Fuzzy Name Matching using Neural Network Approach

Practical Implementation of Convolution Neural Networks for fuzzy matching
Master thesis by Oleg Burik
Date of submission: June 6, 2023

1. Review: Prof. Dr. Stefan Ulbrich
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Mathematics Department
Optimization

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Oleg Burik, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 6. Juni 2023

O. Burik

Contents

1. Introduction

This thesis addresses the persistent problem faced by businesses across diverse fields, including B2B, B2C, E-commerce, service-based, healthcare, NPOs, financial institutions, telecommunications, retail, real estate, and transportation. Specifically, nearly all companies in these fields use one or more Customer Relationship Management (CRM) systems to manage their internal and external relationships, as well as store and protect various types of data, such as product information, customer data, and shareholder information. This data can be used to develop prediction or suggestion models to better understand customer behavior, market trends, and pricing development.

However, despite the availability of numerous tools to solve these problems, their effectiveness is contingent upon the availability of valid and reliable data. To obtain insights from the data, companies must first collect and prepare it for use. The collection and processing of data constitute the foundation of any application that relies on this data over its lifetime.

Consider a company that hosts various events, both in-person and online, and collects participant data through a registration form that includes the participant's name. While the company can use it for the variety of internal tasks, such as advertising or product suggestions, the data is not directly useful. One potential issue with this kind of data is the validity, which requires matching the data from the registration sheet to the data in the CRM system. While the CRM data can be assumed to be accurate, there is no guarantee of the quality of the registration data, which may contain errors such as misspellings, typos, or even conscious misspellings, such as using a nickname instead of the real name. Moreover, the data may be incomplete and contain sparse information, especially when the event includes participants from outside the company. This problems, that are faced by big companies on everyday basis, serves as inspiration for this theses.

We define a set of problems that arise from matching two large datasets of character strings without contextual information or semantic sense, which include proper names and common errors. We refer to this set of problems as *Fuzzy Name Matching* (FNM). Our goal is to develop a mapping solution that matches misspelled names to their valid counterparts, create an effective solution architecture, and perform the matching task in feasible time.

In summary, this paper presents a critical issue in data management faced by businesses across various fields and proposes a solution to the problem of fuzzy name matching. The rest of the paper is organized as follows: Chapter 2 discusses related work and preliminaries in the area of fuzzy string matching, Chapter 3 presents the modern neural network solutions and Chapter 4 utilizes this tools to create a robust solution. In conclusion, the results will be summarized and future work will be discussed.

2. Preliminaries

To have a better overview of the *Fuzzy Name Matching* problem, we first explore the existing solutions and tools. We start with definitions and assumptions in Section 2.2, that we will refer to throughout this theses. In Section 2.3 we introduce the string (dis)similarity metrics, that can measure how similar two strings are. We also devote ourselves into exploration of the phonetic algorithms in Section 2.4, that are used for exactly the same task of FNM previously. We implement and analyse all the explored solutions consequently in Section 2.5. The idea behind it, is to create a benchmark for the proposed solution in Chapter 4. During this Section we present, implement and analyse the simple dictionary search. We conclude the Chapter with the short summary of the performance of the presented existing solution in Section 2.6.

2.1. Motivation

> As computers are used in an ever-widening variety of lexical processing tasks, the problem of error detection and correction becomes more critical. Mere volume, if nothing else, will prevent the employment of manual detection and correction procedures.
>
> —Frank Damerau [Dam64]

As highlighted by Michail Boytsov [Boy11], detection and correction of errors constitute a well-established task that predates the birth of the modern computers. With the emergence of the computer era, extensive research has been conducted to explore the potential of computational power in solving this task. This endeavor has given rise to a diverse range of approximate search algorithms. Presently, these algorithms find applications in various domains such as spellcheckers, computer-aided translation systems, optical character recognition systems, spoken-text recognition and retrieval systems, computational biology, phonology, and phonetic matching. Approximate dictionary searching plays a crucial role in computer security by facilitating the identification of "weak" passwords susceptible to dictionary attacks. Another related application is the detection of similar domain names, primarily utilized for typo-squatting. Such detection is often crucial in disputes involving nearly identical domain names, especially when brand names are implicated[1].

In addition to other sources of corrupted data, such as recognizing spoken and printed text, many errors stem from human factors, encompassing cognitive and mechanical errors. Notably, English orthography is notorious for its inconsistency, presenting significant challenges. For example, one of the most frequently

[1] A dispute in 2006 concerning the domains *telefónica.cl* and *telefonica.cl* resulted in the court canceling the "accented" domain version due to its striking similarity to the original one [Zal06]

misspelled words is "definitely." Despite its apparent simplicity, it is often written as "defin**a**tly" [Col09]. A prevalent mechanical error is the reversal of adjacent characters, referred to as a transposition. Transpositions account for 2-13 percent of all misspelling errors [Pet86]. Hence, it is imperative to consider methods that are aware of transpositions.

In this Chapter, we are going to stick to the algorithms and methods for fuzzy string matching that were outlined by Boytsov [Boy11]. We will use the same theoretical ground to create and test the algorithms, that were used previously. We use them to create a benchmark for our own solution. For the implementation part, we will rely on most recent implementations of the search algorithms for Python programming language, that this theses will rely upon.

As mentioned by Boytsov, search methods can be classified into **on-line** and **off-line** methods. On-line search methods include algorithms for finding approximate pattern occurrences in a text that cannot be preprocessed and indexed. A well-known on-line search algorithm is based on the dynamic programming approach [Sel74]. It is $O(n \cdot m)$ in time and $O(n)$ in space, where n and m are the lengths of a pattern and a text, respectively. The on-line search problem was extensively studied and a number of algorithms improving the classic solution were suggested, including simulation of non-deterministic finite automata (NFA) [WM92][Nav01][Nav01], simulation of deterministic finite automata (DFA) [Ukk85][Mel96], and bit-parallel computation of the dynamic programming matrix [Mye99].

However, the running time of all on-line search methods is proportional to the text size. Given the volume of textual data, this approach is inefficient. The necessity of much faster retrieval tools motivated development of methods that rely on text preprocessing to create a search index. These search methods are known as **off-line** or **off-line** search methods.

There are two types of indexing search methods: *sequence-oriented* methods and *word-oriented* methods. Sequence-oriented methods find arbitrary substrings within a specified edit distance. Word-oriented methods operate on a text that consists of strings divided by separators. Unlike sequence-oriented methods, they are designed to find only complete strings (within a specified edit distance). As a result, a word-oriented index would fail to find the misspelled string “wiki pedia” using “wikipedia” as the search pattern (if one error is allowed). In the case of natural languages – this is a reasonable restriction.

The core element of a word-oriented search method is a *dictionary*. The dictionary is a collection of distinct searchable strings (or string sequences) extracted from the text. Dictionary strings are usually indexed for faster access. A typical dictionary index allows for exact search and, occasionally, for prefix search. In this survey, we review associative methods that allow for an approximate dictionary search. Given the search pattern p and a maximum allowed edit distance k, these methods retrieve all dictionary strings s (as well as associated data) such that the distance between p and s is less than or equal to k.

Commonly used indexing approaches to approximate dictionary searching include the following:

1. *Full and partial neighborhood generation* [RO05];
2. *n-gram indices* [NST05];
3. *Prefix trees (tries)* [CGL04]; [MS04];
4. *Metric space methods* [BYN98]; [Fre07].

We will however not use any of the indexing approaches, since it is not the goal of this theses. Instead, we want to establish the "ground truth" in this Chapter. Meaning, that we’re looking for the optimal

performance on the unvalidated data for each of the further introduced algorithms. Hence, we will sacrifice the possible increase in performance using (e.g. tries) for the sake of accuracy of the matchings.

In the further investigation of the fuzzy matching algorithms, we refer to the theoretical conclusions from [Boy11] and will focus on the defining of the tools and their programmatic counterparts. We are going to introduce the common distance metrics to measure (dis)similarities between two strings as well as the phonetic algorithms to approach to the solution of the FNM problem .

2.2. Definitions and Assumptions

Let $\Sigma = \{\Sigma_i\}$ be a finite ordered alphabet of the size $|\Sigma|$. A string is a finite sequence of characters over Σ. The set of all strings of length n over Σ is denoted by Σ^n, while $\Sigma^* = \sum_{n=1}^{\infty} \Sigma^n$ represents the set of all strings.

Unless otherwise specified, we use p, s, u to represent arbitrary strings and a, b, c to represent single-character strings, or simply characters. The empty string is represented by ε. For any string $s \in \Sigma^*$, its length is denoted by $|s|$. A series of string variables and/or character variables represents their concatenation.

To avoid confusion between string variables with indices and string characters, we denote the i-th character of the string s by $s[i]$. A contiguous subsequence of string characters is a substring. The substring of s that starts at position i and ends at position j is denoted by $s[i:j]$, i.e., $s[i:j] = s[i]s[i+1]\dots s[j]$. The reversed string, i.e., the string $s[n]s[n-1]\dots s[1]$, is denoted by $\text{rev}(s)$.

Assume that the string s is represented as a concatenation of three possibly empty substrings s_1, s_2, and s_3, i.e., $s = s_1 s_2 s_3$. Then substring s_1 is a prefix of s, while substring s_3 is a suffix of s.

A substring of fixed size n is called a n-gram (also known as an q-gram). Consider a string s of length m. We introduce the function n-grams(s) that takes the string s and produces a sequence of $m-n+1$ n-grams contained in s:

$$\text{n-grams}(s) = s[1:n], s[2:n+1], \dots, s[m-n+1:n].$$

If $m < n$, n-grams(s) produces the empty sequence. It can be seen that n-grams(s) is essentially a mapping from Σ^* to the set of strings over the alphabet Σ^n, which is comprised of all possible n-grams.

The string s can be transformed into a frequency vector (we also use the term unigram frequency vector). The frequency vector is a vector of size $|\Sigma|$, where the i-th element contains the number of occurrences of the i-th alphabet character in the string s. The frequency vector produced by the string s is denoted by vect(s).

Because n-grams(s) is itself a string over the alphabet Σ^q, it can also be transformed into a frequency vector vect(n-grams(s)). To distinguish it from the (unigram) frequency vector obtained directly from the string s, we use the term n-gram frequency vector.

A signature is a variant of a frequency vector. It is a binary vector of size $|\Sigma|$, where the i-th element is equal to one if and only if the i-th alphabet character belongs to s, and is zero otherwise. The signature produced by the string s is denoted by signat(s).

One of the key concepts of this Section is an *edit distance* ED(p, s), which is equal to the minimum number of edit operations that transform a string p into a string s. A *restricted* edit distance is computed as the minimum number of non-overlapping edit operations that make two strings equal (and do not act twice on the same substring). If the edit operations include only insertions, deletions, and substitutions, it is the Levenshtein distance. If, in addition, transpositions are included, it is the Damerau-Levenshtein distance. The Hamming distance is a variant of the edit distance where the only basic edit operations are substitutions.

The restricted Levenshtein distance is always equal to the unrestricted Levenshtein distance, but this is not always the case for the Damerau-Levenshtein distance.

We define approximate dictionary searching as finding a set of strings that match the pattern within k errors.

In our work we refer to the unvalidated third-party dictionary of names as **LHS** [Left Hand Side] and to the validated CRM dictionary of names as **RHS** [Right Hand Side].

In the coming Section we define the necessary tools more formally.

