w_{i\to j})$ will be much less than $B_w[j]$ when the length of the suffix $w_{j\to}$ is much longer than the word $w_{i\to j}$, leading to inaccurate computations sometimes.

	H	e	l	l	o	w	a	r	l	d
i	1	2	3	4	5	6	7	8	9	10
B	22.69	12.80	10.18	11.63	12.79	15.13	1.45	0.0	0.0	0.0
nxt	6	6	4	6	6	11	9	9	10	11

Table 4: Table of values of retokenization of 'Hellowarld'. The grey cells mark the beginning of words as chosen by the traceback array *nxt*.

An example of retokenization of the string 'Hellowarld' is shown in table 4, the given string is retokenized into 'Hello world'. The table shows the values of B and the traceback indices nxt, which are the ending indices of the first token for each suffix. We start by picking the first token for the whole string (suffix $B[1]$), then pick the next token starting from index $6 = nxt[1]$, then stop because $nxt[6] = 11$. The beginnings of tokens are marked with grey color in the table, which also form the traceback path.

6.2.3 Grouping

The third layer of the algorithm is grouping the tokens $T_1, T_2, \cdots, T_n$, which is partitioning the sequence of tokens into a sequence of groups of tokens, while keeping the same elements with the same order. The subsequences should include all the original tokens. The grouping can be used in the DP approach, by choosing a grouping in order to maximize the score of the retokenization of each group, and return the retokenized tokens $Q_1, Q_2, \cdots, Q_m$, where the retokenization is computed

Algorithm 10 Retokenization

```
function RETOKENIZE(t)
    w = t.word
    sp = t.split
    B := Array(|w| + 1, −∞)
    nx := Array(|w| + 1, |w| + 1)          ▷ Size |w| + 1, default value |w| + 1
    B[|w| + 1] := 0
    for i := |w| to 1 do
        for j := i + 1 to |w| + 1 do
            θ := (|w|−j+1)/(|w|−i+1)
            t := B[j] · θ + (1 − θ) · Score(w_[i,j))
            if t > B[i] then
                B[i] := t
                nx[i] := j
            end if
        end for
    end for
    res := []
    i = 1
    while i ≤ |w| do
        j := nx[i]
        if j ≤ |w| then
            res.APPEND(Token(w_[i,j), ' '))
        else
            res.APPEND(Token(w_[i,j), sp))
        end if
        i := j
    end while
    return B[1], res
end function
```

as in subsection 6.2.2.

The grouping problem can also be solved by dynamic programming. Using a function F, which operates on a state consisting of a single integer i where $1 \leq i \leq n+1$ encapsulating the suffix of tokens $T_i, T_{i+1}, \cdots, T_n$. The function F will be interpreted as the optimal score of grouping a suffix of tokens.

The heuristic used in the grouping, is that if there's a tokenization mistake that needs to be fixed, then the needed information for the fixing will be found in one of the neighboring tokens, or it won't be found at all. Therefore we will consider groups of small window size ω (which was set to 8 throughout the experiments).

The function F operates by choosing the size of the first group d, for the tokens $T_i, T_{i+1}, \cdots, T_{i+d-1}$, then try to retokenize the chosen group and solve the remaining suffix of tokens recursively using F. Additionally, there is a helper function G which tries to solve a sequence of tokens, by joining the tokens or by retokenizing them. This is given by the equation:

$$F[i] := \max_{1 \leq d \leq \omega, n+1-i} \{F[i+d] + G(i,d)\} \tag{15}$$

The helper function G takes as input two integers i, d, and it tries to solve the subsequence of tokens $T_i, T_{i+1}, \cdots, T_{i+d-1}$. It first tries to join them, and see if it's a valid word in the dictionary, and if it's not in the dictionary then it attempts to retokenize the joined words.

$$G(i,d) = \max \begin{cases} \text{Score}(t_i \circ t_{i+1} \circ \cdots \circ t_{i+d-1}) & \text{if not } 0 \\ \text{Retokenize}(t_i \circ t_{i+1} \circ \cdots \circ t_{i+d-1}) & \text{otherwise} \end{cases} \tag{16}$$

Using these equations together, we can construct the grouping step, and the whole DP approach as shown in algorithm 11, where the function F is computed in a bottom-up approach from smaller suffixes to longer ones. As a demonstration of the grouping and the DP approach, I will show an example of fixing the string 'He llowarl d td ay', which consists of 5 tokens. Ideally, this would be grouped as '(He llowarl d) (td ay)' and gets retokenized and fixed as 'Hello world today'. Table 5 shows how the helper function G is computed, and the corresponding retokenization or joined tokens, and also table 6 shows two sub-tables, the values of G and the values of the recursive function F (which shows the optimal scores of groups starting from a suffix of tokens). The cells marked in grey in the table of the function G demonstrate the positions where the text was not retokenized (the tokens are joined, and consequently, found in the dictionary). The cells marked in grey in the table of

the function F show the beginnings of the optimal groups. The value $\omega = 3$ is used in this example.

i	d	Tokens group	Retokenization	Joined tokens	Score $G(i, d)$
1	1	He	He	-	0.0
1	2	He, llowarl	Hello, war	-	21.21
1	3	He, llowarl, d	Hello, world	-	22.69
2	1	llowarl	l, to, war	-	6.49
2	2	llowarl, d	l, to, world	-	10.19
2	3	llowarl, d, td	l, to, world, to	-	8.72
3	1	d	d	-	0.0
3	2	d, td	d, to	-	1.93
3	3	d, td, ay	d, today	-	7.92
4	1	td	-	to	2.90
4	2	td, ay	-	today	9.90
5	1	ay	ay	-	0.0

Table 5: Helper function G values using 1st and 2nd layers. The inputs are i and d, which considers the tokens i to $i + d - 1$. The considered tokens are shown in the third column (comma separated), the fourth column has the retokenization of the considered tokens. The fifth column shows the joining of the considered tokens. The joining and retokenization are mutually exclusive, as shown and also as defined by the function G.

Algorithm 11 Dynamic Programming approach

```
function Fix(T)
    t := tokenize(T)
    F := Array(|t| + 1, −∞)
    tb := Array(|t| + 1, nil)
    F[|t| + 1] := 0
    for i := |t| to 1 do
        for j := 1 to min{ω, |t| + 1 − i} do
            G_ij, _ := Score(t_i ∘ t_{i+1} ∘ · · · ∘ t_{i+j−1})
            a := WHOLE
            if G_ij ≈ 0 then
                G_ij, _ := Retokenize(t_i ∘ t_{i+1} ∘ t_{i+j−1})
                a := RETOK
            end if
            if F[i + j] + G_ij ≥ F[i] then
                F[i] := F[i + j] + G_ij
                tb[i] := (j, a)
            end if
        end for
    end for
    i := 0
    res := []
    while i ≤ |t| do
        j, a := tb[i]
        if a = RETOK then
            _, ts := Retokenize(t_i ∘ t_{i+1} ∘ t_{i+j−1})
            res.extend(ts)
        else
            _, ts := Score(t_i ∘ t_{i+1} ∘ · · · ∘ t_{i+j−1})
            res.append(ts)
        end if
        i := i + j
    end while
    return join-tokens(res)
end function
```

		He	llowarl	d	td	ay
	G	1	2	3	4	5
He	1	0.0	21.21	22.69	-	-
	2	-	6.49	10.19	8.72	-
	3	-	-	0.0	1.93	7.92
td	4	-	-	-	2.90	9.90
	5	-	-	-	-	0.0

	He	llowarl	d	td	ay
i	1	2	3	4	5
F	32.6	20.1	9.9	9.9	0.0
$nxts$	4	4	4	6	6

Table 6: The right table shows the values of F and the grey cells mark the beginnings of the chosen groups, as computed by the traceback array $nxts$. The left table shows the values of G, and the grey cells are the ones chosen by F, which mark the endings of groups

7 Learning-based approaches

In this chapter, we will first explore the outline of a probabilistic fixer algorithm that utilizes beam search in order to find the best possible fixing. Followingly, details about specific common components will be explored, namely, going through the input processing for different models and the approaches attempted to solve the problem. There are mainly three approaches, the first one is using two separate character-based language models, one looking backward and one looking forward, then a combination of these two is tuned using an optimization model described in subsection 7.3.3, then this tuned combination is used to predict fixing decisions that fixes a given text, this combined model is described in section 7.3. The second approach is a baseline learning approach, which uses the first approach with replacing the language models by a simpler probabilistic model, based on n-Gram Markov models [30]. The third approach is a combination of all the components of the first approach in only one end-to-end recurrent neural network that is trained to fix texts.

7.1 Maximum likelihood sequence estimation

Given a model M that predicts the posterior probability of an element of a sequence given the prefix sequence before the predicted element, that is $p_M(A_n|A_1, \cdots, A_{n-1})$. We can estimate the most likely sequence S^* according to the model M, this sequence S^* is estimated by the formula:

$$S^* = \arg\max_A \{\prod_{i=1}^{n} p_M(A_i|A_1, \cdots, A_{i-1})\} \tag{17}$$

However, an alternative of this formula is used in practice, because this one is problematic as it often leads to too small numbers that are hard to compare, so we can use the logarithm function which is monotonically increasing, in order to

Algorithm 12 Beam search

```
function BEAM-SEARCH(S_0, δ, TERMINAL, B)
    Q := Queue()
    Q.PUSH((0, S_0))                                  ▷ (cost, state) pair
    R := ∞, nil
    while Q is not empty do
        P := PriorityQueue()
        while Q is not empty do
            (d, S) := Q.POP()
            if TERMINAL(S) and d < R.cost then    ▷ cost it the first element in R
                R := (d, S)
            else
                foreach (a, c) ∈ δ(S) do          ▷ (transition action, action cost)
                    U := UPDATE(S, a)
                    if u is not nil then
                        P.PUSH(d + c, U)  ▷ or normalized cost: d(1 − 1/|H_{|U|}|) + c·1/|H_{|U|}|
                    end if
                end for
            end if
        end while
        while P is not empty AND |Q| < B do          ▷ B is beam size
            t := P.POP()                      ▷ comparison is done by cost
            Q.PUSH(t)
        end while
    end while
    _, S_f := R
    return S_f
end function
```

reformulate the formula as:

$$\begin{aligned}
S^* = &\quad \arg\max_A\{\textstyle\sum_{i=1}^n \log p_M(A_i|A_1,\cdots,A_{i-1})\}) \\
\approx &\quad \arg\max_A\{\textstyle\sum_{i=1}^n \log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}\} \\
= &\quad \arg\min_A\{\textstyle\sum_{i=1}^n -\log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}\} \\
= &\quad \arg\min_A\{F_n(A)\} \\
\text{where } F_i(A) = &\quad F_{i-1}(A) - \log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}, F_0(A)=0
\end{aligned}$$

Additionally, we can also normalize the probabilities by the sequence length:

$$\begin{aligned}
S^* = &\quad \arg\max_A\{(\textstyle\prod_{i=1}^n p_M(A_i|A_1,\cdots,A_{i-1}))^{\frac{1}{n}}\} \\
= &\quad \arg\max_A\{\tfrac{1}{n}\textstyle\sum_{i=1}^n \log p_M(A_i|A_1,\cdots,A_{i-1})\}) \\
\approx &\quad \arg\max_A\{\tfrac{1}{n}\textstyle\sum_{i=1}^n \log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}\} \\
= &\quad \arg\min_A\{\tfrac{1}{n}\textstyle\sum_{i=1}^n -\log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}\} \\
= &\quad \arg\min_A\{G_n(A)\} \\
\text{where } G_i(A) = &\quad G_{i-1}(A)(1-\tfrac{1}{i}) - \tfrac{1}{i}\log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}, G_0(A)=0
\end{aligned}$$

Where $\epsilon \approx 10^{-30}$ is for smoothing. We can compute the optimal sequence S^* using a search algorithm, where the transition from the prefix $A_1, \cdots A_{i-1}$ to the prefix $A_1, \cdots, A_i$ is done by adding A_i with the cost $-\log\{p_M(A_i|A_1,\cdots,A_{i-1})+\epsilon\}$. In our scenario, we are trying to estimate an optimally fixed text by applying a sequence of fixing decisions, as shown by the *Apply* function defined in subsection 3.2.1. We will find an approximation of it using the beam search algorithm.

7.1.1 Beam search

Beam search is an approximation algorithm, that searches for a target state with the shortest path. However, since some state spaces could have a gigantic number of states and transitions, beam search uses a heuristic of limiting the exploration of considered states in order to reach a compromise between finding an optimal target state and not exploring the gigantic number of states. Beam search operates in a way similar to breadth-first search [13], which is expanding the states level by level, however the beam search limits the size of considered states in each level up to a fixed size [31], which is referred to as the beam size B. The states selected in each level are the states with the shortest paths from the starting state. Beam search is described in algorithm 12. The essential difference between the different approaches

(that will be introduced later) is how the transition function δ (including its costs) will be computed. A further note, if $B = 1$ then the beam search becomes a greedy search, and when $B \to \infty$ then the algorithm tends be a level-by-level breadth-first search. The beam search has shown promising results in a variety of applications like sequence-to-sequence based applications [12] and in speech recognition [32].

7.1.2 State space

The proposed approaches will use the same state space in the beam search, therefore it will be presented here. The function *update* is also common between approaches, and it is explained in the next subsection 7.1.3. Given a text T to be fixed, and assuming F is the resulting fixed text, we will construct states which are defined as 7-tuples $(B, v, A, i, R, added, U; T)$ where i is the index where the fixer is standing in order to make a fixing operation at T_i, B is a context string of the part before F_j (j is the index that aligns the F_j with T_i), A is a context string of the part after T_i, v is the character T_i, R is the so far accumulated fixed string, *added* is the number of added characters since the last fixing operation that is not a character addition, U is a list denoting the history of used fixing operations so far and T is an augmentation of the given text, T is not a part of the state, however it is a global constant for all the states. The initial state S_0 is defined as $(\dot{\$}^h, T_1, T_{2 \twoheadrightarrow h+1}, 1, \varepsilon, 0, [\,])$, where h is a fixed context length that will be used in the RNN models.

The fixing search goes on the text from beginning to end (left to right), and tries to fix the text on the way. The context B is already acquired from the fixed portion of the string, so it is preferred to be taken from the fixed text and not the corrupt text because it will make more accurate decisions later on. On the other hand, the after context A is acquired from the corrupt text. In subsection 7.2.2, a technique will be introduced, which is used to synthesize the training data in order to make the models more robust, since some predictions are made using the context A from the corrupt text and B from the (probably non-perfect) fixed text.

Additionally, the approaches operate by using/generating tuples where suffixes of T and F are aligned in some certain manner, where T_i is aligned with F_j for some j. The alignment concept might be unclear here, this concept will be explored in details in subsection 7.2.3, and how such alignments could be generated beforehand in order to get training data for fixing operations.

The variable *added* in the state is used to control bias that could happen by the fixer when it is biased or confused. The confusion can happen when all the decisions are so uncertain (uniformly distributed predictions), or when there is some bias in the

model, then this causes the fixer to start adding a bulk text that it thinks suitable in the given context, so it ends up adding a long fake text rather than actually making progress on fixing the given text, even if the actual fixing decision is not so certain at the given step. This ends up destructing the given text. Therefore, the variable *added* controls this behavior by putting an upper bound on the number of consequently added characters, therefore it only adds reasonable characters otherwise it ends up adding a small portion of fake text that makes the whole fixation unreasonable and hence gets eliminated early on during the beam search. During my experiments, I made an upper limit on *added* to be 3, which means that there should be no states explored if they have $added > 3$.

The variables R and U are used to accumulate the resulting fixed text and the history of taken actions, respectively. The fixed text F mentioned earlier is the result R of the final state. U can be used to trace the tokenization mistakes and the corresponding fixing actions taken.

Finally, I point out that the variables B, A, v are redundant because they all could be computed by $B = R_{-h\rightarrow}$, $A = T_{i+1\twoheadrightarrow i+h}$ and $v = T_i$, so they could be eliminated from the state, in order to simplify the state and save more memory in case the beam search has a large beam size or there are gigantic texts being fixed (which makes R and U already huge). However keeping them was helpful in debugging and visualizing the state, and they didn't consume so much memory during my experiments because they are additional $2h + 1$ characters, also the texts were not particularly long and I used a small beam size.

7.1.3 State updates

In the presented approaches that will be presented, they both will use beam search with the same state space and also the same possible actions outcome. Namely, the 4 basic edit operations ADD, CHG, DEL, and NOP. Each action applies an edit operation, and accordingly adjusts the contexts, the current character, and the accumulated resulting fixed string, then moves the fixing pointer i forward. The updates of a given state $(B, v, A, i, R, added, U; T)$ depending on the used action a (also demonstrated in figure 5) is computed by:

- (NOP, i, v) operation: Copy the character v as is, then move the fixing pointer forward, and as a result, append v it to the before context and resulting string, and update the after context, which results in the state:

$$(B \circ v, A_1, A_{2\rightarrow} \circ T_{i+h+1}, i + 1, R \circ v, 0, U \circ a; T)$$

Fix op	Before context	Current	After context	Fixed text
original:	$B = R_{-h\rightarrow}$	$v = T_i$	$A = T_{i+1\twoheadrightarrow i+h}$	R
NOCHG:	B v	A_1	$A_{2\rightarrow}$ T_{i+h+1}	R v
DEL:	B	A_1	$A_{2\rightarrow}$ T_{i+h+1}	R
CHG s:	B s	A_1	$A_{2\rightarrow}$ T_{i+h+1}	R s
ADD s:	B s	v	A	R s

Figure 5: The effects of all fixing operations NOP, CHG, DEL and ADD, on the updates of states. Namely, how the after context, before context, current character and resulting string are updated. The first row is the state before any update, the remaining rows are the updated state after using a fixing operation on the state.

- (DEL, i, v) operation: Delete the character v, and move the fixing pointer forward, and as a result, update the after context, which results in the state:

$$(B, A_1, A_{2\rightarrow} \circ T_{i+h+1}, i+1, R, 0, U \circ a; T)$$

- (CHG, i, s) operation: Change v to s, then move the fixing pointer forward, and as a result, append s to the before context and the resulting string, and adjust the after context accordingly, which results in the state:

$$(B \circ s, A_1, A_{2\rightarrow} \circ T_{i+h+1}, i+1, R \circ s, 0, U \circ a; T)$$

- (ADD, i, s) operation: Adds a character s before the character v, and as a result, append it to the before context and the resulting string, which results in the state:

$$(B \circ s, v, A, i, R \circ s, \mathit{added} + 1, U \circ a; T)$$

These updates are depicted in algorithm 13. If $i + h + 1 > |T|$, then we will assume that $T_{i+h+1} = \dot{\$}$, which is a padding from the end. A state is a terminal state if and only if the fixer pointer has finished the string $i > |T|$.

Algorithm 13 Update(S) function in the beam search algorithm

function UPDATE($S = (B, v, A, i, R, added, U; T), a$)
  $typ,, s := a$
  if typ = NOP **then**
    return $(B \circ v, A_1, A_{2\rightarrow} \circ T_{i+h+1}, i+1, R \circ v, 0, U \circ a; T)$
  end if
  if typ = DEL **then**
    return $(B, A_1, A_{2\rightarrow} \circ T_{i+h+1}, i+1, R, 0, U \circ a; T)$
  end if
  if typ = CHG **then**
    return $(B \circ s, A_1, A_{2\rightarrow} \circ T_{i+h+1}, i+1, R \circ s, 0, U \circ a; T)$
  end if
  if typ = ADD and $added + 1 \leq 3$ **then**
    return $(B \circ s, v, A, i, R \circ s, added + 1, U \circ a; T)$
  end if
  return nil
end function

7.2 Input processing

The datasets that were extracted consist of text files. However, we need a different format to be able to use it with the presented models. In addition to that, there is synthesization of the input data that needs to be considered, that makes the training data richer, and to generate more robust models, this is explained in subsection 7.2.2. Furthermore, the tuner in subsection 7.3.3 and the end-to-end approach in section 7.5 need a different input format than the other models, so I have to elaborate how this format is computed in subsection 7.2.3.

7.2.1 Input format for RNN models

The recurrent neural networks, as described in subsection 3.5.2, need as input a 3-dimensional tensor, that consists of the batch size (or number of examples), length of the input sequence and the number of possible values for each element in the sequence. The size of the first dimension can be left as unknown, which is decided on demand depending on the size of training data, the second dimension's size will be referred to as the context length h, and the third dimension's size is the size of the character-set Λ. The output of the RNN is a 2 dimensional tensor, which has as sizes the batch size and the character-set size respectively, the output for each example is a probability distribution over the values, denoting the probabilities of

the values following the given sequence. In practice, the input tensor $\mathbf{X}$ is stored in 2-dimensional format, because the third dimension is a one-hot vector, so the input is stored without applying one-hot on the 2nd dimension, and only do it for the training batch (which has much smaller size) on demand, in order to save RAM memory.

7.2.2 Input perturbation

The character-based language model is trained with a corpus of only correct texts, which could make the model sensitive if there are few mistakes in the given text. For example if we are given the two contexts 'he is going t o the stadiu', ' he is going to the stadiu', they both should predict that the next character is 'm' with high probability, the extra space in the word 'to' in the first example shouldn't significantly change the predicted probabilities because the context is still having the same content more or less.

The robustness against such minor mistakes is essential for making a strong fixing model because the fixing models will rely on the given corrupt text in order to fix it, which means that in some intermediate fixings, the model will use a context that contains some mistakes (from the corrupt text) in order to predict how to fix a certain part.

In order to overcome this issue, I used an idea introduced in stacked denoising autoencoders [33], which is including correct examples as well as some examples that have some random variation of the correct examples in order to make the neural network have more robust representation of the input examples, and therefore making more consistent and more robust predictions, even when some noise is introduced in the input context. These examples with random variations will be referred to as perturbed examples, and the number of perturbations is defined as the number of perturbed examples per each correct example in the training set.

A perturbation on a given context is generated by introducing a random corruption operation on the context, then flipping a biased coin with probability $p = 0.8$ to try making further perturbations or it halts. This behavior will make exactly t edit operations with probability $p^{t-1}(1-p)$, which generates an expected number of $\frac{1}{1-p} = \frac{\sum_{t>0} tp^{t-1}(1-p)}{\sum_{t>0} p^{t-1}(1-p)}$ edit operations in each perturbed example. Additionally, the corruption operations types DEL, CHG, ADD are chosen uniformly, however when it is CHG or ADD, the new character is not chosen uniformly, it is chosen depending on an approximation of how frequent the characters are in the datasets, in addition to extra manual changes, that increases the chances of the newline characters, and the pattern '-[newline]' particularly, in order to address the 3rd type of tokenizaiton

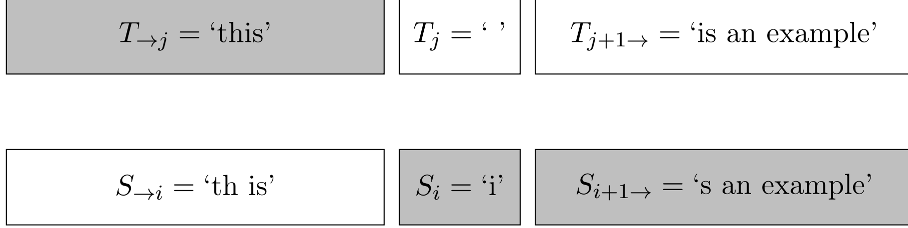

Figure 6: Alignment example, where $op = (6, 5, (ADD, 6, \text{' '}))$, $i = 6$, $j = 5$ S = 'th isis an example', T = 'this_is an example'. The alignment quadruple, with contexts of length 3, is: ('his', 'i', 's n', $(ADD, 6, \text{' '})$)

errors. Also it avoids introducing a perturbed context that contains two consequent delimiters. The chances of adding (or changing to) a space is 21.8%, a newline is 3.4%, a lower case character is 69.6%, an upper case character is 1.8%, a digit is 0.9% and a special character is 2.5%. Furthermore, the perturbations are generated in a sequence only in the portion that contains characters without the padding character $. As an example, given a correct example $$$Hello worl. → d, would be perturbed into $$He llo warl. → d or $$$$Hell worm. → d, but not $$Hello-[newline] worl. → d because it has two consequent delimiters, the space and newline.