2.3. Similarity metrics for two strings

Definition 2.3.1 (Dictionary) *A dictionary $W = (s_1, s_2, \ldots, s_N)$ is an ordered set of strings, where N is the number of dictionary strings.*

Definition 2.3.2 (Approximate dictionary searching) *Let p be a search pattern (query string) and k be a maximum allowed restricted edit distance. Then the problem of approximate dictionary searching consists in finding all indices i in $W = (s_1, s_2, \ldots, s_N)$ such that ED$(s_i, p) \leq k$.*

Note that Definition 2.3.2 employs a restricted distance. This definition also implies that we consider associative methods that are capable of retrieving both strings and associated data, such as string identifiers. Associated data is also known as *satellite data*.

2.3.1. Edit Script and Basic Edit Operations

In a more general perspective, one string can be transformed into another by a sequence of atomic substring transformations. This sequence is an edit script (also known as a trace), while atomic substring transformations are basic edit operations. A basic edit operation that consists of mapping string u into string v is represented by $u \to v$ (for simplicity of exposition, we omit specification of exact positions where basic operations are applied). We denote the set of basic edit operations by $\mathbb{B}$.

Basic edit operations are usually restricted to the following set of single-character operations:

- Insertion: $\varepsilon \to b$
- Deletion: $a \to \varepsilon$
- Substitution: $a \to b$ (replacement)

In some cases, $\mathbb{B}$ is expanded to include transpositions, which consist of the reversal of adjacent characters: $ab \to ba$.

Property 2.3.1 *We assume that $\mathbb{B}$ satisfies the following:*

- *If $u \to v \in \mathbb{B}$, then the reverse operation $v \to u$ also belongs to $\mathbb{B}$ (symmetry).*
- *$a \to a \in \mathbb{B}$ (single-character identity operations belong to $\mathbb{B}$).*
- *$\mathbb{B}$ is complete: for any two strings p and s, there always exists an edit script that transforms p into s.*

Note that $\mathbb{B}$ is not necessarily finite.

2.3.2. Definition of Edit Distance

Similarity of two strings can be expressed through the length of an edit script that makes the strings equal:

Observation 2.3.1 *Given a set of basic edit operations, the edit distance $ED(p, s)$ is equal to the length of the shortest edit script that transforms string p into string s. A shortest script that transforms p into s is an optimal edit script. If the set of basic edit operations contains only insertions, deletions, and substitutions, it is the Levenshtein distance (an identity operation is a special case of substitution). If, in addition, the set of basic edit operations includes transpositions, it is the Damerau-Levenshtein distance.*

Definition 2.3.3 (Edit distance) *Given a set of basic edit operations $\mathbb{B}$ and a function $\delta()$, which assigns costs to all basic edit operations from $\mathbb{B}$, the generic edit distance between strings p and s is defined as the minimum cost of an edit script that transforms p into s.*

Property 2.3.2 *We assume that the cost function $\delta(u \to v)$ satisfies the following:*

- $\delta(u \to v) \in \mathbb{R}$ *(the cost function is real-valued)*
- $\delta(u \to v) = \delta(v \to u)$ *(symmetry)*
- $\delta(u \to v) \geq 0$, $\delta(u \to u) = 0$, *and* $\delta(u \to v) = 0 \Rightarrow u = v$ *(positive definiteness)*
- *For all $\gamma > 0$, the set of basic operations $\{u \to v \in \mathbb{B} \mid \delta(u \to v) < \gamma\}$ is finite (finiteness of a subset of basic edit operations whose costs are bounded from above)*

Note that the last property holds automatically for finite $\mathbb{B}$.

The edit distance can be interpreted as the minimum cost at which one string can be transformed into another. Basic edit operations can be assigned individual costs $\delta(a \to b)$ [Wagner and Fischer 1974]. We extend the cost function $\delta()$ to an edit script $E = a_1 \to b_1, a_2 \to b_2, \ldots, a_{|E|} \to b_{|E|}$ by defining $\delta(E) = \sum_{i=1}^{|E|} \delta(a_i \to b_i)$. We now let the distance from a string p to a string s be the minimum cost of all edit scripts that transform p into s. This edit distance variant is commonly referred to as a *generalized Levenshtein distance*.

Theorem 2.3.1 *From Properties 2.3.1 and 2.3.2, it follows that:*

- *For any two strings p and s, there exists a script with the minimum cost, i.e., the edit distance from p to s is properly defined.*
- *The generic edit distance described by Definition 2.3.3 is a metric [WF74].*

The proof of this theorem can be found at Appendix D of [Boy11].

Subadditivity of Edit Distance The subadditivity of edit distance makes it possible to use edit distance with metric space methods, such as the Burkhard-Keller tree (BKT) [BK73]. Nonetheless, the problem of minimization over the set of all possibly overlapping edit operations may be hard. To offset computational complexities, a similarity function defined as the minimum cost of a restricted edit script is commonly used. The restricted edit script does not contain overlapping edit operations and does not modify a single substring twice. We refer to the corresponding edit distance as the *restricted edit distance*. Although, it's a

common approach to the approximate search problem, we won't deepen into this topic. Instead we will use the greedy search for benchmarking the algorithms based on the edit distances, since it will always produce the optimal matches.

Observation 2.3.2 *Any unrestricted edit distance is a lower bound for the corresponding restricted edit distance.*

Definition 2.3.4 (Alignment) *Let strings p and s be partitioned into the same number of possibly empty substrings: $p = p_1 p_2 \dots p_l$ and $s = s_1 s_2 \dots s_l$, such that $p_t \to s_t \in \mathbb{B}$. Additionally, we assume that p_t and s_t cannot be empty at the same time. We say that this partition defines an **alignment** $A = (p_1 p_2 \dots p_l, s_1 s_2 \dots s_l)$ between p and s, in which substring p_t is aligned with substring s_t. The alignment represents the restricted edit script $E = p_1 \to s_1, p_2 \to s_2, \dots, p_l \to s_l$. We define the cost of alignment A as the cost of the corresponding edit script and denote it as $\delta(A)$:*

$$\delta(A) = \sum_{t=1}^{l} \delta(p_t \to s_t) \tag{2.1}$$

*An **optimal alignment** is an alignment with the minimum cost.*

In following we introduce the *dynamic programming algorithm* to compute edit distance [WF74].

The main principle of the algorithm is to express the cost of alignment between strings p and s using costs of alignments between their prefixes. Consider the prefix $p[1:i]$ of length i and the prefix $s[1:j]$ of length j of strings p and s, respectively. Assume that $A = (p_1 p_2 \dots p_l, s_1 s_2 \dots s_l)$ is an optimal alignment between $p[1:i]$ and $s[1:j]$, whose cost is denoted by $C_{i,j}$.

Using Equation 2.1 and the definition of an optimal alignment, it is easy to show that $C_{i,j}$ can be computed using the following generic recursion [Ukk85]:

$$C_{0,0} = 0, C_{i,j} = \min \left\{ \delta(p[i':i] \to s[j':j]) + C_{i'-1,j'-1} \,\middle|\, p[i':i] \to s[j':j] \in B \right\} \tag{2.2}$$

Recursion 2.2 is an example of a *dynamic programming solution*. The set of $(|p|+1) \cdot (|s|+1)$ numbers $\{C_{i,j}\}$ is commonly referred to as a *dynamic programming matrix* (or shortly DP matrix). It can also be seen that:

- The cost of alignment between strings p and s is equal to $C_{|p|,|s|}$.
- All optimal alignments can be recovered by backtracking through Recursion 2.2.

Let us now consider the case of Levenshtein distance, where $p[i':i] \to s[j':j]$ is a unit-cost single character insertion, deletion, or substitution. Therefore,

$$\delta(p[i':i] \to s[j':j]) = \begin{cases} 1, & \text{if } p[i':i] \neq s[j':j] \\ 0, & \text{otherwise} \end{cases}$$

Furthermore, there are three possible combinations of i' and j', which correspond to deletion, insertion, and substitution, respectively:

- $i' = i - 1$ and $j' = j$.

- $i' = i$ and $j' = j - 1$.
- $i' = i - 1$ and $j' = j - 1$.

Definition 2.3.5 (Levenshtein distance computing algorithm) *Considering previous simplifications, we can rewrite generic recursion (2.2) for Levenshtein distance as follows:*

$$C_{i,j} = \min \begin{cases} 0, & \text{if } i = j = 0 \\ C_{i-1,j} + 1, & \text{if } i > 0 \\ C_{i,j-1} + 1, & \text{if } j > 0 \\ C_{i-1,j-1} + [p[i] \neq s[j]], & \text{if } i, j > 0 \end{cases} \tag{2.3}$$

It can be seen that the DP matrix can be computed by a column-wise top-down traversal (one column at a time) or by row-wise left-to-right traversal (one row at a time) in $O(|p| \cdot |s|)$ time and space. The cost of optimal alignment alone can be computed in $O(|p| \cdot |s|)$ time and $O(min(|p|, |s|))$ space (the algorithm has to remember only last column or row to compute a new one) [cite: Navarro 2001a].

Furthermore, Lowrence and Wagner [WF74] show, that one can also successfully compute the unrestricted Damerau-Levenshtein distance in the same recursive manner:

Definition 2.3.6 (unrestricted Damerau-Levenshtein distance computing algorithm) *The generic recursion 2.2 can be rewritten for unrestricted Damerau-Levenshtein distance as follows [WF74]:*

$$C_{i,j} = \min \begin{cases} 0, & \text{if } i = j = 0 \\ C_{i-1,j} + 1, & \text{if } i > 0 \\ C_{i,j-1} + 1, & \text{if } j > 0 \\ C_{i-1,j-1} + [p_i \neq s_j], & \text{if } i, j > 0 \\ \min\limits_{\substack{0<i'<i, 0<j'<j \\ p[i]=s[j'], p[i']=s[j]}} C_{i'-1,j'-1} + (i - i') + (j - j') - 1 \end{cases} \tag{2.4}$$

In addition, Lowrance and Wagner demonstrated that the inner minimum in Recursion 2.4 is achieved at the largest $i' < i$ and $j' < j$ that satisfy $p[i] = s[j']$ and $p[i'] = s[j]$. Lowrance and Wagner proposed an algorithm to compute Recursion 2.4 in $O(|p| \cdot |s|)$ time.

These two edit distances are probably the most common distance metrics for measuring string dissimilarities.

We will use the introduced distance metrics for approximate matching of two datasets, by iterating through the RHS name-by-name. This way we will build the benchmark for the neural network solution introduced in Chapter 4. To create a broader picture and have the broader set of metrics for our work, we will also introduce several other distance metrics.

The Jaro-Winkler distance is a string similarity measure between two strings, defined as follows:

Definition 2.3.7 (Jaro-Winkler distance) *The Jaro-Winkler distance is a string metric for measuring the similarity between two strings s and t.*

Let l be the length of the common prefix between s and p, and let m be the number of character matches where the characters are not in the common prefix.

The Jaro distance d_J is defined as:

$$d_J(s,p) = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s|} + \frac{m}{|p|} + \frac{m-l}{m}\right) & \text{otherwise} \end{cases}$$

The Jaro-Winkler distance d_{JW} is a modification of the Jaro distance, that uses a prefix scale λ which gives more favourable ratings to strings that match from the beginning for a set prefix length l:

$$d_{JW}(s,p) = d_J(s,p) + l * \lambda \cdot l \cdot (1 - d_J(s,t))$$

where λ is a constant scaling factor between 0 and 0.25, typically set to 0.1.