7.2.3 Edit alignments

The two approaches that will be introduced will need an alignment of the correct and corrupt texts, and where exactly should a fixing operation be introduced. Both models will use states that have two contexts B and A and a character v, B is the context before v and A is the context after v. The model will have to predict the fixing operation a to do with the character v, in order to fix the text at the current state. Additionally, A will be a part of the corrupt text and B will be a part of the fixed text. In order to generate input data that matches this format, we will use the edit distance described in subsection 3.2.1 in order to generate such alignments. If we have a corrupt text S and correct text T, then the edits generated by $EditOperations(S, T)$ are on the form of triples (i, j, f), where f is the first edit (fixing) operation that transforms the suffix $S_{i\rightarrow}$ from the corrupt text to the suffix $T_{j\rightarrow}$ from the correct text. This triple aligns the positions in both strings for the fixing operation. Consequently, if the fixing pointer is pointing at i-th character (which is aligned with j in the correct text), from this we can obtain the before context from the correct text (because it is the ground truth of the fixed text) as $B = T_{\rightarrow j}$, the current character being fixed as $v = S_i$, the not-yet-fixed after context

$A = S_{i+1\to}$ after the character being currently fixed and $f = a$ is the ground truth fixing operation. Figure 6 demonstrates this alignment, where the grey parts are the parts that will be taken for alignment. Eventually, we can use all triples generated by the edit distance as the ground truth of fixing operations at all locations of corrupt and correct texts alignments. This will be used later in subsection 7.3.3 and section 7.5 as training data to optimize models for making good fixing operations. The output of edit alignments are quadruples on the form (B, v, A, f) which are given by the values $(T_{\to i}, S_i, S_{i+1\to}, f)$.

7.3 Bicontext model

The first approach used to solve the problem is the bicontext approach, which uses two separate character-based language models, implemented by recurrent neural networks, one to predict the successor character in a backward direction, and the other is predicting the successor character in a forward direction, then they are combined in order to decide how to fix the text. The combinations will predict a fixing operation, in order to fix the text. The model that predicts forward will be referred to by p_f and the model predicting backward will be referred to by p_b, as stated in subsection 3.5.2.

7.3.1 Long looking and occurrence functions

Using the two models, I construct $P_o(X, Y)$ which is defined as the expectancy of occurrence of the string Y after the string X. However, in order to compute it, we need to define two helper functions f and b. $f(X, Y, s)$ is defined as the expected length of a prefix from Y that comes after the context X, in other words, how likely the string $X \circ Y$ will occur in the language. The parameter s puts an upper bound on the prefix length from Y, since the change added is exponentially decaying in the length of the prefix, therefore limiting the prefix length to a small fixed length provides a good approximation. $b(X, Y, s)$ is defined in a similar manner, which is the expected length of a suffix from X that comes before the context Y. The parameter s limits the length of the suffix. The function f is therefore defined as:

$$f(X, Y, s) := \sum_{i=1}^{\min\{|Y|+1, s\}} p_f(Y_1, \cdots, Y_i | X) \tag{18}$$

where i is the length of the matched prefix, and the term $p_f(Y_1, \cdots, Y_i|X)$ is the probability that this prefix of length i is matched. Each of the terms in the summation

adds 1 expected character to the matched prefix. Similarly the function b is defined as:

$$b(X, Y, s) := \sum_{i=1}^{\min\{|X|+1, s\}} p_b(X_{-1}, \cdots, X_{-i}|Y) \tag{19}$$

We will assume if a predicted context is empty, then we are predicting the probability of having a padding character $\dot{\$}$, which will act like an 'end of text' character. This will be encountered when $|Y| < s$ for the function f and $|X| < s$ for the function b. The two functions can be also computed recursively as:

$$f_r(X, Y, s) = \begin{cases} 0 & \text{if } s \leq 0 \\ p_f(\dot{\$}, X) & \text{if } |Y| = 0 \text{ and } s > 0 \\ p_f(Y_1|X) \cdot (f_r(X \circ Y_1, Y_{2\rightarrow}, s-1) + 1) & \text{otherwise} \end{cases} \tag{20}$$

$$b_r(X, Y, s) = \begin{cases} 0 & \text{if } s \leq 0 \\ p_b(\dot{\$}, Y) & \text{if } |X| = 0 \text{ and } s > 0 \\ p_b(X_{|X|}|Y) \cdot (b_r(X_{\rightarrow|X|}, X_{|X|} \circ Y, s-1) + 1) & \text{otherwise} \end{cases} \tag{21}$$

Since both functions b_r and f_r have a similar recursive pattern, the claim will be proven on the function f, that $f_r = f$. There are two cases, if $|Y| = 0, s > 0$, then by definition $f(X, Y, s) = p_f(\dot{\$}, X) = f_r(X, Y, s)$. Otherwise, the claim is proven by induction on s:

Proof.

- **Induction basis:** $s = 0 \Rightarrow$ the function $f(X, Y, s) = 0 = f_r(X, Y, s)$.
- **Induction hypothesis:** $\forall k < s$ the recursive formula for f_r equals to the summation formula of f, that is $f(X, Y, k) = f_r(X, Y, k)$.
- **Induction step:** The induction step is divided into two cases $s > |Y| + 1$ and $s \leq |Y| + 1$. If $s > |Y| + 1$, then $s - 1 \geq |Y| + 1 > 0$, $s - 1 > |Y_{2\rightarrow}| + 1$ and

$s-2 \geq |Y_{2\rightarrow}|+1.$

$$
\begin{aligned}
f(X\circ Y_1, Y_{2\rightarrow}, s-1) &= \textstyle\sum_{i=1}^{|Y_{2\rightarrow}|+1} p_f(Y_2,\cdots,Y_i|X\circ Y_1) && \text{definition}\\
f(X\circ Y_1, Y_{2\rightarrow}, s-2) &= \textstyle\sum_{i=1}^{|Y_{2\rightarrow}|+1} p_f(Y_2,\cdots,Y_i|X\circ Y_1) && \text{definition}\\
&= f(X\circ Y_1, Y_{2\rightarrow}, s-1) &&\\
f(X,Y,s-1) &= \textstyle\sum_{i=1}^{|Y|+1} p_f(Y_1,\cdots,Y_i|X) && \text{definition}\\
f(X,Y,s) &= \textstyle\sum_{i=1}^{|Y|+1} p_f(Y_1,\cdots,Y_i|X) && \text{definition}\\
&= f(X,Y,s-1) &&\\
&= f_r(X,Y,s-1) && \text{ind hypothesis}\\
&= p_f(Y_1,X)\cdot(1+f_r(X\circ Y_1, Y_{2\rightarrow}, s-2)) && \text{definition}\\
&= p_f(Y_1,X)\cdot(1+f(X\circ Y_1, Y_{2\rightarrow}, s-2)) && \text{ind hypothesis}\\
&= p_f(Y_1,X)\cdot(1+f(X\circ Y_1, Y_{2\rightarrow}, s-1)) &&\\
&= p_f(Y_1,X)\cdot(1+f_r(X\circ Y_1, Y_{2\rightarrow}, s-1)) && \text{ind hypothesis}\\
&= f_r(X,Y,s) && \text{definition}
\end{aligned}
$$

otherwise if $s \leq |Y|+1$

$$
\begin{aligned}
f(X,Y,s) &= \textstyle\sum_{i=1}^{s} p_f(Y_1,\cdots,Y_i|X) && \text{definition}\\
&= p_f(Y_1|X)+\textstyle\sum_{i=2}^{s} p_f(Y_1,\cdots,Y_i|X) && \text{excluding first term}\\
&= p_f(Y_1|X)+\textstyle\sum_{i=2}^{s} p_f(Y_1|X)\cdot p_f(Y_2,\cdots,Y_i|X\circ Y_1) && \text{Bayes rule}\\
&= p_f(Y_1|X)+\textstyle\sum_{i=1}^{s-1} p_f(Y_1|X)\cdot p_f(Y_2,\cdots,Y_{i+1}|X\circ Y_1) && \text{shifting sum}\\
&= p_f(Y_1|X)\cdot(1+\textstyle\sum_{i=1}^{s-1} p_f(Y_2,\cdots,Y_{i+1}|X\circ Y_1)) && \text{factoring} P_f(Y_1|X)\\
&= p_f(Y_1|X)\cdot(1+f(X\circ Y_1, Y_{2\rightarrow s+1}, s-1)) && \text{definition}\\
&= p_f(Y_1|X)\cdot(1+f_r(X\circ Y_1, Y_{2\rightarrow s+1}, s-1)) && \text{ind. hypothesis}\\
&= f_r(X,Y,s) &&
\end{aligned}
$$

□

Using these 2 functions, the formula for P_o is then defined as:

$$P_o(X, Y) := \frac{b_r(X, Y, s)}{s} \cdot \frac{f_r(X, Y, s)}{s} \tag{22}$$

where s is a fixed constant, $s = 3$ was chosen through the experiments, because it provides a good approximation as well as not computing too many terms which can make the time performance worse.

The division by the term s for both functions b_r and f_r is to normalize the expressions to become probabilities between 0 and 1. However, as it will shown later in subsection 7.3.3, we will tune the probabilities P_o using linear functions on the logarithmic scale, therefore dividing P_o by the term s^2 is equivalent to adding a bias $\log s^{-2}$ on logarithmic scale, which will not matter because the tuning will also tune the bias values, as a consequence, a simpler expression will be used instead:

$$P_o(X, Y) := b_r(X, Y, s) \cdot f_r(X, Y, s) \tag{23}$$

The main advantage of the recursive formulas is caching some predictions of the RNN models; if we keep track of the states that call the transition function $\delta(S)$ during the beam search in a sequence $S_1, S_2, \cdots$, we will find that there is a lot of shared contexts between neighboring states. This enables using a cache of limited capacity (that drops the values, that are not recently used) for the predictions p_b and p_f values made by the RNNs, similar to the memoization in dynamic programming as explained in subsection 3.2.1, which allows a big speedup because the RNN model is one of the most computationally intense and most used operations in the fixer. The caching data structure is briefly explained in subsection 7.6.1. How these functions are used will be shown in the demonstration in subsection 7.3.5.

7.3.2 Fixing operations

Given the occurrence function P_o defined earlier, we can, therefore, construct probabilities for making different decisions, in order to compute the transition function $\delta(S)$ of a given state $S = (B, v, A, i, R, added, U; T)$:

- Copy / No-change fixing operation (NOP) is estimated by a combination of the two expressions $P_o(B \circ v, A)$ and $P_o(B, v \circ A)$.
- Delete fixing operation (DEL) is estimated by the value $P_o(B, A)$.
- Addition fixing operation (ADD, s) is estimated by a combination of the two

expressions $P_o(B \circ s, v \circ A)$ and $P_o(B, s \circ v \circ A)$, or 0 depending on certain conditions described in subsection 7.3.4

- Change fixing operation (CHG, s) is estimated by a combination of the two expressions $P_o(B \circ s, A)$ and $P_o(B, s \circ A)$.