Definition 2.3.8 (Jaccard similarity coefficient) *The Jaccard similarity coefficient is a measure of similarity between two sets A and B. It is defined as the size of the intersection of A and B divided by the size of the union of A and B:*

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Alternatively, the Jaccard similarity coefficient can also be expressed in terms of the number of elements that are common to both sets divided by the total number of distinct elements in both sets:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The Jaccard similarity coefficient ranges from 0 to 1, with 0 indicating no similarity and 1 indicating complete similarity.

Definition 2.3.9 (Hamming distance) *The Hamming distance is a measure of the difference between two strings of equal length. It is defined as the number of positions at which the corresponding symbols are different:*

$$d_H(s,p) = \sum_{i=1}^{|s|} [s_i \neq p_i]$$

where s and p are the two strings of equal length, $|s|$ is the length of s, and $[s_i \neq p_i]$ is the Iverson bracket, which evaluates to 1 if s_i is different from p_i and 0 otherwise.

The Hamming distance is always non-negative, and it is zero if and only if the two strings are identical.

2.3.3. Learnable similarity string metrics

One further development of the similarity metrics was listed by Michail Bilenko [Bil04]. In his dissertation he discusses the importance of similarity estimation in machine learning and data mining tasks, particularly in clustering and information integration applications. While standard distance functions like edit distance are commonly used, they may not capture the appropriate notion of similarity for specific domains or applications. Bilenko proposes the use of learnable similarity functions to address this issue.

Learnable similarity functions can be trained using supervision in the form of similar or dissimilar pairs of instances, allowing accurate estimates tailored to the specific domain and task. Bilenko explores this approach in the context of record linkage, clustering, and blocking tasks.

For record linkage, the goal is to identify records referring to the same entity. Learnable similarity functions are employed to estimate similarity between field values and overall similarity between records. Two variants of learnable edit distance for string comparison are presented, along with a record-level similarity function that combines field similarities using Support Vector Machines.

In clustering, similarity functions determine the grouping of instances. This framework supports various distance measures, including Bregman divergences and directional measures like cosine similarity. The learnable similarity functions, trained within the framework, lead to significant improvements in clustering accuracy.

Blocking, a critical step for scalability in record linkage and clustering, involves selecting approximately similar pairs without considering all possible pairs. Bilenko proposes learning blocking functions automatically from linkage and semi-supervised clustering supervision. This allows for the construction of efficient and accurate blocking methods without the need for manual construction or parameter tuning.

This approach, in fact, looks promising for our task. We are also going to include it in our benchmarking analysis. One major drawback of this approach however implies the supervised or semi-supervised learning whenever two datasets are presented. Current implementation of this approach is presented by dedupe.io. After careful estimation of this implementation, it turned out, that for the best performance, one has to re-train the weights of the model using manual semi-supervised learning every time new data arises. Since we're interested in the stable solution for business purposes, the re-training of the model every time will lead to additional complexity and manual supervision to support such system over time. It makes this approach less attractive. We will however present its performance in Section 2.5 together with other metrics.

2.4. Phonetic algorithms

While using edit distance as a measure of a string (dis)similarity is a general approach for string matching, it might not necessarily be the optimal one. In this Section we outline some common generic algorithms for the exact our purpose - *Fuzzy Name Matching*. These algorithms are build to make use of information about names and their misspelling, as well as their phonetic similarity. The general idea is to create a hashing algorithm $h(\cdot)$, that takes a name as an input string s over an alphabet Σ and projects it to string $h(s)$ over reduced alphabet σ. The algorithm $h(\cdot)$ induces a character-wise projection from the set of strings over the original alphabet Σ to the set of strings over the reduced alphabet σ in a straightforward way. Given a string s of the length n, a corresponding projection $h(s)$ is given by:

$$h(s[1]s[2]s[3]...s[n]) = h(s[1])h(s[2])h(s[3])...h(s[n]).$$

Note that the output string may change, if e.g. $h(s[i]) = \varepsilon$.

2.4.1. NYSIIS

The New York State Identification and Intelligence System (NYSIIS) algorithm is a phonetic algorithm used for FNM. It was developed by the New York State Identification and Intelligence System in the 1970s for use in name matching in their criminal justice system.

The NYSIIS algorithm is a rule-based algorithm that converts a given input string into a phonetic code. The phonetic code is designed to represent the pronunciation of the input string, rather than its spelling. The idea is that two strings that sound the same should have the same phonetic code, even if they are spelled differently.

The algorithm can be described as a hash function $h_{nysiis} : \Sigma \to \sigma$, that transforms the name s into its phonetic code. In case of NYSIIS algorithm Σ is the English alphabet, whereas σ is the uppercase subset of Σ. Let's s_1 be the prefix and s_3 a suffix of the name s, then n_{s_i} is the length of the given substring for $i \in \{1, 2, 3\}$, n_s is the length of the name and the transformation functions are defined as follows:

$$t_1(s) = \begin{cases} \text{MCC} & \text{if } s = \text{MAC}, \\ \text{NN} & \text{if } s = \text{KN}, \\ \text{C} & \text{if } s = \text{K}, \\ \text{FF} & \text{if } s \in \{\text{PH, PF}\}, \\ \text{SSS} & \text{if } s = \text{SCH}, \\ s & \text{otherwise.} \end{cases}$$

$$t_2(s) = \begin{cases} \text{Y}\varepsilon & \text{if } s \in \{\text{EE, IE}\}, \\ \text{D}\varepsilon & \text{if } s \in \{\text{DT, RT, RD, NT, ND}\}, \\ s & \text{otherwise.} \end{cases}$$

$$t_3(s,i) = \begin{cases} s[i:i+1] \to \text{AF} & \text{if } s[i] \text{ is a vowel and } s[i:i+1] = \text{EV}, \\ s[i:i+1] \to \text{A} & \text{if } s[i] \text{ is a vowel and } s[i:i+1] \neq \text{EV}, \\ s[i] \to \text{G} & \text{if } s[i] = \text{Q}, \\ s[i] \to \text{S} & \text{if } s[i] = \text{Z}, \\ s[i] \to \text{N} & \text{if } s[i] = \text{M}, \\ s[i] \to \text{N} & \text{if } s[i] = \text{K and } s[i+1] = \text{N}, \\ s[i] \to \text{C} & \text{if } s[i] = \text{K and } s[i+1] \neq \text{N}, \\ s[i:i+2] \to \text{SSS} & \text{if } s[i] = \text{S and } s[i+1] = \text{C and } s[i+2] = \text{H}, \\ s[i:i+1] \to \text{FF} & \text{if } s[i] = \text{P and } s[i+1] = \text{H}, \\ s[i-1] & \text{if } s[i] = \text{H and neither } s[i-1] \text{ nor } s[i+1] \text{ is a vowel}, \\ s[i-1] & \text{if } s[i] = \text{W and } s[i-1] \text{ is a vowel}, \\ s[i] & \text{otherwise.} \end{cases}$$

$$t_4(s) = \begin{cases} s[n_s] \to \varepsilon & \text{if } s[n_s] \in \{S, A\}, \\ s[n_s - 1 : n_s] \to \text{Y} & \text{if } s[n_s - 1 : n_s] = \text{AY}, \\ s & \text{otherwise.} \end{cases}$$

Algorithm 1 NYSIIS Encoding

procedure h_{nysiis}(s)
    $s = uppercase(s)$
    Remove any character $s[i]$, that is not in Σ
    $s_1 = t_1(s_1)$
    $s_3 = t_2(s_3)$
    while $i \leq n_s$ **do**
        $s = t_3(s[2:n_s], i)$
        $i = i + 1$
    end while
    $s_3 = t_4(s_3)$
    $s = s_1 s_2 s_3$
end procedure

The resulting string is the phonetic code for the input string according to the NYSIIS algorithm. Observe, that if we have the preprocessed data (i.e. uppercase names with only English alphabet characters), we can ignore the first two steppes. Thus, each encoding can be computed in $O(n)$ time.

The NYSIIS algorithm is primarily used for name matching, but it can also be used for other types of fuzzy string matching. It has been shown to improve the matching accuracy by 2.7% over the classical Soundex algorithm [RJ07].

2.4.2. Soundex

The Soundex algorithm is also a phonetic algorithm used for indexing and matching names based on their pronunciation. It was developed by Robert C. Russell and Margaret King Odell in 1918 and has since been

widely used for various purposes such as genealogy, record linkage, and information retrieval [Knu98].

The algorithm can be described as a hash function $h_{soundex} : \Sigma \to \sigma$, that transforms the name s into its phonetic code. In case of Soundex algorithm Σ is the English alphabet, whereas σ consists of the numbers from 0 to 9. Let's s_1 be the first letter and s_2 the rest of the name s, then the encoding is performed only on the s_2. s_1 is then concatenated to the start of the emerged phonetic code. The transformation function $t(c)$ for each character $c = s[i]$ is defined as follows:

$$t(c) = \begin{cases} 1 & \text{if } c \in \{\text{B, F, P, V}\}, \\ 2 & \text{if } c \in \{\text{C, G, J, K, Q, S, X, Z}\}, \\ 3 & \text{if } c \in \{\text{D, T}\}, \\ 4 & \text{if } c \in \{\text{L}\}, \\ 5 & \text{if } c \in \{\text{M, N}\}, \\ 6 & \text{if } c \in \{\text{R}\}, \\ c & \text{otherwise.} \end{cases}$$

Algorithm 2 Soundex Encoding

procedure h_{SIMPLEX}(s)
  $s \leftarrow$ uppercase(s)
  Remove any character $s[i]$ that is not in Σ
  $r \leftarrow [\cdot, \cdot, \cdot, \cdot]$
  $r[0] \leftarrow s[0]$
  $l \leftarrow t(s[0])$
  $j \leftarrow 1$
  while $i \leq n_s$ **do**
    $p \leftarrow t(s[i])$
    if $p \neq s[i]$ **then**
      if $p \neq l$ **then**
        $r[i] \leftarrow p$, $l \leftarrow p$, $j \leftarrow j + 1$
      end if
    else
      if $p \notin \{\text{H, W}\}$ **then**
        $l \leftarrow \varepsilon$
      end if
    end if
    if count $= 4$ **then**
      break
    end if
  end while
  if count < 4 **then**
    $r[\text{count} : 4] \leftarrow 0$
  end if
  return r
end procedure

The resulting string is the Soundex code for the input name. Observe, that we mainly loop only one time through the given name-string which gives us the time complexity of $O(n)$.

For example, the Soundex code for the name "Smith" is S530, because the first letter "S" is kept and the remaining letters are assigned their respective codes 5 and 3. The resulting string is always padded with a zero to obtain three groups.

2.4.3. Double Metaphone

Double Metaphone is a phonetic algorithm used to encode words based on their pronunciation. It was developed by Lawrence Philips in 1990 as an improvement over the original Metaphone algorithm. The Double Metaphone algorithm is designed to generate phonetic encodings for a given word, which can be used to improve matching accuracy and reduce false positives in various applications such as record linkage, information retrieval, and data mining.

It is called "Double" because it can return both a primary and a secondary code for a string; this accounts for some ambiguous cases as well as for multiple variants of surnames with common ancestry. For example, encoding the name "Smith" yields a primary code of SM0 and a secondary code of XMT, while the name "Schmidt" yields a primary code of XMT and a secondary code of SMT—both have XMT in common.

Original Metaphone contained many errors and was superseded by Double Metaphone, and in turn Double Metaphone and original Metaphone were superseded by Metaphone 3[2].