It is important to consider the edited character v (or s in case of addition), in both given contexts B and A, because the prediction can significantly change if the edited character is in the input context or not. For example, to predict what comes after 'Hello ', it could be potentially a lot of possibilities, however predicting what comes after 'Hello W' narrows down the possibilities which makes more accurate decision. This applies to both language models. The demonstration in subsection 7.3.5 will show the effect of the values.

In the expressions above, that predict estimations by combining two terms, we can use a simple combination using geometric mean of the two values, or use a more sophisticated combination as described in subsection 7.3.3. After that, if the estimated probability for an action f is p, $\delta(S)$ is then computed as an enumeration of all pairs $(f, -\log\{p + \epsilon\})$, where $\epsilon \approx 10^{-30}$ is for smoothing to avoid computing $\log 0$. After a slight manual tuning, the combinations that lead to good results are:

$$
\begin{aligned}
\mathbb{P}_{\text{DEL},v\in\Gamma} &= P_o(B, A) \Rightarrow \log \mathbb{P} = \log P_o(B, A) \\
\mathbb{P}_{\text{DEL},v\notin\Gamma} &= 0.005 P_o(B, A) \Rightarrow \log \mathbb{P} = \log P_o(B, A) + \log 0.005 \\
\mathbb{P}_{\text{NOP}} &= \sqrt{P_o(Bv, A) \cdot P_o(B, vA)} \Rightarrow \log \mathbb{P} = \frac{1}{2} \log P_o(Bv, A) + \frac{1}{2} \log P_o(B, vA) \\
\mathbb{P}_{\text{ADD s},s\in\Gamma} &= \sqrt{P_o(Bs, vA)} \cdot P_o(B, svA) \Rightarrow \log \mathbb{P} = \frac{1}{2} \log P_o(Bs, vA) + \log P_o(B, svA) \\
\mathbb{P}_{\text{ADD s},s\notin\Gamma} &= 0.005^2 \sqrt{P_o(Bs, vA)} \cdot P_o(B, svA) \Rightarrow \\
&\log \mathbb{P} = \frac{1}{2} \log P_o(Bs, vA) + \log P_o(B, svA) + 2 \log 0.005
\end{aligned}
\tag{24}
$$

The resulting equations raised a motivation to develop an automatic tuning of the log probabilities obtained. By weighting every contributing factor and adding a bias value. This will be discussed in the decisions tuner subsection 7.3.3. The change (CHG) operations were not used during my experiments, because they made the time performance worse, and they can be replaced by a combination of DEL and ADD operations.

7.3.3 Decisions tuner

The decisions tuner is a simple optimization model that attempts to find the combinations of the probabilities of the fixing actions as described in subsection 7.3.2. Given a before context B, after context A and current character v, then the delete operation is decided by $P_o(B, A)$, the copy operation is decided by $P_o(B \circ v, A)$ and $P_o(B, v \circ A)$ and the addition operation is decided by $P_o(B \circ s, v \circ A)$ and $P_o(B, s \circ v \circ A)$. For each of the mentioned values, I made a distinction depending if the edited character v (or s in case of addition) is a delimiter character ($\in \Gamma$) or not, because the delimiter characters are much more frequent, and also they are essential to handle the first three types of tokenization mistakes. As a result, they could need more specific tuning than the tuning of the non-delimiter characters. For each of the probabilities p listed above, we will use the corresponding value $-w \cdot \log\{p + \epsilon\} - b$ as action cost in the beam search. There will be ten coefficients pairs (w, b) depending on which of the five values above is used and also if the edited character is a delimiter or not.

In order to make predictions and tune the model using the weights and biases pairs (w, b), we will formulate them in a matrix form, in order to easily formulate their tuning as a multi-classification task. The ten pairs will be listed in the two vectors $\mathbf{w}$ and $\mathbf{b}$. Furthermore, given an triple (B, v, A) of a before context, current character, after context and fixing action, we can then define a corresponding vector $\mathbf{q}$ of their fixing probabilities by:

$$\mathbf{q} := \begin{bmatrix} P_o(B, A) \cdot [v \in \Gamma] \\ P_o(B, A) \cdot [v \notin \Gamma] \\ P_o(Bv, A) \cdot [v \in \Gamma] \\ P_o(B, vA) \cdot [v \in \Gamma] \\ P_o(Bv, A) \cdot [v \notin \Gamma] \\ P_o(B, vA) \cdot [v \notin \Gamma] \\ P_o(Bs, vA) \cdot [s \in \Gamma] \\ P_o(B, svA) \cdot [s \in \Gamma] \\ P_o(Bs, vA) \cdot [s \notin \Gamma] \\ P_o(B, svA) \cdot [s \notin \Gamma] \end{bmatrix} \tag{25}$$

After that we can compute the tuned probabilities vector $\mathbf{d}$ by:

$$\mathbf{d} = \mathbf{M}^T (\mathbf{w} \odot \log\{\mathbf{q} + \epsilon\} + \mathbf{b}) \tag{26}$$

where

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{w} := \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \\ w_9 \\ w_{10} \end{bmatrix}, \mathbf{b} := \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \\ b_{10} \end{bmatrix} \tag{27}$$

which is expanded as:

$$\mathbf{d} := \begin{bmatrix} (w_1 \log\{P_o(B, A)[v \in \Gamma] + \epsilon\} + b_1) \\ (w_2 \log\{P_o(B, A)[v \notin \Gamma] + \epsilon\} + b_2) \\ (w_3 \log\{P_o(Bv, A)[v \in \Gamma] + \epsilon\} + w_4 \log\{P_o(B, vA)[v \in \Gamma] + \epsilon\} + b_3 + b_4) \\ (w_5 \log\{P_o(Bv, A)[v \notin \Gamma] + \epsilon\} + w_6 \log\{P_o(B, vA)[v \notin \Gamma] + \epsilon\} + b_5 + b_6) \\ (w_7 \log\{P_o(Bs, vA)[s \in \Gamma] + \epsilon\} + w_8 \log\{P_o(B, svA)[s \in \Gamma] + \epsilon\} + b_7 + b_8) \\ (w_9 \log\{P_o(Bs, vA)[s \notin \Gamma] + \epsilon\} + w_{10} \log\{P_o(B, svA)[s \notin \Gamma] + \epsilon\} + b_9 + b_{10}) \end{bmatrix} \tag{28}$$

Where the elements of $\mathbf{d}$ correspond to the tuned log probabilities of the decisions delete delimiter, delete non-delimiter, copy delimiter, copy non-delimiter, add delimiter and add non-delimiter respectively. We can use the values depending on the category of the decision f we need to use, as shown in algorithm 14.

As a result, we can compute the transition function $\delta(S)$ as an enumeration of delete, copy, add-delimiter, add non-delimiter s_b and add non-delimiter s_f, where $s_b = \arg\max_s p_b(s|A_{|S})$ and $s_f = \arg\max_s p_f(s|B_{|S})$, with the corresponding tuned log probability predicted scores from the vector $\mathbf{d}$. Trying characters other than s_f and s_b will lead to more possibilities, which might improve the results, however, we didn't try that because it makes the model computationally intensive. The steps of computing the transition function $\delta(S)$ are depicted in algorithm 14.

Tuner training

In order to get training data to tune $\mathbf{w}, \mathbf{b}$, we will use the quadruples extracted by edit alignments, as described in subsection 7.2.3, as the ground truth data. Depending

on a given quadruple (B, v, A, f), we will construct the corresponding vector $\tilde{\mathbf{d}}$ as a one-hot vector for the decision f over the set of the six possible decisions mentioned earlier, and also we construct $\tilde{\mathbf{q}}$, by the equation:

$$\tilde{\mathbf{q}} := \begin{bmatrix} P_o(B,A)\cdot[v\in\Gamma] \\ P_o(B,A)\cdot[v\notin\Gamma] \\ P_o(Bv,A)\cdot[v\in\Gamma] \\ P_o(B,vA)\cdot[v\in\Gamma] \\ P_o(Bv,A)\cdot[v\notin\Gamma] \\ P_o(B,vA)\cdot[v\notin\Gamma] \\ \max_{s\in\Gamma} P_o(Bs,vA) \\ \max_{s\in\Gamma} P_o(B,svA) \\ 1-(1-\max_{s^*\in\{s_f,s_b\}\setminus\Gamma} P_o(Bs^*,vA))\cdot[(f=(\text{ADD},s) \text{ and } s\notin\Gamma) \text{ or } \forall s\cdot f\neq(\text{ADD},s)] \\ 1-(1-\max_{s^*\in\{s_f,s_b\}\setminus\Gamma} P_o(B,s^*vA))\cdot[(f=(\text{ADD},s) \text{ and } s\notin\Gamma) \text{ or } \forall s\cdot f\neq(\text{ADD},s)] \end{bmatrix} \tag{29}$$

Tuning $\mathbf{w}$ and $\mathbf{b}$ is achieved by training a multi-class classifier, using *categorical cross-entropy* as the loss function, that compares between the predicted classes $\text{softmax}(\mathbf{M}^T(\mathbf{w}\odot\log\{\tilde{\mathbf{q}}+\epsilon\}+\mathbf{b}))$ and the ground truth output $\tilde{\mathbf{d}}$. It is important to note that the multi-classification task will maximize the weights corresponding to the probabilities of the ground truth action. This is achieved by tuning w, b to maximize particular values in the vector $\mathbf{M}^T(\mathbf{w}\odot\log\{\tilde{\mathbf{q}}+\epsilon\}+\mathbf{b})$, which is equivalent to minimizing the corresponding values in $-\mathbf{M}^T(\mathbf{w}\odot\log\{\tilde{\mathbf{q}}+\epsilon\}+\mathbf{b})$, which will be the returned costs for the fixing actions during the beam search. This assists the beam search to assign the correct fixing actions as the best steps (with the least costs) leading to the approximated shortest path.

The values of $\tilde{\mathbf{q}}$ are the same as $\mathbf{q}$ except the last two elements, they are set to be the maximum possible value ($= 1.0$) in case $f = (\text{ADD}, s)$, and the corresponding character s^* with maximum score is not the same as s (s^* is different for each term). The reason for this is no matter how we tune the weights, we will never get a combination that can classify the correct character to add, because the wrong character s^* will always have better score, and hence the weights of this example should be intentionally punished in order to avoid being wrongly tuned. Similar to the predictions part earlier, we will also try the non-delimiters that are top predicted characters s_f and b_f by p_f and p_b respectively, in order to save computation time.

Futhermore, in case the tuner needs to be disabled, the weights and biases vectors $\mathbf{w}$ and $\mathbf{b}$ can be set to default values, which will make the generated values identical to the values obtained by manual tuning in equation 24. The default values will be

given by:

$$\mathbf{w} := \begin{bmatrix} 1 \\ 1 \\ 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \\ 1 \\ 0.5 \\ 1 \end{bmatrix}, \mathbf{b} := \begin{bmatrix} 0 \\ \log 0.005 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \log 0.005 \\ \log 0.005 \end{bmatrix} \tag{30}$$

7.3.4 Look-forward in additions

The addition fixing operation particularly is handled with additional technique, which is looking forward. One of the concerns, that were observed when using an addition fixing operation, is that the bicontext model tends to use the same addition operation in an earlier position. For example, a text like 'Helloworld' which needs the fixing operation (ADD, ' ') at position 6, when is being fixed from left to right, the bicontext model would predict that there should be a space to add when the fixer is still at position 4, in other words the fixer at the earlier position 4 finds out that to add a space is better than just copying or deleting the character at position 4, so it ends up introducing a wrong fixing 'Hel loworld' and consequently, it decides that adding a space at position 6 is not the best choice, so we end up having a bad fixing, because the correct fixing operation was introduced few positions earlier.