Here we outline the main algorithm informally. The algorithm phonetically codes words by reducing them to 16 consonant sounds: B, X, S, K, J, T, F, H, L, M, N, P, R, 0, W, Y. Zero represents the "th" sound; X stands for "sh", and the others represent their usual English pronunciations. The vowels A, E, I, O, U are also used, but only at the beginning of the code. This procedure summarizes most of the rules in the original implementation[Phi00]:

Procedure

- Drop the second letter of doubled letters, except C.
- If the word begins with KN, GN, PN, AE, WR, drop the first letter.
- Drop B at the end of a word after M.
- C $\rightarrow$ X in CIA or CH; C $\rightarrow$ S in CI, CE, or CY; C $\rightarrow$ K otherwise.
- D $\rightarrow$ J in DGE, DGY, or DGI; D $\rightarrow$ T otherwise.
- Drop G in GH and if not at the end or before a vowel in GN or GNED; G $\rightarrow$ J before I or E or Y if not double GG; G $\rightarrow$ K otherwise.
- Drop H after a vowel and if no vowel follows.
- Drop K after C.
- P $\rightarrow$ F in PH.
- Q $\rightarrow$ K.

[2]Metaphone 3 is also developed by Lawrence Phillips and was commercialized subsequently. Hence, we will not observe it in our work.

- S → X in SH or SIO or SIA.
- T → X in TIA or TIO; T → 0 in TH; T is dropped in TCH.
- V → F.
- If the word begins with WH, drop H; drop W if not followed by a vowel.
- If the word begins with X, then X → S; X → KS otherwise.
- Drop Y if not followed by a vowel.
- Z → S.
- Vowels are kept only when they are the first letter.
- In all other cases, the letters do not change.

Examples

- ALEXANDRE → ALEKSANTRE → ALKSNTR
- ALEKSANDER → ALEKSANTER → ALKSNTR

The resulting Double Metaphone codes may be further processed or compared to other codes using various techniques such as similarity measures or clustering algorithms.

The Double Metaphone algorithm is designed to be language- and dialect-independent, and it can handle many variations in English pronunciation and spelling. The algorithm may not be suitable for languages other than English or for very short or very long words. Nevertheless, the Double Metaphone algorithm has been shown to be effective in reducing errors and improving matching accuracy in various applications.

2.5. Benchmarking

In order to simulate the validated data we use the "*name.basics*" database provided by IMDB.com, Inc non-commercial use. This data is a perfect choice for our research for couple of reasons. First, the provided names are the names of real persons and are open to everyone, which provides us with a perfect opportunity to test our framework avoiding any compliance issues. Second, it is big enough to create arbitrary big subsets of it for testing purposes. The dataset consist of following attributes:

Column name	Type	Description
nconst	string	Alphanumeric unique identifier of the name/person
primaryName	string	Name by which the person is most often credited
birthYear	YYYY format	Year of birth
deathYear	YYYY format or '\N'	Year of death if applicable, else '\N'
primaryProfession	array of strings	The top-3 professions of the person
knownForTitles	array of tconsts	Titles the person is known for

Table 2.1.: Description of the IMDb *"name.basics"* dataset

For our research we're particularly interested in "primaryName" field of the given data. The data originally contains $12.310.514$ unique names. We prepare it in the following way: first, we split the "primaryName" by spacing between words, claiming the first word to be the first name and all others to be the last name; second, we drop every emerged last name, that contain three or less characters; third, we chose 10^4 last names at random. We do the second step on one hand, because the data is not necessary contains full names and by dividing the column into first and all other words in the first step, we get some empty last names. On the other hand, because the last names that contain three or less letters are rather specific and small set of names overall. We also use just the subset of the whole dataset in the third step for couple of reasons. First, since we aim to create an optimal matching for each of the outlined metrics and algorithms, we will use the greedy search since we want to compare each name from the LHS to each name from the RHS and remember the pair with the lowest dissimilarities. Even a comparison of 10^4 unvalidated names against the complete "*name.basics*" dataset will take days. We're going to discuss the computational complexity in the Section 2.5.1. Second, the subset is already big enough to make the given task quite difficult and tedious to solve it per hand. Third, as we will see in the Chapter 4, our main goal is to prove the concept and for it we create several models for data classification, incrementally increasing their complexity. This task also requires time and computational power. That may involve the computational capacities over those, that we have at our disposal. We will show how to up-scale the proposed solution, but will bound ourselves for smaller subsets of the "*name.basics*" dataset nevertheless. Ultimately we end up with the dictionary **RHS**, that consists of 10^4 unique last names. This dataset will simulate our CRM data. Additionally we encode every unique name with according label using Label Encoding technique by Sklearn [PVG+11] as satellite data. It will provide us the possibility to quickly recognize, if the match was successful and will be useful for the Name Classification task, that we address in the next Chapter.

For validation purposes we chose 10^3 last names from our RHS dictionary uniformly random. We then create four variations of each name that going to imitate human-like spelling mistakes. We use term "imitate" to address the fact, that for production of the large number of such misspellings we use GPT-3.5 [Ope23] - a Large Language Model released by OpenAI, version from Match 23, 2023. The emerged dictionary **LHS** with according labels as its satellite data will simulate our unvalidated third-party data, that we want to match against the RHS. Note, that it is a difficult task to find the exact misspellings of

names, since there's no general definition, of what a misspelled name should exactly look like. Hence, we rely on the training of the underlying LLM as well as the author's own experience to create a closest to reality dictionary of misspelling names.

2.5.1. Benchmarking analysis

We execute the exact dictionary search by comparing each name of LHS dictionary to every name of the RHS dictionary. We're particularly interested in the one-to-one matches. Thus, we always chose the first occurrence. We claim a match if the corresponding labels are the same and a mismatch if they aren't.

In the Table 2.2 we summarize the matching results for each algorithm, introduced in this Chapter.

Technique	Complexity, ξ	Number mismatches	
		Regular metric	Modified metric
Levenshtein	$O(\|p\|\|s\|)$	199 (4.97%)	171 (4.27%)
Damerau-Levenshtein	$O(\|p\|\|s\|)$	159 (3.97%)	**131 (3.27%)**
Hamming	$O(max(\|p\|, \|s\|))$	884 (22.10%)	832 (22.80%)
Jaro-Winkler	$O(\|p\| + \|s\|^2)$	167 (4.17%)	-
Jaccard	$O(\|p\| + \|s\|) + O(min(\|p\|, \|s\|))$	617 (15.43%)	417 (10.42%)
Supervised similarity	-	792 (19.8%)	-
NYSIIS	$O(\|p\|)$	1377 (34.42%)	-
Soundex	$O(\|p\|)$	256 (6.4%)	-
Double Metaphone	$O(\|p\|) + O(\|p\|)$	**91 (2.28%)**	-

Table 2.2.: Benchmarking results for exact dictionary searching

As we can see, we can achieve some good results using common methods for fuzzy string matching. Let's now discuss the advantages and drawbacks of each method.

Edit distances We first observe, that the simple greedy approach of matching the names between two dictionaries involves iteration over the RHS while also iterating over the LHS, which creates a double loop. Thus we end up with $O(|\text{LHS}| \cdot |\text{RHS}| \cdot \xi)$ complexity for each run. The match is then defined by $min(\text{ED}(p_i, s_j))$ for $i \leq |\text{LHS}| \in \mathbb{N}$, $j \leq |\text{RHS}| \in \mathbb{N}, p_i \in \text{LHS}, s_i \in \text{RHS}$.

Since the regular Edit Distance produces the integer values, they're also often a source of ambiguity between multiple matches, that share the same distance, since we always choose the first occurrence. To avoid such collisions we **modify** the Edit Distances by using their normalized counterpart

$$\text{ED}_{Norm}(p, s) := 1 - \frac{\text{ED}(p, s)}{max(|p|, |s|)} \in [0, 1] \tag{2.5}$$

for Levenshtein, Damerau-Levenshtein und Hamming distances, where 0 indicates the complete similarity. We see, that normalized Edit Distance provides us with more distinguishable scoring, allowing less ambiguity than the regular Edit Distance. We observe, that Normalized Damerau-Levenshtein distance of all Edit Distance metrics produces the lowest number of mismatches.

As for Jaccard similarity coefficient, we observe that it works similar to comparing two signature vectors $signat(s)$ and $signat(p)$ for two strings s and p. We modify it by comparing frequency vectors instead. We thus relax the requirement of transforming strings into sets, by transforming them into sets that allow duplicates. E.g. If Σ is an English alphabet, s="*hamming*" will be encoded as a vector of length $|\Sigma|$, that gets the number of occurrences of the corresponding letter at each position. We can represent s as a hash map (or dictionary) of the form s_h={'h':1, 'a':1, 'm':2, 'i':1, 'n':1, 'g':1, 'b':0, 'c':0, ...}. Now if we want to compare it to the other sting p="*hampton*", we subtract the values of the vector p_h from the s_h and get sp_h={'h':0, 'a':0, 'm':1, 'i':1, 'n':0, 'g':1, 'p':-1, 't':-1, 'o':-1 ...}. Taking the sum of the absolute values of

sp_h will give us a relaxed form of the intersection of two sets of string characters, which also penalizes for differences. Dividing by the total sum of values $sum(p_s) + sum(h_s)$, which represents the relaxed union of two sets of string characters, we get the modified Jaccard similarity coefficient. We observe that this approach also produces better name matches by creating much more distinguishable scoring of two name-strings. We show the distributions of matches for each Edit Distance in Appendix A. Please note, that all of the outlined histograms contain gaps between the bars for better visual perception alone and don't indicate the absence of matches in this areas. Rather, the matches are grouped around the middle values of each bar. That's especially the case for Jaccard and Jaro Winkler Similarity coefficients, since their output it continuous. Levenshtein, Damerau-Levenshtein and Hamming distances produce categorical data in $\mathbb{N}$ and thus don't have any intermediate values.

As can be seen from the distribution of matches in Appendix A.2 - A.5, the normalized distribution usually wins more correct matches on names, whose regular Edit Distance is > 0. In general we observe the decrease of False Matches along with increasing of the Edit Distance for modified metrics in contrast to the regular ones. This tells us, that the matching in greedy way is more precise, when using normalized distance.

Supervised similarity distance For the supervised similarity metric we use the algorithm provided by Dedupe.io based on Bilenko's work [Bil04]. The algorithm provides us with moderate results ending up with slightly better result than Hamming distance. We provide the active learning table in the Appendix A.1. We label in summary 50 possible matches, ending up with 25 positive (y), 16 negative (n) and 4 unsure (u) labels. We also don't specify any ξ for matching of two strings, since the matching of corresponding labels happens by hand.

Phonetic encodings The phonetic encodings avoid the approximate matching problem by reducing it to exact matching. They also offer the opportunity to pre-compute the phonetic representations for RHS dictionary beforehand. The proposed algorithms are fairly fast for in the sense of time complexity for name encoding. The exact matching (or data join) is also represented in every common programming language with fairly fast solutions (e.g. SQL)[3]. Analysing the matching distribution for each of the proposed encoding algorithms represented in the Figure 2.1, we note, that although the produced matches are showing very good results (e.g. Double Metaphone algorithm shows only 91 mismatches), they cannot provide us with one-to-one matches, that we aim for. Rather they produce the clusters for each match, because of the ambiguity of the name representation. We also note, that Double Metaphone requires exact matching among both produced encodings, which requires the four times of the time and space complexity.

At this point we could modify our search, by employing the greedy search using Edit Distances on the provided subsets for each name in order to achieve one-to-one matches.

In summary, each of the represented techniques has its own advantages and disadvantages. The main problem remains the same, no matter how fast is the underlying algorithm for fuzzy matching, we have to do the approximate dictionary searching. Since the phonetic algorithms reduce the search dimensionality, we will base our solution in Chapter 4 on the same idea. We will try reduce the search space, so the greedy search using any of the introduced Edit Distances can be executed in a feasible time. Our goal would be to outperform the presented algorithms both in computational time and in accuracy using a neural network approach.

[3] modern servers can use HASH JOIN and MERGE JOIN which are faster than $O(|LHS||RHS|)$

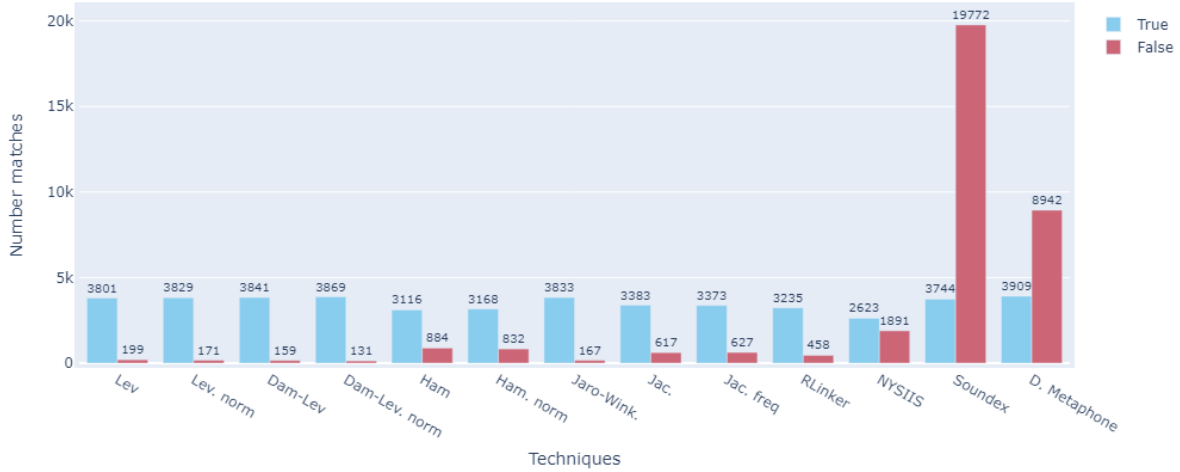


Figure 2.1.: Matching summary for each metric

2.6. Conclusion

In this chapter we presented and analysed performance of many common metrics and algorithms for FNM problem. We saw, that there was a good amount of research in this field over the last decades. We introduced the most common algorithm to solve this problem including Edit Distances, Similarity Coefficients and Phonetic Algorithms. We have seen, that they produce accurate matches and produce a good quantification to the idea of similarity between two strings. However, they share a common problem of approximate dictionary searching - its time complexity. There were a lot of research in the past years on how to solve this problem including prefix trees, neighborhood generation, metric-space pivoting methods, pattern partitioning and vector-space frequency-distance methods [Boy11]. The variety of this methods shows, that there're no unique solution to this problem and there's still a continuous research happening in this field. In the coming Chapters we will devote ourselves to another approach. Namely, we want to utilize modern computational capacities and employ the neural networks for FNM. Using this approach we aim to reduce the search space and use the introduced algorithms for the quantification of the produced matches.

3. Classification Task Using Neural Network Approach

In this chapter, we will motivate and explore the state-of-the-art neural network concepts that will contribute to our final solution. We start with the technical set-up and explore the existence of the solution to our task within the required framework in Section 3.2. We establish the loss function, that we are going to utilize for the training of the neural network for classification purposes in Section 3.3. Afterwards, we explore the broad field of tools that we're going to utilize throughout our experiments in Chapter 4. In particular, we introduce the concept and different architectures of neural networks in Section 3.5. Specifically, we introduce the main concept of a feedforward neural network in Section 3.5.1 and its variations in the form of RNNs in Section 3.5.3, LSTMs in Section 3.5.4, and CNNs in Section 3.5.5. We then explore the possibilities of the name representation as a mel spectrogram in Section 3.6 and introduce the state-of-the-art text-to-speech neural network called Tacotron2 in Section 3.6. Lastly, we summarize the explored knowledge and conclude the Chapter by establishing the idea-to-prove.

3.1. Motivation

Prior to delving into the technical aspects, we shall embark on a preliminary exploration of the matter and solicit ideas from a human standpoint. Let us consider the objective of determining if two names are associated with the same person. Typically, we encounter instances similar to the following: "Abdil Ben Halyl" versus "Abdil Benhalil." Naturally, real-world data also encompasses supplementary information concerning the individual, such as their country of origin, professional domain, or email address. Nevertheless, none of this data furnishes us with more direct knowledge than their name, as the name serves as a representation of the person's identity. We might employ the individual's country, city, or field of work as satellite information to support our decision. However, we must first establish the premise that we are potentially examining the same person. Admittedly, certain identifiers like the email address are even more reliable for this task, and we would gladly match each record to the internally validated data solely based on an email. Nonetheless, the real world invariably entails uncertainty that we must account for. For instance, while an individual's name functions as a unique identifier, the same cannot be said for their email address. This is especially true when dealing with customers who are represented by distinct accounts or even products represented by their respective names. Once such data is collected manually or via internet crawlers that lack the capability to verify the gathered information, we encounter a *Fuzzy Matching Problem*. This predicament commonly arises when matching two records that are uniquely identified by a particular field. We will refer to this field as a "name" to establish a connection with the task at hand. This designation is also sensible, because names typically lack contextual or semantic meaning, and it is challenging to establish specific rules to differentiate a genuine name from a nickname, for example.

Despite all these considerations, it is still highly probable that a human can identify the association with a given person based on their misspelled name, even if not encountered that name previously. We will

encounter this problem in similar way a human does. On one hand, we want to consider the spelling mistakes and on other hand, the similarity in the pronunciation. To entangle the both concepts as well as the neural network interface, that relies on numeric inputs, we consider the phonetic encoding of the names as mel spectrograms. We will show, that the classification task using the mel spectrograms is mostly the same as the common task of image recognition. However the generation of mel spectrograms requires the spoken audio wave-signals. In the Section 3.6 we going to show how we can generate mel spectrograms from the written text while skipping the part of the audio generation.

3.2. Set-up

First we set-up the framework and formalize our task.

Deep convolutional neural networks (CNNs) have achieved remarkable success in various applications, especially in visual recognition tasks, see, e.g., [LBH15], [KSH12a]. The aim of this theses is to show that CNN based network architectures can also be applied for fuzzy matching purposes.

To achieve this goal we state the *Fuzzy Matching Problem* as an image (mel spectrogram) classification task over the dictionary W which we formalize as follows:

Let $d_1, d_2 \in \mathbb{N}$, $N = |W|$ and let $(\mathbb{X}_1, Y_1), \dots, (\mathbb{X}_n, Y_n) \in X \times Y$ be independent and identically distributed random variables with values in

$$\mathbb{R}^{d_1 \times d_2} \times \mathbb{N}_0^N.$$

Each class Y is represented as a number $c \leq N \in \mathbb{N}_0$ in the finite number of classes, that correspond to the names in the dictionary. We describe a (random) image from (random) class Y by a (random) matrix $\mathbb{X}$ with d_1 rows and d_2 columns, which contains at position (i, j) the grey scale value of the pixel of the image at the corresponding position.[1] Our aim is to predict Y given $\mathbb{X}$. We assume that the pair (X, Y) arises due to the (joint) probability distribution $\mathbb{P}$. Our goal is to find a prediction function (classifier) $h : X \to Y$ s.t. the risk of misclassification is small:

$$\inf_h R(h) := \mathbb{P}[h(\mathbb{X}) \neq Y] = \mathbb{E}[\mathbb{I}_{\{h(\mathbb{X} \neq Y\}}],$$

where $\mathbb{I}_{\{\cdot\}}$ is the indicator function. To achieve this goal we would like to take

$$h^*(\mathbb{X}) = \arg\max_c \mathbb{P}(Y = c|\mathbb{X}). \tag{3.1}$$

The equation (3.1) is the *Bayes classifier* (or Bayes optimal function) in multi-label case. Because we do not know the distribution of (X, Y), we cannot find h^*. Instead, we estimate h^* by using the training data $D_n = (\mathbb{X}_1, Y_1), \dots, (\mathbb{X}_n, Y_n)$. A popular approach is estimating h^* by the empirical risk minimization, i.e.,

$$R_n(h) = \arg\min_{h \in C_n} \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}\{h(\mathbb{X}_i) \neq Y_i\},$$

where C_n is a given class of classifiers. In practice R_n is not computational feasible, since minimizing the empirical risk with the 0/1 loss over C_n is NP hard [BJM06]. By replacing the number of misclassifications by a surrogate loss l, one can overcome computational problems. Instead of a class of classifiers C_n, we consider a class of real-valued functions F_n. As the function space F_n we choose a class of CNNs based neural networks. For a given loss l, we are searching for an estimate $\hat{h}_n \in F_n$ such that the surrogate empirical risk

$$\frac{1}{n} \sum_{i=1}^{n} l(Y_i, \hat{h}_n(\mathbb{X}_i)) \tag{3.2}$$

[1] In terms of a spectrogram, we get d_1 frequencies and d_2 time steps.

is small. Which is equal to

$$\arg\min_{\hat{h}_n \in F_n} \frac{1}{n} \sum_{i=1}^{n} l(Y_i, \hat{h}_n(\mathbb{X}_i)) \tag{3.3}$$

There are different loss functions to choose (see [HTF09] for an overview). In the context of CNNs and image classification it is a standard to use *cross-entropy loss*, which we define in the next Section.

One question remains still: Is it possible to create a successful classifier using a neural network?

Definition 3.2.1 (VC dimension) *Let H be a family of functions from X to Y. The Vapnik-Chervonenkis (VC) dimension of H is the maximal number K of points $x_1, ..., x_K \in X$ such that for any assignment of labels from Y to these points, there exists $h \in H$ such that h correctly classifies all these points [UP20].*

Using this definition, we can introduce the following result:

Corollary 3.2.0.1 *Let H be a class of prediction functions with finite VC dimension. Then the risk values $R(h_N)$ of minimizers h_N of the empirical risk of misclassification converge to the Bayes error $R(h^*)$ for the Bayes optimal function with high probability if the number of samples N tends to infinity [UP20].*

Now, since we're using a computer with underlying hardware and software, we cannot hope to represent every number and parameter arbitrary precise. If we consider a neural network[2] $h_{V,E,\sigma} \in F_n$, where V is the set of layers, E is the set of tunable parameters and σ is the sigmoid activation function, it can be shown that the VC dimension of such a network is the number of tunable parameters squared. Since in practice we only consider networks in which the weights have a short representation as floating point numbers with $O(1)$ bits, by using the discretization trick one can show that such networks have a VC dimension of $O(|E|)$ [SSBD14]. Thus, the Corollary 3.2.0.1 holds and we're assured, that our task is indeed possible to achieve if we're able to create enough valid training data. We will show in Section 3.6, that our approach to mel spectrogram generation allows us to create arbitrary big training sets if necessary.

[2] We will introduce the concept of a neural network in more detail in the Section 3.5.1.

3.3. Loss-functions and training

Loss functions play a crucial role in neural networks and machine learning, serving as objective measures of model performance. When training neural networks, optimization algorithms like gradient descent or Adam are typically employed to adjust the model parameters and minimize the loss function. This function quantitatively measures the discrepancy between the predicted output of the neural network and the true target value. Minimizing the loss function minimizes the risk of misclassification, enabling the model to make more accurate predictions and improve its overall performance. To further enhance generalization and prevent overfitting[3], loss functions can incorporate regularization techniques such as L1 or L2 regularization.