However, another observation was found, that whenever a fixing operation is introduced in an earlier position and then we make a look-forward and try to add the same character after one or two positions, this forward addition will have higher score. Consequently, whenever these scores are found, the addition score at the current position is punished strongly to be 0 (or the smoothed $\log \epsilon$, in case of log probabilities), which prevents the fixer from adding fixings in earlier positions than where they actually should be made. This is depicted in the function "Look-Forward" in the algorithm 14.

To demonstrate, when the fixer is standing at position 4 in the given example 'Helloworld', and finds that to add a space at 4 with score 0.1, and we look forward and try to add the space instead at position 5 with score 0.05 and at position 6 with

Algorithm 14 Bicontext tuned transition function $\delta(S)$

```
function BICONTEXT-TRANSITION(S = (B, v, A, i, R, added, U; T))
    if v ∈ Γ then
        c_DEL = w_1 log{P_o(B, A) + ε} + b_1
        yield ((DEL, i, v), c_DEL)
        c_NOP = w_3 log{P_o(Bv, A) + ε} + w_4 log{P_o(B, vA) + ε} + b_3 + b_4
        yield ((NOP, i, v), c_NOP)
    else
        c_DEL = w_2 log{P_o(B, A) + ε} + b_2
        yield ((DEL, i, v), c_DEL)
        c_NOP = w_5 log{P_o(Bv, A) + ε} + w_6 log{P_o(B, vA) + ε} + b_5 + b_6
        yield ((NOP, i, v), c_NOP)
    end if
    S := {s_b, s_f, ' '}
    foreach s ∈ S do
        if s ∈ Γ then
            c_ADD = w_7 log{P_o(Bs, vA) + ε} + w_8 log{P_o(B, svA) + ε} + b_7 + b_8
            c_ADD := LOOK-FORWARD(c_ADD, B, A, v, s)
            yield ((ADD, i, s), c_ADD)
        else
            c_ADD = w_9 log{P_o(Bs, vA) + ε} + w_10 log{P_o(B, svA) + ε} + b_9 + b_10
            c_ADD := LOOK-FORWARD(c_ADD, B, A, v, s)
            yield ((ADD, i, s), c_ADD)
        end if
    end for
end function
function LOOK-FORWARD(c, B, A, v, s)
    if c < log{max_{1≤i≤2}{P_o(B, A_{↠i} s v A_{i+1→})} + ε} then
        return log ε
    else
        return c
    end if
end function
```

score 0.4, then we will punish the 0.1 to 0, because adding the character in 2 positions later has better score. This will go on and it should be the case that at position 6, it will have a higher score than looking forward further after position 6, and hence apply the addition operation in the right place.

7.3.5 Wrapping up

Algorithm 15 Bicontext approach

global h ▷ History length
global p_f ▷ Forward character-based language model
global p_b ▷ Backward character-based language model
global use BICONTEXT-TRANSITION as δ ▷ Use bicontext's transition function
global function P_o ▷ Occurence function
global $w_1, \cdots, w_{10}, b_1, \cdots, b_{10}$ ▷ Tuned weights and biases
global $B = 2$ ▷ Beam size
function FIX(T)
    terminal $:= \lambda S : S_{|i} > |T|$ ▷ Terminal state if the text is processed
    $S_0 := (\dot{\$}^h, T_1, T_{2 \twoheadrightarrow h+1}, 1, \varepsilon, 0, [\,]; T),$
    $S_f :=$ BEAM-SEARCH($S_0, \delta,$ **terminal**$, B$)
    return $S_f|_R$ ▷ Retrieve the resulting string R from the state
end function

In order to wrap up all the components, algorithm 15 shows the final presented "Fix" function of the bicontext approach. Additionally, to demonstrate several components of the approach, two snapshots are shown during the fixing of an article from the Simple-Wikipedia dataset with the title "1976 Summer Paralympics". Similar to section 4.3, the first text is the correct text, with markers which types of corruptions are used, and the second text is the resulting corrupt text which will be fixed.

'The 1976 Summer[2] Paralympics[1] took[2] place[2] in Toronto, Ontario, Canada. 1,657 athletes from[2] 38[1] were[1] at[4] the[2] Games. People[2] with these[1] types of[4] disabilities[1] competed at the games[2]: spinal[2] injury[2], amputee, blindness, and[1] Les Autres.'

'The 1976 Summe r Paraly mpicstook pla ce in Toronto, Ontario, Canada. 1,657 athletes fro m 3awereat th e Games. Pe ople with thesetypes f disabilitiescompeted at the gam es: sp inal i njury, amputee, blindness, andLes Autres.'

This text was almost totally fixed, except for the token '38' wasn't recovered. The token was stroke out in the fixed text for demonstration, it is not actually there though. The results of the evaluation metrics precision, recall and F_1-score are $0.895, 0.944, 0.919$ respectively. The resulting fixed text is:

'The 1976 Summer Paralympics took place in Toronto, Ontario, Canada. 1,657 athletes from ~~38~~ were at the Games. People with these types of disabilities competed at the games: spinal injury, amputee, blindness, and Les Autres.'

X	Y	$f(X,Y,3)$	$b(X,Y,3)$	$P_o(X,Y)$	$\log P_o$	$-w \log P_o(X,Y) - b$
B_1	A_1	10^{-6}	$6 \cdot 10^{-6}$	10^{-10}	-23.68	$-0.45 * -23.68 - 0.88 = 13.35$
B_1v_1	A_1	0.004	0.028	10^{-4}	-9.09	$-0.22 * -9.09 - 1.89 = 0.11$
B_1	v_1A_1	0.014	0.0033	$4 \cdot 10^{-6}$	-10.00	$-0.76 * -10.00 - 1.89 = 5.71$
B_1s_1	v_1A_1	1.98	2.06	4.08	1.41	$-1.05 * 1.41 + 0.37 = -1.11$
B_1	$s_1v_1A_1$	2.59	1.33	3.44	1.24	$-1.03 * 1.24 + 0.37 = -0.91$
B_2	A_2	0.53	1.55	0.82	-0.19	$-0.45 * -0.19 - 0.88 = -0.79$
B_2v_2	A_2	0.06	0.0007	$4 \cdot 10^{-5}$	-10.11	$-0.47 * -10.11 - 2.57 = 2.18$
B_2	v_2A_2	0.001	0.058	$8 \cdot 10^{-5}$	-9.43	$-0.76 * -9.43 - 2.55 = 4.61$
B_2s_2	v_2A_2	0.87	0.26	0.22	-1.49	$-1.05 * -1.49 + 0.37 = 1.93$
B_2	$s_2v_2A_2$	0.63	0.60	0.38	-0.97	$-1.03 * -0.97 + 0.37 = 1.37$

Table 7: Two snapshots during the execution of the bicontext fixer. The two snapshots are underlined in the corrupt text. The first has the contexts: B_1 = 'le with these types ', v_1 = 'f', s_1 = 'o', A_1 = ' disabilitiescompete', and the second has the contexts: B_2 = ' the games: spinal i', v_2 = ' ', s_2 = 's', A_2 = 'njury, amputee, blin'. In the first snapshot, $\mathbf{d}_{\text{DEL}} = 13.353$, $\mathbf{d}_{\text{NOP}} = 5.914$, $\mathbf{d}_{\text{ADD, o}} = -1.994$, and in the second snapshot $\mathbf{d}_{\text{DEL}} = -0.792$, $\mathbf{d}_{\text{NOP}} = 6.781$, $\mathbf{d}_{\text{ADD, s}} = 3.295$. Which predicts the correct decisions in both scenarios, with minimal scored as designed.

Table 7 shows two snapshots of the values of the occurrence function, the functions f, b, log probabilities and weighted log probabilities; the snapshots are at two points which are underlined in the corrupt text. The weights vector that was used in the example is: $\mathbf{w} = [0.45, 0.40, 0.47, 0.76, 0.22, 0.76, 0.81, 0.72, 1.05, 1.03]^T$ and the corresponding bias vector is: $\mathbf{b} = [0.88, -3.90, 2.57, 2.55, 1.89, 1.89, 0.41, 0.41, -0.37, -0.37]^T$. In the first snapshot, the values are obtained by $\mathbf{d}_{\text{NOP}} = 5.914 \approx 5.71 + 0.11$ and $\mathbf{d}_{\text{ADD, o}} = -1.994 \approx -0.91 + -1.11$, and similarly in the second snapshot: $\mathbf{d}_{\text{NOP}} = 6.781 \approx 2.18 + 4.61$ and $\mathbf{d}_{\text{ADD, s}} = 3.295 \approx 1.93 + 1.37$. Furthermore, to give

an intuition about the character-based language model, I list the top 3 characters predicted (with corresponding probabilities), in the four scenarios:

- p_b(' disabilitiescompete') predicts $e : 0.17, s : 0.13, n : 0.09$.
- p_f('le with these types ') predicts $o : 0.88, a : 0.02, i : 0.02$.
- p_b('njury, amputee, blin') predicts $i : 0.79, e : 0.09, a : 0.06$.
- p_f(' the games: spinal i') predicts $n : 0.53, s : 0.34, m : 0.03$.

7.4 Baseline 3-Gram Markov model

After introducing the bicontext approach, a baseline learning approach can be presented. The baseline learning approach is the same as the bicontext approach, without weights tuning and input perturbation. The main difference is replacing the character-based language models by a simple implementation of n-Gram Markov model [30]. The joint probability of predicting a character v to come after a context C can be given by the equation:

$$p(v|C) = \frac{\text{count}(C \circ v)}{\text{count}(C) + \epsilon} \tag{31}$$

Where $\epsilon \approx 10^{-10}$ is for smoothing. The function 'count' is the count of the given context string in the training corpus. The contexts C considered are 3-grams, meaning that they are contexts of length 3. Furthermore, this model could be improved using smoothing techniques as shown in [30, 34], however, they were not used in our experiments because this was aimed to be for baseline comparison only.

7.5 End-to-end model

The final approach for solving the given problem is using an end-to-end deep learning approach. This approach is mixing all the different components of the first approach into one recurrent neural network, this model consists of input that concatenates the before context B, current character v and after context A into an input sequence X, then the output of the model is probability distribution over the list of possible fixing decisions $[\text{DEL}, \text{NOP}, \text{ADD } \Lambda_1, \cdots, \text{ADD } \Lambda_{|\Lambda|}, \text{CHG } \Lambda_1, \cdots, \text{CHG } \Lambda_{|\Lambda|}]$. We can also include perturbed examples by perturbing the two contexts B and A as in the bicontext model. Consequently, we can compute the transition function

$\delta(S)$, for a given state $S = (B, v, A, i, R, added, U; T)$, as an enumeration of all pairs $(-\log\{F(B, v, A) + \epsilon\}, f)$ for all possible decisions f, where $F(B, v, A)$ is the prediction of the model on the $B \circ v \circ A$ of the given state S, and $\epsilon \approx 10^{-30}$ is for smoothing 0. The training data for this model are obtained directly by the edit alignments described in subsection 7.2.3, similar to how it was obtained for the tuner in subsection 7.3.3. The model is trained as a multi-classification task, that tries to classify the fixing actions to apply.

The main advantages of this model are its ultimate simplicity, very efficient computation of the transition function $\delta(S)$ (it's computed by making only one time prediction using the RNN model) and utilizing deep learning to give more capacity to optimize how the components are connected together instead of making these connections manually and tuning the connections in a shallow sense as in the bicontext approach. However the main disadvantage of this model is its need for a much bigger amount of training data because of its tendency to underfit due to the strong unbalance of fixing decisions in the training data. The decisions NOP, DEL, (ADD, ' ') (adding a space) are much more dominant in training set than the remaining decisions. This issue was addressed by weighting the classes in the cost function, by giving significantly lower weights to dominant classes in the training set, which ends up having the effect of undersampling the dominant classes.

7.6 Utility

7.6.1 Caching

Caching is a strategy that was used in order to make the computations of the predictions of the RNN models more efficient, since most predictions made in the bicontext fixer are computed few times, consequently, if the computed predictions are cached properly then they will achieve a speedup. On the other hand, there is a numerous number of possible prediction inputs (context string with length between $20 \sim 100$), so it would consume a lot of resources to cache all the input strings encountered, resources like memory and query time if too many predictions are cached, therefore we will put a fixed upper bound on the size of the cache.

In addition to that, if we consider, for example, the predictions made by the bicontext fixer, in order to compute $P_o(B, xyzA)$, we will compute the values $p_f(B)$, $P_f(Bx)$, $P_f(Bxy)$ using the RNN, then in the next step while computing $P_o(Bx, yzA)$, the values $p_f(Bx), p_f(Bxy), p_f(Bxyz)$ will be computed, and while computing $P_o(Bxy, zA)$, the values $p_f(Bxy), p_f(Bxyz)$ will also be computer. As

observed, there are many shared computations between nearby states, therefore the values that are frequently used are recently computed values. From the mentioned pieces of information, I made a caching data structure with a fixed size, that occasionally gets cleared except for the recently added or used values.

The data structure is initialized with a variable S, in order to construct the history of the cache with size $2S$. Whenever we add or search a value, we mark the most recent time id used of the query key, in order to be able to identify if it's recently used or not. When the cache is full after an addition, then all the entries that were not recently used are removed from the cache, and only up to S elements remain. This is depicted in algorithm 16.

Algorithm 16 Caching data structure

```
procedure CACHER(S)
    D := HASH-TABLE()                                    ▷ Cached values
    H := QUEUE()                                         ▷ History
    I := HASH-TABLE()                                    ▷ most recent IDs
    function ADD-VALUE(k, V)
        D[k] := V
        id := |I|
        I[k] := id                                       ▷ Update most recent use
        H.PUSH-BACK(k)
        CLEAR-OLD( )
    end function
    function GET-VALUE(k)
        if k ∈ D then
            V := D[k]
            ADD-VALUE(k, v)                              ▷ To mark the most recent use
            return V
        else
            return nil
        end if
    end function
    function CLEAR-OLD( )
        if |H| > 2S then                                 ▷ Only clear if the cache is full
            for k ∈ H_{1↠S} do
                if k ∈ D and I[k] < S then               ▷ Most recent retrieval is old
                    delete D[k]
                end if
            end for
            while |H| > S do
                H.POP-FRONT()                            ▷ Remove all non-recent entries
            end while
            I.CLEAR()
            for i, k ∈ ENUMERATE(H) do
                I[k] := i                                ▷ Rename the time id's
            end for
        end if
    end function
end procedure
```

8 Experiments

In this chapter, we present an empirical evaluation of the models described throughout the thesis, along with variations of some components. In summary, we have five different models to compare, two of them are non-learning dictionary-based and three are deep learning based. Since the bicontext RNN approach is the main proposed method to solve the given problem, there will be further experiments of different variants of this model; the variants include RNN architecture, context length h, number of layers, reversing the input context, different beam sizes, not using the decisions tuner and using less trained models.

There are mainly two perspectives to compare the models from; the most essential perspective is the one stated in the formal problem definition at the end of section 5.1, namely the fixing performance as computed by the F_1-score. The second perspective is only for the bicontext RNN model, which is comparing the train/test losses and metrics of the character-based language models, and if it affects the corresponding fixing performance.

8.1 Experiments specifications

Implementation specifications

The code was purely implemented in Python 3, using the deep learning framework Keras (with tensorflow as backend) for the RNN models, because Keras provides simple-yet-efficient code for deep learning; furthermore, the weights optimization in subsection 7.3.3 was implemented using the framework tensorflow, because the matrix form was ideal for using tensorflow. Additionally, the libraries NumPy and scikit-learn were used for data containers and basic mathematical functions.

Datasets specifications

The evaluations were made on two datasets, a sample of 10,000 articles from the Simple-Wikipedia dataset and a sample of 1,500 articles from the Reuters-21578 dataset were used. Then they were carried out using K-fold cross validation (with

$K = 3$ folds) [35], which divides the files into 3 parts, each one is used once as test-data and the remaining files are used for training. Each dataset has its own dictionary of words (that have frequency > 2). The corrupt versions of the texts were created using the corruptor algorithm 7 with corruption rate $p = 0.5$, and tokenization errors rates $4:4:1:2$; another corrupt versions were created using only the first 3 types of tokenization errors (no typos/garbled characters) with rates $4:4:1$. For each of those, the result of detailed edit operations (for edit alignments), as computed by algorithm 4, was dumped into a Python pickle file, in order to save running time to compute them in every experiment.

Dictionary-based approaches specifications

In the both dicionary-based approaches, a Trie dictionary was used (with words depending on the dataset's dictionary). The dictionary had mismatch damping factor $\varphi = 0.5$ and upper limit on the edit distance to be 1 (it doesn't partially match a word if the edit distance is > 1). Additionally, if a word w is matched in the Trie dictionary, then the matching score will be slightly decreased by the relative frequency of w in the dictionary, therefore the score will be $f_w^{0.05}$ (which is mostly > 0.5) instead of 1. In the dynamic programming (DP) approach, the grouping window size ω was set to 8, and the parameters $\alpha = 1.15, \beta = 0.1, \gamma = 1.0, \zeta = 2.0$ were used.

Bicontext model specifications

The bicontext model is using two RNN character-based language models, forward and backward, each model consists of one or two RNN layers of certain size and history length, followed by a dropout layer with drop-rate 0.2, followed by a final fully-connected layer with softmax activation function, in order to predict the character, like figure 3. Additionally, in a different model like the sequence-to-sequence (seq2seq) [36], reversing the input sequence enhanced the results of their predictions, and I thought to experiment this idea as well. The character-set I used had nearly 100 characters, consisting from lower case alphabet, upper case alphabet, digits, special characters, delimiter characters, an "unknown" character and a padding character. The model was used with beam search size 2.

Decisions tuner in bicontext model specifications

A decisions tuner was started for each fixing evaluation experiment, it constructs the training input from the edit alignments of 500 training files for the Simple-Wikipedia

dataset and 100 training files for the Reuters-21578 dataset. Then, the corresponding multi-classification task is trained using Adam optimizer [20], for 10,000 epochs using a learning rate 0.01. After that, the weights are saved and reloaded during evaluations.

End-to-end model specifications

The end-to-end model is a character-based RNN that predicts how to fix a text, the architecture of the model consists of a bidirectional LSTM layer (which performs similar to two LSTM layers), followed by a dropout layer of drop-rate 0.5, followed by a fully connected layer with softmax as activation function to classify which fixing decision should be made, additionally, the weight matrices of the LSTMs were also dropped out with drop-rate 0.5. The model uses an input as the concatenation $B \circ v \circ A$ (of length $2h + 1$).

Models notations

Since there are a lot of RNN models used in the experiments for the bicontext approach, a short notation for the models had to be presented. The short name will be a concatenation of the RNN architecture, hidden units size, history size, number of layers and a flag N or R to indicate if the input sequence is reversed or not. For example, GRU256H40L2R is a bicontext model with 2-layered GRU recurrent neural networks, with 256 hidden units, context length 40 and the input sequence is reversed; LSTM128H20L2R+N is a bicontext model with 2-layered LSTM recurrent neural networks, with 128 hidden units, context length 20 and the input of the backward model is reversed and the forward model is not reversed.

Character-based models training specifications

The training was done with input perturbation, as described in subsection 7.2.2, with 2 perturbed examples for each correct example. The training was done for 100 epochs, and the perturbations were randomly regenerated every 10 epochs in order not to overfit over them. Additionally, also each 10 epochs, the training data was augmented with a space-separated concatenation of 30,000 randomly chosen words from the dataset's dictionary. For the end-to-end bidirectional model there was 7 perturbed examples instead of 2, because this model was suffering from underfitting. All the 7 perturbed examples had perturbed after context, but only 2 of them had perturbed before context, in order to mimic a fixed text. The training was made using a GPU (Titan X), while using a large batch size of 8,192. The Simple-Wikipedia

dataset after adding the perturbed examples, had around 17,500,000 inputs examples, which was around 2,100 batches, on the other hand, the Reuters-21578 dataset had around 3,000,000 input examples, which was around 370 batches. For the 2-layered architectures, with history 20 and 128 hidden units, the training epoch on the GPU took around 8 minutes for the Simple-Wikipedia dataset and around 90 seconds for the Reuters-21578 dataset. Furthermore, the number of parameters to be tuned in each architecture are given by:

- The 1-layered LSTM128 had around 130,000 parameters.
- The 2-layered LSTM128 had around 260,000 parameters.
- The 2-layered LSTM256 had around 780,000 parameters.
- The 1-layered GRU128 had around 100,000 parameters.
- The 2-layered GRU128 had around 200,000 parameters.

Fixings evaluation specifications

The evaluation of the fixings of different models was ran on a subset of 1,400 files from the test fold, which is half of the test fold in the Simple-Wikipedia dataset, and the whole test fold in the Reuters-21578 dataset. The evaluation was ran using multiprocessing of ~ 8 parallel processes (or ~ 23 in a bigger cluster), in order to evaluate multiple files at the same time.

8.2 Character-based language models evaluation

In order to understand the relation between the character-based language model's accuracy and its fixing performance, I evaluated the different architectures using the training/test data, but the evaluation is done without the perturbed training examples, because they were changed frequently during training.

The models are evaluated using three metrics, the first is the optimized cost/loss (L) function *categorical cross-entropy*, the second is the accuracy (A) which is the ratio of examples that the true class is predicted with the highest probability, the third metric is the top-5-categorical which is ratio of examples that the true class is predicted in the top 5 predicted categories.

In table 8, the evaluation metrics of the train and test sets are shown, the evaluation was done on all the RNN architectures using the Simple-Wikipedia dataset, and similarly table 9 is for the Reuters-21578 dataset.

Dir	Model	train L	train A	train T5	test L	test A	test T5
back	BiLSTM128H20L1R	1.461	0.575	0.846	1.503	0.565	0.838
back	GRU128H20L1R	1.580	0.546	0.828	1.607	0.54	0.822
back	GRU128H20L2R	**1.423**	**0.588**	**0.853**	**1.469**	**0.576**	**0.844**
back	LSTM128H20L1N	1.667	0.524	0.816	1.691	0.519	0.812
back	LSTM128H20L1R	1.569	0.547	0.828	1.597	0.542	0.823
back	LSTM128H20L2N	1.519	0.562	0.838	1.555	0.554	0.831
back	LSTM128H20L2R	**1.407**	**0.587**	**0.853**	**1.454**	**0.577**	**0.844**
forw	BiLSTM128H20L1R	1.46	0.577	0.848	1.504	0.566	0.84
forw	GRU128H20L1R	1.688	0.522	0.815	1.709	0.516	0.811
forw	GRU128H20L2R	1.541	0.562	0.838	1.574	0.553	0.832
forw	LSTM128H20L1N	1.563	0.55	0.833	1.591	0.543	0.828
forw	LSTM128H20L1R	1.668	0.523	0.818	1.691	0.517	0.814
forw	LSTM128H20L2N	**1.403**	**0.59**	**0.856**	**1.45**	**0.579**	**0.847**
forw	LSTM128H20L2R	1.530	0.56	0.839	1.565	0.551	0.832

Table 8: Simple-Wikipedia train/test loss and accuracies. L is the categorical-cross entropy, A is the accuracy and T5 is the top-5-categorical accuracy. The shown results are the mean of the 3-fold cross validated metrics.