The choice of a specific loss function depends on the task or objective of the neural network. Different tasks like regression, classification, or generative modeling necessitate distinct loss functions tailored to capture the desired properties of the output. In this thesis, our aim is to develop multiple neural networks for classification purposes, for which we will utilize the cross-entropy loss.

Cross-Entropy Loss

Entropy represents the level of uncertainty or randomness associated with a random variable or probability distribution. It measures the amount of information required to describe or predict the outcome of an event or system state.

The concept of entropy originates from information theory, pioneered by Claude Shannon. Shannon's entropy provides a mathematical measure of the average information content, measured in bits, needed to encode or transmit a message based on a given probability distribution. It is closely related to Shannon information, which represents the expected value of self-information for a random variable. Self-information quantifies the average surprise or unexpectedness of the random variable. In the context of a real number $b > 1$ and an event x with probability $\mathbb{P}$, the information content can be defined as follows [McM07]:

$$I(x) := -log_b(\mathbb{P}(x)) \tag{3.4}$$

Thus, given a random variable X and probability distribution F_X the self-information of measuring X as outcome x is defined as:

$$I_X(x) := -log_b(F_X(x)) \tag{3.5}$$

This form creates a good basis to understand the idea behind entropy. We introduce the definition of the entropy in the discrete case.

Definition 3.3.1 (Entropy) *Given a random variable X on $(\Omega, \mathcal{F}, \mathbb{P})$ and defined by a probability distribution $F_X(x)$, the entropy of X for $b > 1$ is given by*

$$E(X) = -\Sigma_{x\in\Omega} F_X(x) log_b(F_X(x)) \tag{3.6}$$

[3]The effect to perform too accurately on a given training set at the risk of bad prediciton performance on other data [UP20]

If the $b = 2$ in the above equation, then the entropy is expressed in *bits*. If $b = e$, then the entropy is expressed in *nats*. Those are the common units for measuring the entropy.

Now if we want to measure the the cross-entropy between two probability distributions F_X and F_Y over the same underlying set of events, we get the following definition:

Definition 3.3.2 (Cross-entropy) *Given two random variables X, Y on $(\Omega, \mathcal{F}, \mathbb{P}_X)$ and $(\Omega, \mathcal{F}, \mathbb{P}_Y)$ respectively and defined by a probability distribution F_X and F_Y. The cross-entropy loss between X and Y for $b > 1$ is given by*

$$l(X, Y) = -\Sigma_{x \in \Omega} F_Y(y) log_b(F_X(x)) \tag{3.7}$$

Since we don't have any previous knowledge regarding the underlying distributions, we instead use the empirical formulation of the cross-entropy in order to measure the loss between the predictions produced by the model during the training and the true values.

Definition 3.3.3 (Softmax) *The function $s : \mathbb{R}^N \to [0, 1]^N$ is called **Softmax***

$$s(\mathbf{x}) = [s_1, s_2, \cdots s_N], \quad s_i = \frac{exp(x_i)}{\sum_{j=1}^{N} exp(x_j)}, \quad \forall i \in \{1, 2, ..., N\} \tag{3.8}$$

where $\mathbf{x} = [x_1, x_2, ..., x_N]$ represents the input vector.

The softmax function computes the probability distribution over N elements, ensuring that each element of the output vector $s(\mathbf{x})$ lies within the range $[0, 1]$ and the sum of all elements in $s(\mathbf{x})$ is equal to 1. We're going to use the Softmax function to simulate the probability distribution of the model prediction. More specifically, we define the empirical cross-entropy as follows:

Definition 3.3.4 (Empirical Cross-entropy) *Let's N be the number of classes, $\hat{x} = s(x) \in \mathbb{R}^N$, $y \in R^N$. Then the **unreduced** cross-entropy loss is described as*

$$l(x, y) = -\Sigma_{n=1}^{N} w_n log(\hat{x}_n) y_n = -w^\top (log(\hat{x}) \odot y), \tag{3.9}$$

where $w \in \mathbb{R}^N$ is the weight vector, that balances the importance of the class in case of the unbalanced training data, the log is the logarithm to basis $b = e$ and $\odot$ denotes the elemenwise vector multiplication.

Training

Now that we defined our classification function can formulate our classification problem formally. Let $\hat{h}_n \in F_n$ be our classificator as described by Equation (3.3). Then the formal training problem is defined as follows

$$\min_l l(\hat{h}_n(\mathbb{X}), y) + r \tag{3.10}$$

where r is some regularization term.

Since $\hat{h}_n$ represented as a CNN, we're going to "train" it by assigning the according weights for the network parameters, that solve the described problem. One possibility to learn the weights of the network is by slowly approaching the target value using Stochastic Gradient Descent (SGD) [UP20]. Over the past years there were a major development of the numerous techniques to solve the stated problem more efficiently. Adaptive gradient methods, such as AdaGrad [DH11], Adam [KB17] and AMSGrad [RKK19] have become a default method of choice for training feed-forward and recurrent neural networks [XBK$^+$15]; [RMC16]. For our task we will use the AdamW algorithm described by [LH17]. It has shown to be an improved version of the Adam algorithm with l_2-regularization [ZLCO22].

3.4. Mel spectrogram generation

In order to convert an audio signal into its visual representation, we introduce the general concept as follows. Given a time an audio signal $a(t) \in \mathbb{R}$ we define the Time-Frequency Representation as

$$TF_a(t,\omega) = \int a(\tau)\phi^*_{t,\omega}(\tau)d\tau =< a, \phi_{t,\omega} >,$$

where $\phi_{t,\omega} \in L^2(\mathbb{R})$ represents the basis functions (also called the Time-Frequency atoms) and $*$ represents the complex conjugate [SDJ09]. Short-time Fourier transform (STFT) [Grö01], wavelets [Stu07], [Dau92], and matching pursuit algorithms [MZ93] are typical examples in this category.

In the continuous-time case, the audio signal is multiplied by a window function which is nonzero for only a short period of time. The Fourier transform (a one-dimensional function) of the resulting signal is taken, then the window is slid along the time axis until the end resulting in a two-dimensional representation of the signal. This can be formalized as Short-Time Fourier Transform (STFT):

$$STFT_a(t,\omega) = \int a(\tau)w(\tau - t)e^{-i\omega\tau}d\tau.$$

Where w is a (smooth) window function to extract the short-time audio-signal from. Popular choices for the window function are the Rectangular, Triangular, Cosine or Hann window functions[4].

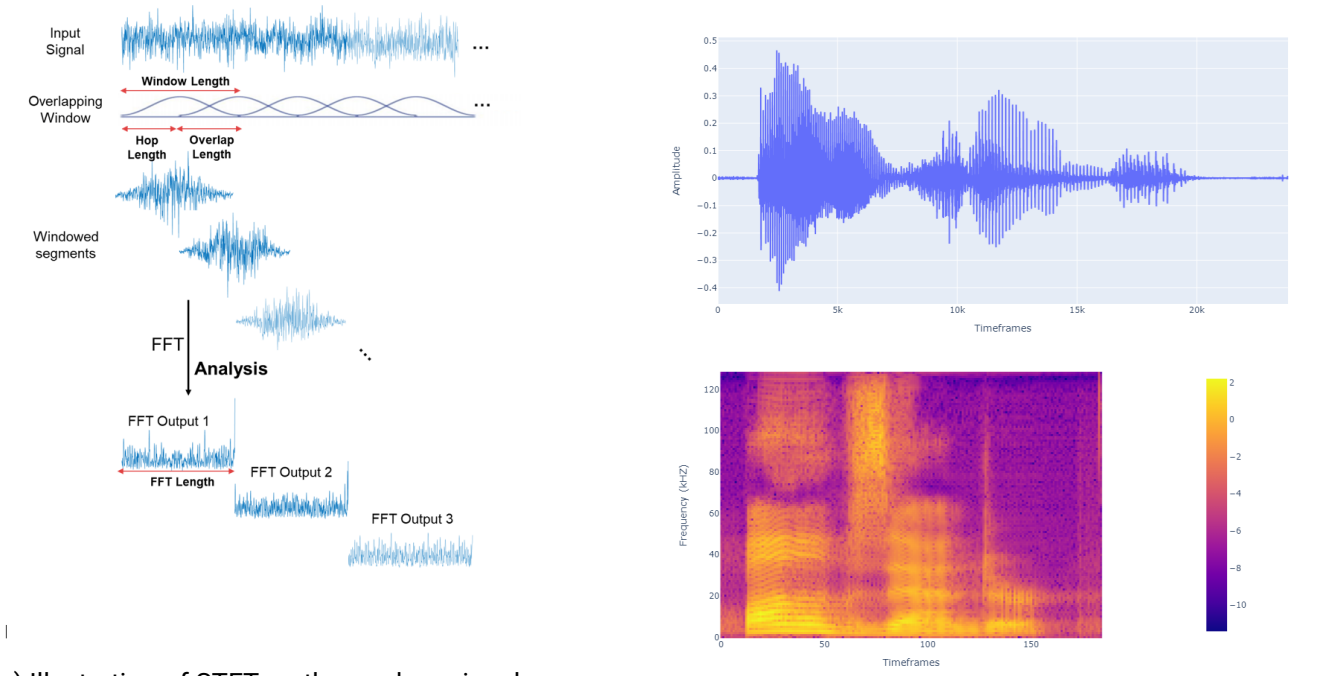
We now consider the discrete case of the STFT. Let $a : \mathbb{N} \to \mathbb{R}$ be a real-valued discrete-time (DT) signal of length $T \in \mathbb{N}$ (max. time) obtained by equidistant sampling from the underlying audio-signal with respect to a fixed sampling rate F given in Hertz. Furthermore, let $w : \mathbb{N} \to \mathbb{R}$ be a sampled window function of length $N \in \mathbb{N}$. For example, in the case of a rectangular window one has $w(n) = 1$ for $n \in \{0, \cdots, N-1\}$. The length parameter N determines the duration of the considered sections, which amounts to N/F seconds. One also introduces an additional parameter $H \in \mathbb{N}$, which is referred to as the *hop size*. The hop size parameter is specified in samples and determines the step size in which the window is to be shifted across the signal. With regard to these parameters, the discrete STFT of the signal a is given by [The19b]

$$STFT_a(m,k) := \sum_{n=0}^{N-1} a(n)w(m - nH)\exp\left(-i\frac{2\pi k}{N}n\right) \tag{3.11}$$

with $m \in \{0, \cdots, M\}$ and $k \in \{0, \cdots K\}$. The number $M := \left\lfloor \frac{L-N}{H} \right\rfloor$ is the maximal frame index such that the window's time range is fully contained in the signal's time range. Furthermore, $K = \frac{N}{2}$ (assuming that N is even) is the frequency index corresponding to the Nyquist frequency[5] [Lei11]. The complex number $STFT_a(m,k)$ denotes the k-th Fourier coefficient for the m-th time frame. Note that for each fixed time frame m, one obtains a spectral vector of size $K+1$ given by the coefficients $STFT_a(m,k)$. The computation of each such spectral vector amounts to a Discrete Fourier Transformation (DFT) of size N, which can be done efficiently using the Fast Fourier Transform (FFT) Cooley-Tukey algorithm [CT65].