We can conclude from the evaluations that, by comparing LSTM128H20L1N, LSTM128H20L2N against LSTM128H20L1R, LSTM128H20L2R, RNNs are better to predict using the original order of sequence, that is predicting a_t after the sequence $a_1, \cdots, a_{t-1}$ is better than predicting it after the sequence $a_{t-1}, \cdots, a_1$, since all the forward models got higher score when the input's natural English order was not reversed. Additionally, the backward models also confirm this conclusion, because their non-reversed input is the natural English order, so they also prefer that the sequence is not reversed in the prediction direction (which corresponding to reversing the input's given natural English order). For example, to predict what comes before 'ello', the backward model yielded better performance when the input is fed as 'olle' and not 'ello'. The sequence (in the prediction direction) is given by $a_1 =$ 'o', $a_2 =$ 'l', $a_3 =$ 'l', $a_4 =$ 'e' and $a_5 =$ 'H'.

The second conclusion we can draw is a confirmation of the findings of [25], that the performance of GRUs is almost as good as LSTMs, this we can conclude because the presented GRUs get nearly the same accuracy and top-5-categorical metrics like the corresponding LSTMs (with the same settings), and very close loss value. This conclusion is drawn by comparing GRU128H20L1R, GRU128H20L2R against LSTM128H20L1R and LSTM128H20L2R. This will be further demonstrated in the

next section, when evaluating the fixing performance.

The last conclusion we can draw is that, deeper models tend to outperform shallower models as shown on all the results of both datasets, by comparing LSTM128H20L1N, LSTM128H20L1R, GRU128H20L1R, against LSTM128H20L2N, LSTM128H20L2R, GRU128H20L2R; additionally, bigger models (measured by hidden units size) tend to outperform smaller models, by comparing BiLSTM64H20L1N, BiLSTM128H20L1N, BiLSTM256H20L1N against each others, and BiLSTM64H40L1N, BiLSTM128H40L1N, BiLSTM256H40L1N against each others.

8.3 Fixing evaluations

Different models

Table 10 shows a comparison of the fixers performance for the Simple-Wikipedia dataset and table 11 shows the same comparison for the Reuters-21578 dataset, the results are comparing different neural network architectures. The performance is evaluated using the mean F_1-score, as defined at the end of section 5.1.

The score of the baseline greedy approach is not high because there are two major drawbacks of the greedy approach. The first drawback is matching the first word greedily, this doesn't consider if the taken word would make it hard to fix the remaining text or not, hence it can easily get trapped in choosing a word that leaves the remaining text with portions that cannot be fixed at all, or fixed with a lot of destructive modifications. The second drawback is not considering any neighboring words, and how they might affect the choice of the correct words, which can result in fixings that don't have any particular meaning (for humans).

The DP approach gets much higher score than the baseline greedy approach, because it has a strategy to deal with the first drawback, which is not getting stuck with a word that makes it hard to fix the surrounding context, this is achieved via the grouping layer of the approach which makes sure that neighboring contexts are fixed in a reasonable fashion, with respect to the given dictionary. However, still the second drawback remains, which is fixing words while taking in consideration if such a fixing will make the output fit in context or not, because the dictionary-based approach in general is just checking of the words are correct in the language without further inspection of the context.

The deep learning based approaches take in consideration how to handle the drawback that an introduced fixing doesn't fit in context, this is due to the predictive power of RNN language models to predict text from a given context. The prediction

in both directions backward and forward, made by the bicontext approach, provides a method to predict how likely a given context string to appear in the language (using the occurrence function P_o), then both models behave as if they are peer-reviewing each others, collectively they make a certain decision if both of them are certain about it, and uncertain decision if one of them is certain and the other isn't. The drawbacks of the dictionary-based approaches are handled in the bicontext approach by checking long enough contexts which gives an idea if an introduced fixing would make the text unlikely to occur in the language. The first drawback in the baseline greedy approach is additionally handled by the beam search, which gives some sort of a buffer for the model to try few fixings and pick the ones that don't damage the surrounding text at a given point.

The effectiveness of deep learning is shown by the huge difference against the baseline learning approach, which uses the bicontext approach with replacing the character-based language models from RNN to 3-Grams Markov models, this replacement made the results to be the worst results, which concludes how reliant the bicontext model on the character-based language model, and how effective the RNN model is. This is also supported by comparing the losses on the test sets in the training (in section 8.2), and observing the better the loss and accuracy values are, the better is the corresponding fixer evaluation.

The end-to-end model achieved relatively good results, however they are still much worse than the bicontext approach, in fact they are comparable to the DP approach. The reasons for this are not very clear, but I suspect that the amount of data and the model size might not have been enough for conducting a robust end-to-end training.

The effect of increasing the context history h doesn't seem to show a big enhancement on the fixing performance, and in the smaller dataset (Reuters-21578), it even showed a slightly worse performance. However, the number of hidden units affected the performance noticeably, as shown in the evaluations of the Reuters-21578 dataset. The effects of the history length could be experimented in a future work with much bigger datasets.

Another confirmation of the similar performance of the GRU and LSTM is shown by this experiment, by comparing GRU128H20L1R, GRU128H20L2R against LSTM128H20L1R and LSTM128H20L2R; all the GRU models get very similar performance like the corresponding LSTM.

Finally, from this experiment, we can conclude that the results of the test evaluation, as shown in the previous section, can be used as an indicator for how well the model will fix a text, because an improvement in the backward or forward models tests'

evaluation is correlated with an improvement in the fixing results.

Different bicontext approach features

In order to measure the effectiveness of different components of the bicontext approach, few experiments were ran while changing one setting per experiment. The RNN architectures used BiLSTM128H20L1R+N for the Reuters-21578 dataset and LSTM128H20L2R+N for the Simple-Wikipedia dataset. The default settings for this experiment, is training the models for 100 epochs, with 2 perturbation examples per each correct example. The default setup of the fixing is using beam of size 2 in the beam search, decisions tuner and look-forward strategy. The model was used to fix texts with the four tokenization errors.

The experiments will change one of these per experiment. Beam size 1 and 4 will be experimented, the look-forward strategy will be disabled in one experiment (by not calling the corresponding function in algorithm 14), the decisions tuner will be disabled in one of the experiments (by using default weights in equation 30). Additionally, different training settings are also experimented, once by using models trained for only 50 epochs instead of 100, and secondly by training models without any perturbed examples. There is an experiment on only one fold, which is using 20,000 articles instead of 10,000 during training. Lastly, we try to drop one of the constraints of the problem, which is using text without any typos, so only the first three types of tokenization mistakes, these are generated as in the datasets specifications in section 8.1. The fixing then is evaluated using 3-Fold cross validation, using the mean F_1-score as performance measure. The results are presented in table 12 for both datasets.

The main factor that affected the performance is the decisions tuner, using the default weights (which are not tuned by the decisions tuner) yielded much worse results on both datasets. All the factors of dropping the look-forward, training for less epochs or training with no perturbations, showed that the models get worse, which concludes that all these factors improve the model. However, the effect is not as significant as dropping the decisions tuner. The reason why removing perturbations got worse results is not totally clear; there are two possibilities, either the perturbations made the model more robust as hypothesized or the other possibility could be less training because training with two perturbation examples simply multiples the number of training batches by three (one actual example and two perturbed examples), this can be resolved, in a future work, by training a model with no perturbations for 300 epochs and comparing its results. Using a beam size 4 resulted in a very slight

improvement on the Simple-Wikipedia dataset and slightly worse performance on the Reuters-21578 dataset. However, the beam of size 1 made the performance worse on both datasets. Thus we can conclude using beam size 2 is a reasonable choice, further experiments could be conducted in a future work to inspect if much bigger beam size (10 or 12) would make a difference or not. Using more training data of double the size in the Simple-Wikipedia showed a reasonable improvement, however this result was only evaluated on one fold due to time restrictions. Further exploration of this can be conducted in a future work. Finally, simplifying the problem by excluding the typos from corruptions, resulted in a much better performance up to 96.3% on the Simple-Wikipedia dataset and 91.9% on the Reuters-21578 dataset, which concludes that the typos present a great difficulty in the given problem. Additionally, it also presents a difficulty in terms of time performance, because fixing typos is only possible by trying to add a variety of non-delimiter characters.

Dir	Model	train L	train A	train T5	test L	test A	test T5
back	GRU128H20L1R	1.433	0.582	0.860	1.495	0.566	0.847
back	GRU128H20L2R	1.256	0.630	0.886	1.358	0.603	0.866
back	LSTM128H20L1N	1.652	0.522	0.828	1.687	0.513	0.821
back	LSTM128H20L2N	1.488	0.565	0.852	1.534	0.554	0.843
back	LSTM128H20L1R	1.469	0.572	0.854	1.524	0.559	0.843
back	LSTM128H20L2R	1.302	0.614	0.877	1.391	0.592	0.861
back	BiLSTM128H20L1R	1.337	0.604	0.873	1.421	0.583	0.857
back	BiLSTM128H20L1N	1.328	0.606	0.874	1.414	0.585	0.858
back	BiLSTM128H40L1N	1.325	0.604	0.873	1.429	0.579	0.853
back	BiLSTM256H20L1N	1.039	0.684	0.915	**1.282**	**0.620**	**0.875**
back	BiLSTM256H40L1N	**0.976**	**0.705**	**0.925**	1.368	0.602	0.862
back	BiLSTM64H20L1N	1.635	0.527	0.829	1.670	0.519	0.822
back	BiLSTM64H40L1N	1.600	0.531	0.832	1.641	0.523	0.824
forw	GRU128H20L1R	1.693	0.523	0.823	1.723	0.516	0.818
forw	GRU128H20L2R	1.430	0.595	0.861	1.485	0.581	0.850
forw	LSTM128H20L1N	1.445	0.588	0.857	1.505	0.574	0.846
forw	LSTM128H20L2N	1.274	0.631	0.879	1.370	0.607	0.863
forw	LSTM128H20L1R	1.597	0.548	0.836	1.638	0.537	0.829
forw	LSTM128H20L2R	1.448	0.586	0.856	1.499	0.574	0.847
forw	BiLSTM128H20L1R	1.328	0.617	0.873	1.414	0.596	0.858
forw	BiLSTM128H20L1N	1.334	0.615	0.873	1.419	0.594	0.857
forw	BiLSTM128H40L1N	1.323	0.612	0.873	1.427	0.588	0.855
forw	BiLSTM256H20L1N	1.038	0.692	0.912	**1.282**	**0.629**	**0.874**
forw	BiLSTM256H40L1N	**0.974**	**0.710**	**0.921**	1.355	0.613	0.864
forw	BiLSTM64H20L1N	1.613	0.543	0.835	1.649	0.534	0.828
forw	BiLSTM64H40L1N	1.575	0.545	0.839	1.617	0.535	0.831

Table 9: Reuters-21578 train/test loss and accuracies. L is the categorical-cross entropy, A is the accuracy and T5 is the top-5-categorical accuracy. The shown results are the mean of the 3-fold cross validated metrics.

Model	Fold 1	Fold 2	Fold 3	mean
Greedy baseline	0.421	0.420	0.419	0.420
3-Gram Markov baseline	0.466	0.466	0.471	0.468
DP approach	0.695	0.695	0.698	0.696
End-to-end BiLSTM128H20L1N	0.883	0.795	0.782	0.820
LSTM128H20L1N	0.857	0.853	0.858	0.856
LSTM128H20L1R	0.860	0.850	0.859	0.856
LSTM128H20L2N	0.883	0.874	0.880	0.879
LSTM128H20L2R	0.878	0.871	0.880	0.876
LSTM128H40L2R+N	**0.890**	**0.885**	**0.895**	**0.890**
LSTM128H20L2R+N	**0.889**	**0.880**	**0.888**	**0.886**
GRU128H20L1R	0.860	0.850	0.860	0.857
GRU128H20L2R	0.874	0.870	0.877	0.874
BiLSTM128H20L1R	0.874	0.876	0.885	0.878

Table 10: Fixing evaluations of the Simple-Wikipedia dataset, as measured by the mean F_1-scores. The result of each fold in the 3-fold cross validation is shown.