[4] There're also many other choices that depend on the task at hand [Pra13]

[5] Nyquist frequency is the frequency whose cycle-length (or period) is twice the interval between samples, i.e. Nyquist rate for sampling a 22050 Hz signal is 44100 samples/second



(a) Illustration of STFT on the random signal [The19a]

(b) Name "Alexander" as audio-signal and spectrogram

Figure 3.1.: Spectrogram representation of the audio-signal

Lastly, we convert the obtained values into a spectrogram by squaring the magnitude of the STFT [The19a]:

$$spectrogram(m,k) = |STFT_a(m,k)|^2.$$

Figure 3.1a illustrates the process of the STFT. Figure 3.1b shows how the audio-signal of the name "Alexander" is represented using 23808 timeframes on the X-axis and normalized signal amplitude on the Y-axis, where -1 represents the lowest amplitude (silence) and 1 represents the highest amplitude (maximum loudness). The mel spectrogram shows the log magnitude, calculated using frame length 255 and frame step 128. The output frequency on the Y axis is plotted from 0 to Nyquist frequency.

The Mel Scale

Studies have shown that humans do not perceive frequencies on a linear scale. We are better at detecting differences in lower frequencies than higher frequencies. For example, we can easily tell the difference between 500 and 1000 Hz, but we will hardly be able to tell a difference between $10,000$ and $10,500$ Hz, even though the distance between the two pairs are the same.

In 1937, Stevens, Volkmann, and Newmann [SVN37] proposed a unit of pitch such that equal distances in pitch sounded equally distant to the listener. This is called the mel scale. A formula by O'Shaughnessy [O'S87] to convert h (Hertz) into m (Mels) is:

$$m = 2595 \log_{10}\left(1 + \frac{h}{700}\right) \tag{3.12}$$

The time-frequency representation of a signal describes the energy distribution along the frequency axis at each time instant. We will utilize this idea to extract representative features from the given audio signal.

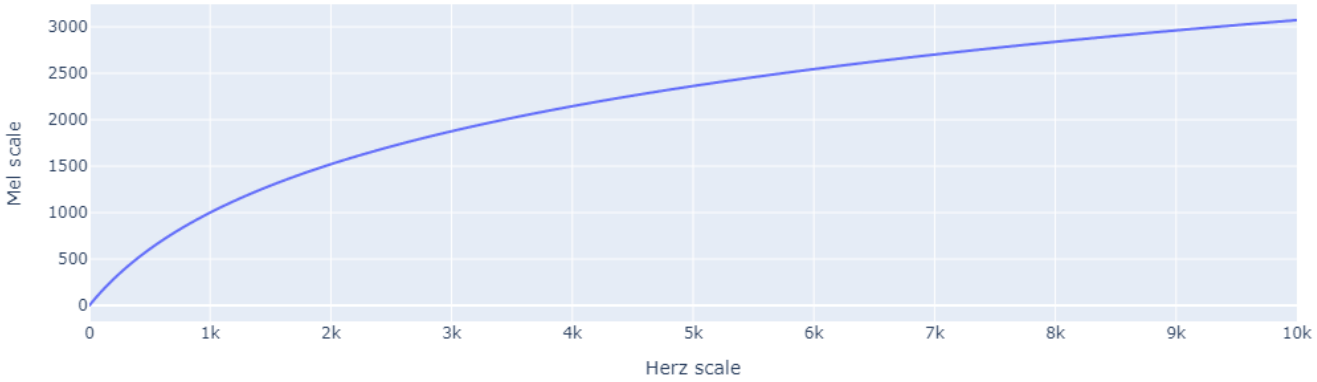


Figure 3.2.: Plots of pitch. Mel scale versus Hertz scale (3.12)

Additionally, it enables the conversion of the textual representation of names into their corresponding numerical counterparts. However, the challenge that remains is obtaining the audio signals for each name in the RHS dictionary.

To tackle this issue, we focus on the recent advancements in the field of Text-to-Speech transformation using Recurrent Neural Networks (RNNs). Several models have emerged in recent years, e.g. Tacotron [WSS+17a], Tacotron2 [SPW+17], Deep Voice [ACC+17] and many others, that facilitate the generation of mel spectrograms without the need to manually produce the audio signals. In this thesis, we adopt Tacotron2 as the underlying mechanism for generating mel spectrograms. An overview of the Tacotron2 model architecture will be presented at the end of this Chapter, where we delve into a comprehensive understanding of its constituent building blocks.

3.5. Neural Network Architectures

This Section is dedicated to the different neural networks architectures, that are applicable for our task. We start by declaring a simple feedforward architecture. Afterwards, we show how it can be utilized in some language models. We then carefully define the most important concepts that will help us to create our own classificators and contribute to the understanding of the workflow of the Text-to-Speech architecture of tacotron2 model. This concepts will include Recurent Neural Networks, Long Short-Term Models and Convolutional Neural Networks.

3.5.1. Feedforward neural networks

The idea behind neural networks is that many neurons can be joined together by communication links to carry out complex computations. It is common to describe the structure of a neural network as a graph whose nodes are the neurons, and each directed edge in the graph links the output of some neuron to the input of another neuron. We will restrict our attention to feedforward network structures in which the underlying graph does not contain cycles and stick to the description provided by [HTF09].

A feedforward neural network is described by a directed acyclic graph, $G = (V, E)$, and a weight function over the edges, $w : E \to \mathbb{R}$. Nodes of the graph correspond to neurons. Each single neuron is modeled as a simple scalar function, $\sigma : \mathbb{R} \to \mathbb{R}$. We will focus on three possible functions for σ: the sign function, $\sigma(a) = \text{sign}(a)$, the threshold function, $\sigma(a) = 1[a > 0]$, and the sigmoid function, $\sigma(a) = \frac{1}{1+\exp(-a)}$, which is a smooth approximation to the threshold function. We refer to σ as the "activation" function of the neuron. Each edge in the graph links the output of some neuron to the input of another neuron. The input of a neuron is obtained by taking a weighted sum of the outputs of all the neurons connected to it, where the weighting is determined by w.

To simplify the description of the calculation performed by the network, we further assume that the network is organized in layers. Specifically, we decompose the set of nodes into a union of disjoint subsets, $V = \bigcup_{t=0}^{T} V_t$, such that every edge in E connects a node in V_{t-1} to a node in V_t, for some $t \in [T]$. The bottom layer, V_0, is referred to as the input layer and consists of $n + 1$ neurons, where n represents the dimensionality of the input space. For each $i \in [n]$, the output of neuron i in V_0 is denoted as x_i. The last neuron in V_0 is referred to as the "constant" neuron, which always outputs 1. We denote the ith neuron of the tth layer as $v_{t,i}$, and the output of $v_{t,i}$ when the network is fed with the input vector x is denoted as $o_{t,i}(x)$. Thus, $o_{0,i}(x) = x_i$ for $i \in [n]$, and $o_{0,i}(x) = 1$ for $i = n + 1$.

We now proceed with the calculation in a layer-by-layer manner. Suppose we have computed the outputs of the neurons at layer t. Then, the outputs of the neurons at layer $t + 1$ can be calculated as follows. Let $v_{t+1,j} \in V_{t+1}$ be a neuron in the next layer. The input to $v_{t+1,j}$ when the network is fed with the input vector x is denoted as $a_{t+1,j}(x)$. It is obtained by summing the weighted outputs of all the neurons connected to $v_{t+1,j}$ in layer t, where the weights are determined by w:

$$a_{t+1,j}(x) = \sum_{r:(v_{t,r},v_{t+1,j})\in E} w((v_{t,r}, v_{t+1,j}))o_{t,r}(x)$$

The output of neuron $v_{t+1,j}$ in layer $t + 1$, denoted as $o_{t+1,j}(x)$, is obtained by applying the activation function σ to its input $a_{t+1,j}(x)$:

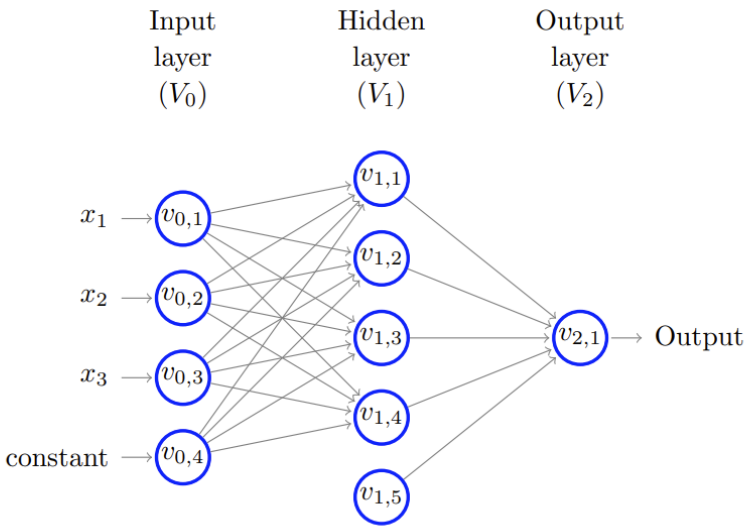


Figure 3.3.: Feedforward neural network [HTF09]

$$o_{t+1,j}(x) = \sigma(a_{t+1,j}(x))$$

In other words, the input to $v_{t+1,j}$ is a weighted sum of the outputs of the neurons in V_t that are connected to $v_{t+1,j}$, where the weighting is determined by w, and the output of $v_{t+1,j}$ is the result of applying the activation function σ to its input.

Layers $V_1, \ldots, V_{T-1}$ are often referred to as hidden layers. The top layer, V_T, is known as the output layer. In simple prediction problems, the output layer contains a single neuron whose output is the output of the network.

We define T as the number of layers in the network (excluding V_0), or the "depth" of the network. The size of the network is $|V|$, and the "width" of the network is $\max_t |V_t|$. An illustration of a layered feedforward neural network with a depth of 2, size of 10, and width of 5 is shown in Figure (3.3). The neuron in the hidden layer that has no incoming edges will output the constant $\sigma(0)$.

Once a neural network is specified by the parameters (V, E, σ, w), we obtain a function $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$. A set of such functions can serve as a hypothesis class for learning. Typically, a hypothesis class of neural network predictors is defined by fixing the graph (V, E) and the activation function σ, and considering all functions of the form $h_{V,E,\sigma,w}$ for some $w : E \to \mathbb{R}$. The triplet (V, E, σ) is commonly referred to as the *architecture* of the network. We will define several testing architectures in the Chapter 4 and test different set-ups for the mel spectrogram classification task.

We define several common activation functions, that we also going to utilize in our experiments.

(Logistic) Sigmoid:

$$S(x) = \frac{1}{1 + e^{-x}}, \tag{3.13}$$

Rectified Linear Unit:

$$ReLU(x) = \max(0, x), \tag{3.14}$$

Hyperbolic tangent:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3.15}$$

Note, that the family of activation functions is quite big and their utilization differs with the underlying idea and architecture.

3.5.2. Embeddings

The objective of embeddings is to develop a word representation that leverages shared structures across words, which can be employed in neural networks. As word representations form the basis of current natural language processing (NLP), various techniques for constructing word representations are initially examined.

Given the significant advancements in this field over the years, we will commence by providing an overview of common methods for creating word embeddings and exploring their applicability to the task of FNM. We now list several famous embedding techniques, that were also introduced in [Bui22] for a set of NLP problems.

Bag-of-Words. The most fundamental approach to representing words in vector form is the bag-of-words method, which describes word occurrences within a document. In cases where words lack meaning and there is no text corpus in which they appear, it may be tempting to utilize letters as words and complete words as sentences, thereby creating letter representations for existing words. However, the bag-of-words technique encounters significant challenges, including the loss of information about word order and semantics, high dimensionality (known as the curse of dimensionality), and the inability to represent out-of-vocabulary tokens[6].