Model	Fold 1	Fold 2	Fold 3	mean
Greedy baseline	0.477	0.479	0.478	0.478
3-Gram Markov baseline	0.329	0.332	0.331	0.331
DP approach	0.684	0.678	0.679	0.680
End-to-end BiLSTM128H20L1N	0.745	0.740	0.725	0.737
GRU128H20L1R	0.831	0.833	0.836	0.833
GRU128H20L2R	0.857	0.853	0.861	0.857
LSTM128H20L1N	0.841	0.835	0.839	0.838
LSTM128H20L2N	0.862	0.862	0.862	0.862
LSTM128H20L1R	0.839	0.829	0.840	0.836
LSTM128H20L2R	0.854	0.847	0.856	0.852
BiLSTM128H20L1R+N	0.854	0.853	0.856	0.854
BiLSTM128H20L1N	0.862	0.857	0.858	0.859
BiLSTM128H20L1R	0.853	0.852	0.856	0.854
BiLSTM128H40L1N	0.856	0.853	0.851	0.853
BiLSTM256H20L1N	**0.877**	**0.875**	**0.866**	**0.873**
BiLSTM256H40L1N	0.864	0.852	0.845	0.854
BiLSTM64H20L1N	0.829	0.826	0.828	0.828
BiLSTM64H40L1N	0.828	0.825	0.830	0.828

Table 11: Fixing evaluations of the Reuters-21578 dataset, as measured by the mean F_1-scores. The result of each fold in the 3-fold cross validation is shown.

Type	F_1-score Simple-Wikipedia	F_1-score Reuters-21578
Architecture	LSTM128H20L2-R+N	BiLSTM128H20L1-R+N
Default	0.886	0.858
Beam size 1	0.872	0.846
Beam size 4	0.887	0.856
No typos	**0.963**	**0.919**
No look-forward	0.877	0.848
Using default tuner weights	0.819	0.814
50 Epochs	0.876	0.840
No perturbations	0.879	0.848
More data	0.903	-

Table 12: Experiements with changing one setting per experiment, using the bi-context model. The default setting is using beam size 2, look-forward, tuned weights, 100 epochs trained models, and fixing files that has all tokenization errors including typos. The shown values are the mean F_1-scores of 3-fold cross validation.

9 Conclusion and future work

9.1 Summary and conclusions

In this thesis, I have addressed the problem of automatically fixing tokenization errors in a given text. A variety of approaches were presented, two of them were non-learning dictionary-based approaches, namely, greedy approach and dynamic programming approach. The other three approaches are based on machine learning, by using character-based probabilistic language models. The main learning approach is the bicontext deep learning based approach, the second one is the same but replaces the character-based language model by a 3-Gram Markov models instead of RNNs, and the third approach merges most of the components of the bicontext approach into one end-to-end deep learning approach. The bicontext approach was demonstrated to show the best performance with F_1-score that is close to 90%, and 96% in case the corrupt texts didn't include typos as tokenization errors.

Both dictionary-based approaches try to reconstruct the words of a given text to make them as correct as possible according to a given dictionary. The greedy based approach tries to match words from the beginning as much as it can, which is usually not optimal to fix the remaining text. The dynamic-programming approach, enhances this by considering a variety of possibilities to match words in order to globally match as much words as possible, which makes the DP approach more powerful.

The learning based approaches are based on joint probabilities over characters, using probabilistic character-based language models, these probabilities are used in order to give scores for different fixing decisions. A beam search is then used to find the most probable fixed text, by fixing a given corrupt text. Additionally, the approaches maintain two contexts in order to properly decide how to fix the character between them. The bicontext approach is more hand-crafted, as it uses two separate RNN language models and combine their output and tune it in order to make fixing decisions. However, the end-to-end approach, on the other hand, leaves all the combinations and tuning for the neural network to learn.

The models were evaluated according to two datasets, Reuters-21578 and Simple-

Wikipedia. The Reuters-21578 dataset had less data and higher specificity of topics; while the Simple-Wikipedia dataset had more data and more diverse topics. The five models were compared against each other, the bicontext approach showed the best performance with F_1-score ≈ 0.88. The end-to-end approach showed also some promising results with F_1-score ≈ 0.78. The baseline learning approach (3-Gram) which is the same like the bicontext approach but using 3-Gram Markov model as the character-based language model, got an F_1-score ≈ 0.40 which is significantly outperformed by the both approaches based on deep learning, which concluded the effectiveness of deep learning to produce robust probabilistic models. On the other hand, the greedy approach showed the worst performance with F_1-score ≈ 0.45, because it often matches words that makes it hard to fix the remaining text. The dynamic-programming approach had relatively good performance with F_1-score ≈ 0.70, because it considers a divide and conquer strategy that attempts to ensure that no poor decisions are made that would corrupt more surrounding tokens. Non-learning based approaches suffer from not choosing words that fit within the given context, in comparison with the learning based approaches.

The bicontext approach was thoroughly experimented with different model settings for the character-based RNN language models, like varying the neural network's depth, hidden units size, context history length, RNN architecture and reversing the input context. The choice of the context history and RNN architecture didn't show a big difference, in particular, increasing the history from 20 to 40, and similarly for using GRU, LSTM or bidirectional LSTMs, they had comparable performance (for models with similar size). On the other hand, deepening the neural network or using more hidden units showed more promising improvements for all the models. Reversing input showed improvement in the case of the backward language models, because the sequence is given in the order more suitable for prediction. In addition to that, the performance of the bicontext approach was improved by the improvement of the character-based language models's ability to predict the successor character as measured by the loss, accuracy and top-5-categorical accuracy metrics on the test set.

Additionally, the bicontext model was used to evaluate a variety of settings, like training the model less (50 epochs instead of 100), using bigger or smaller beam size, turning off the decisions tuner, turning off the looking forward, training with no input perturbations, or restricting the types of tokenization errors to be fixed. The experiments concluded a significant improvement caused by using the decisions tuner; each of the remaining bicontext approach components also showed some improvement.

Last but not least, the bicontext approach achieved much better results when it

was solving tokenization errors that don't include typos, with a best performance of mean F_1-score if 96.3%.

9.2 Future work

In the developed bicontext approach, there are two main components that are designed specifically for the given tokenization errors, mainly input perturbations and decisions tuner. However, these two components could be changed in order to solve other similar problems, that has to do with fixing text according to some language model.

One possible application is to fix text that is parsed from a corrupted picture (blurry, for example) using an Optical Character Recognition (OCR) tool [37], because such parsing results in garbled characters. Unlike the garbled characters used in this thesis (which was one uniformly random garbled character), in such application there might be consistent patterns of corruption. For example, the characters '1', 'l', 'i' might be exchanged during parsing, because they look similar. Such patterns, could be embedded in the input perturbations, in order to be able to identify and separate these particular parsing mistakes, and hence it could fix texts that were not correctly retrieved by an OCR tool, due to the low quality of the images.

As concluded, the strength of the probabilistic model used in the character-based language model significantly affects the fixing performance of the given bicontext fixer. A series of experiments could be conducted, in order to inspect the effects of dropouts, and using different dropout rates. Furthermore, increasing the size of the model also showed an improvement in the performance, mainly increasing the number of layers and the number of hidden units. An experiment could be conducted using deeper networks of three or four layers instead of only two layers, and also increasing the hidden units size. Additionally, evaluations on bigger datasets can be conducted since the difference in the datasets' sizes between both datasets showed an improvement in performance, in addition to the improvement on more data that was also shown on one fold from the Simple-Wikipedia dataset. Finally, even though input perturbations showed an improvement in the results, an experiment could verify the reason of this effectiveness, whether it was due to more training examples or because of the introduced perturbations.

There are two abstract ideas that could also improve the probabilistic model. The first idea is to augment a context of characters with a context of words, in order to better identify the fixes that result in more meaningful fixings. The second idea is to augment the model with an encoder that encodes the whole text before predicting

any decisions, and then the predictions can be made by a given context and the encoding of the text together. This gives the model an idea about the topic of the text, which can decide which words are more likely to appear in text. For example, the word 'RNN' is more likely to be fixed to 'RNA' if the context of the text is about biology, and to 'RNE' if the context is about Spanish radio (or similar topic), or be left as is if the context is about deep learning.

Bibliography

[1] H. Déjean and J.-L. Meunier, "A system for converting pdf documents into structured xml format," in *International Workshop on Document Analysis Systems*, pp. 129–140, Springer, 2006.

[2] A. Voutilainen, "Part-of-speech tagging," *The Oxford handbook of computational linguistics*, pp. 219–232, 2003.

[3] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[5] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[6] Y. Wen, I. H. Witten, and D. Wang, "Token identification using hmm and ppm models," in *Australasian Joint Conference on Artificial Intelligence*, pp. 173–185, Springer, 2003.

[7] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning.," 2017.

[8] S. Ghosh and P. O. Kristensson, "Neural networks for text correction and completion in keyboard decoding," *CoRR*, vol. abs/1709.06429, 2017.

[9] K. Sakaguchi, K. Duh, M. Post, and B. V. Durme, "Robsut wrod reocginiton via semi-character recurrent neural network," *CoRR*, vol. abs/1608.02214, 2016.

[10] S. Chollampatt, D. T. Hoang, and H. T. Ng, "Adapting grammatical error correction based on the native language of writers with neural network joint models," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1901–1911, 2016.

[11] S. Chollampatt, K. Taghipour, and H. T. Ng, "Neural network translation models for grammatical error correction," *arXiv preprint arXiv:1606.00189*, 2016.

[12] Z. Xie, A. Avati, N. Arivazhagan, D. Jurafsky, and A. Y. Ng, "Neural language correction with character-based attention," *arXiv preprint arXiv:1603.09727*, 2016.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[14] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[15] J. J. Tithi, N. C. Crago, and J. S. Emer, "Exploiting spatial architectures for edit distance algorithms," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 23–34, IEEE, 2014.

[16] R. De La Briandais, "File searching using variable length keys," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), (New York, NY, USA), pp. 295–298, ACM, 1959.

[17] S. S. Haykin, S. S. Haykin, S. S. Haykin, and S. S. Haykin, *Neural networks and learning machines*, vol. 3. Pearson Upper Saddle River, NJ, USA:, 2009.

[18] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[19] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[21] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[23] F. J. Pineda, "Generalization of back propagation to recurrent and higher order neural networks," in *Neural information processing systems*, pp. 602–611, 1988.

[24] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, pp. 157–166, Mar. 1994.

[25] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (GRU) neural networks," *CoRR*, vol. abs/1701.05923, 2017.

[28] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[29] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*. Springer Publishing Company, Incorporated, 1st ed., 2008.

[30] F. Peng and D. Schuurmans, "Combining naive bayes and n-gram language models for text classification," in *European Conference on Information Retrieval*, pp. 335–350, Springer, 2003.

[31] P. Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP*. Artificial intelligence programming languages, Morgan Kaufman Publishers, 1992.

[32] P. Hayes-Roth, M. Fox, G. Gill, D. Mostow, and R. Reddy, "Speech understanding systems: Summary of results of the five-year research effort," 1976.

[33] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010.

[34] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Computer Speech & Language*, vol. 13, no. 4, pp. 359–394, 1999.

[35] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1.

[36] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[37] A. Chaudhuri, K. Mandaviya, P. Badelia, S. K. Ghosh, *et al.*, *Optical Character Recognition Systems for Different Languages with Soft Computing.* Springer.