Distributed Word Representation. To address these challenges, distributed word representations were introduced, representing words (and tokens more generally) as distributed lower-dimensional representations. These representations are trained to capture semantic and syntactic relationships between words, based on the underlying notion that "a word is characterized by the company it keeps," known as the distributional hypothesis [Har54]. The typical method for generating distributed word embeddings is to compute word co-occurrence statistics using unsupervised learning on unlabeled free text.

Language Models. Language models (LMs) are the most popular approach to utilizing the distributional hypothesis for creating distributed word representations and serve as the foundation of modern NLP. LMs were initially popularized by Collobert and Weston in 2008 [CW08], who demonstrated the effectiveness of using neural models to generate semantic word embeddings for downstream tasks. This marked the onset of the contemporary NLP paradigm: pre-training neural models to generate word representations for downstream tasks.

Formally, an LM is a model that, given a sequence of tokens $t_1, ..., t_m$ of length m, outputs a probability distribution over the sequence of tokens, i.e., the likelihood of the tokens occurring jointly:

$$P(t_1, ..., t_m) \overset{\text{(chain rule)}}{=} P(t_1) \cdot P(t_2|t_1) \cdot ... \cdot P(t_m|t_1, ..., t_{m-1}) = \prod_{i=1}^{m} P(t_i|t_{1,...,i-1}). \tag{3.16}$$

The chain rule is applied to calculate the joint probability of tokens in the sequence as conditional probabilities of a token given their respective previous tokens. However, this approach becomes computationally infeasible due to the combinatorial explosion of possible previous tokens. To avoid this problem, the Markov assumption is often utilized, i.e. it is assumed that the probability of $P(t_m|t_1, ..., t_{m-1})$ depends only on the previous $n - 1 \ll m$ tokens:

[6] A tokenizer splits the text into smaller units called tokens. Tokens can be words, characters, or subwords. In our thesis, we primarily use the term "word representations" to facilitate better conceptual understanding. Only when necessary do we explicitly refer to tokens. Nevertheless, all presented approaches can be generalized to any token-level.

$$P(t_m|t_1,...,t_{m-1}) \approx P(t_m|t_{m-(n-1)},...,t_{m-1})$$
$$\rightarrow P(t_1,...,t_m) \approx \prod_{i=1}^{m} P(t_i|t_{i-(n-1),...,i-1}).$$

As the joint probability of tokens now only depend on the product of probabilities of the form $P(t_i|t_{i-(n-1),...,i-1})$ thus allowing us to estimate the probability of tokens (or n-grams), which can be done with the maximum likelihood estimate (MLE) on the training corpus. In practice, models are trained to either predict the subsequent tokens (directional) or to predict the missing token given the surrounding context of the word (bidirectional).

Mikolov et al. [MCCD13] proposed an effective approach to learn high-quality word vectors for millions of words in a vocabulary in an unsupervised manner from a large amount of unlabeled text data. Their word embeddings not only capture semantic and syntactic information, but also learn relationships between words, as illustrated by the equation: Paris - France + Italy = Rome. This approach is called word2vec, and it has two model architectures: *Skip-Gram* (SG) and *Continuous Bag-of-Words* (CBOW).

Both CBOW and SG architectures are based on a simple feedforward neural network. CBOW computes the probability of the current token based on the context tokens within a window W of k neighboring tokens, while SG computes the probability of surrounding tokens within a window W of k neighboring tokens given the current token. The network encodes each token t_i into a center token e_i and a context token c_i, which correspond to the i-th row of the center token matrix $E^{|V|\times d}$ and context token matrix $E^{|V|\times d}$, respectively, where $|V|$ is the size of the vocabulary and d is the token vector embedding size. Given a center token t, SG estimates the likelihood of seeing a context token w conditioned on the given center token with the softmax function (3.8):

$$P(c_w|e_t) = \frac{exp(e_t^T c_w)}{\Sigma_{i=1}^{N} exp(e_t^T c_i)}$$

where e_t denotes the embedding for the center token t and c_w denotes the embedding for the context token w in the window $c_w \in W$. Given a text corpus $t_1,...,t_T$ of length T and assuming that the context words are independently generated given any center word, the model parameters (i.e., token embeddings) are learned by maximizing the likelihood function over the text corpus, which is equivalent to minimizing the negative log-likelihood:

$$\max_{e,c} \prod_{t=1}^{T} \prod_{w\in W_t} P(c_w|e_t) \Leftrightarrow \min_{e,c} -\sum_{t=1}^{T} \sum_{w\in W_t} log(P(c_w|e_t)). \tag{3.17}$$

For a downstream task, the final embedding for token t is either the center token or calculated as the element-wise average or the sum of its center and context representations. Therefore, the word2vec objective in equation (3.17) directly uses the language modeling task to generate effective word embeddings.

FastText. Despite the effectiveness of word2vec in creating powerful word representations, there exist notable limitations. Firstly, the denominator in the softmax equation, which sums over the entire vocabulary, results in slower softmax calculations. Although hierarchical softmax and negative sampling can address this issue, word2vec representations suffer from two major conceptual drawbacks: the inability to embed any tokens outside the vocabulary and the failure to account for the linguistic morphology of a word.

Specifically, word2vec learns separate representations (without parameter sharing) for words with different morphological forms, such as "eat" and "eaten," even though they share similar contexts.

To address these challenges, Bojanowski et al. [BGJM16] introduced FastText, a novel embedding approach that extends the idea of word2vec. FastText exploits the internal structure of words to improve word representations. Instead of constructing representations for individual words, FastText learns representations for n-grams of characters and subsequently creates word representations by summing the character n-grams. For example, with n=3, the word "*artificial*" is represented by the sequence *<ar, art, rti, tif, ifi, fic, ici, ial, al>* (where the angular brackets indicate the beginning and end of the word), which allows for more accurate representations of prefixes and suffixes. Additionally, FastText can represent words that are absent from the training data by breaking them down into n-grams.

FastText and word2vec have gained popularity due to their ease of use and effectiveness in various natural language processing (NLP) problems [LBS$^+$16] [KZS$^+$15] [KSKW15]. Moreover, these word representations could be utilized to also construct character representations, that we're actually looking for. Despite these advantages, both methods have several drawbacks, such as the inability to capture phrases and polysemy of words accurately, and their dependence on a fixed-size window of context words for language model conditioning. A more natural way to learn representations is to allow for variable context word quantities during training.

This drawbacks however don't pose a problem for us, if we want co use them for character embedding. One possible idea would be to use the single characters in a string as "words" and the strings themselves as "sentences", projecting the outlined ideas to our case. We then might train the models on the given dictionary of names to predict the likelihood of some setted up string of characters according to the model. We implement this approach programmatically and show that, this idea doesn't work as planned. Indeed, this idea only works under assumption, that the letters are always to find near other letter, that correspond to them. This assumption however fails, since it's impossible to prove, that such connections exist.

3.5.3. Recurrent Neural Network

A recurrent neural network (RNN) is an extension of a conventional feedforward neural network, which is able to handle a variable-length sequence input. We refer to each input index in the input sequence as time. The RNN handles the variable-length sequence by having a recurrent hidden state whose activation at each time is dependent on that of the previous time.

More formally, given a sequence $x = (x_1, x_2, ..., x_m) \in \mathbb{R}^{n \times m}$, the RNN updates its recurrent hidden state h_t by

$$h_t = \begin{cases} 0, & t = 0 \\ \phi(h_{t-1}, x_t), & \text{otherwise} \end{cases} \tag{3.18}$$

where ϕ is a nonlinear function such as composition of a logistic sigmoid with an affine transformation. Optionally, the RNN may have an output $y = (y_1, y_2, ..., y_m) \in \mathbb{R}^{d \times m}$ which may again be of variable length. For our analysis we assume that the input and output sequences are of same lengths. Traditionally, the update of the recurrent hidden state in Equation (3.18) is implemented as

$$h_t = g(Ux_t + Wh_{t-1}) \in \mathbb{R}^h, \tag{3.19}$$

where g is a smooth, bounded function such as a *logistic sigmoid function* (3.13) or a *hyperbolic tangent function* (3.15), and W and U are weight matrices. A generative RNN outputs a probability distribution over the next element of the sequence, given its current state h_t, and this generative model can capture a distribution over sequences of variable length by using a special output symbol to represent the end of the sequence. The sequence probability can be decomposed using distributional hypothesis that is employed by the Language Models (3.16), where the last element is a special end-of-sequence value. Generally speaking the Language Models are usually based on RNN and use the same underlying architecture. We model each conditional probability distribution with

$$P(x_t | x_1, ..., x_{t-1}) = g(h_t),$$

where h_t is from Equation (3.18). Unfortunately, it has been observed by, e.g., Bengio et al. [BSF94] that it is difficult to train RNNs to capture long-term dependencies because the gradients tend to either vanish (most of the time) or explode (rarely, but with severe effects). This is also known as **Vanishing gradient problem**. To understand the nature of it, we first have to understand, where does it come from. Let's first observe, how do forward and backward passes of the RNN look like. To keep things simple, we consider an RNN without bias parameters.

For the forward pass we compute the output with regard of the input in the following way: first, we compute the hidden state at time t as a linear combination of the current input and the previous hidden state wrapped into an activation function as in (3.19). Afterwards we compute the predicted output $\hat{y}_t$ by combining the hidden state in the same manner, e.g. $\hat{y}_t = \sigma(Vh_t)$. Note, that for the linear transformation we use the same weight matrices $U \in \mathbb{R}^{h \times n}, W \in \mathbb{R}^{h \times h}, V \in \mathbb{R}^{d \times h}$ at all time steps as shown in Figure 3.4. The optimal values of these matrices are the very target of the training process of the RNN.

In order to update U, W, V at each learning epoch, one have to calculate their gradient in order to minimize given loss function. For it we have to make a backward pass going from the last step to the first backwards,

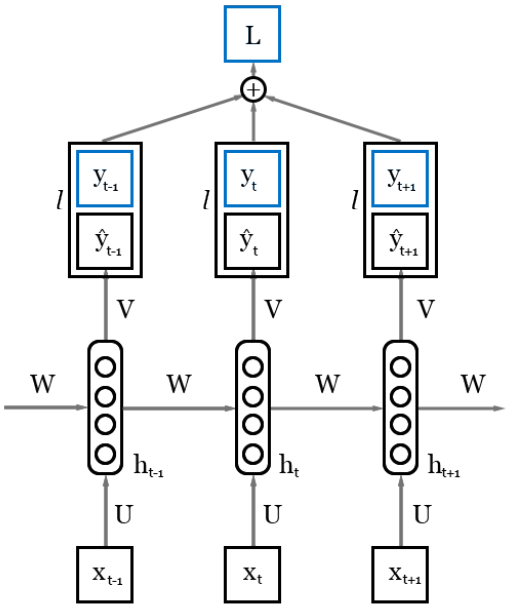


Figure 3.4.: Forward pass of the RNN illustrated on three time-steps.

according to the array directions of the Figure 3.4. We start with calculating the loss of the prediction. Let's assume the overall loss function to be

$$L := \frac{1}{m}\sum_{t=1}^{m} l(\hat{y}_t, y_t),$$

where $l(\hat{y}_., y_.)$ is a loss function at each time step. Then the gradient of L with regard to $\hat{y}_t$ is

$$\frac{\delta L}{\delta \hat{y}_t} = \frac{\delta l(\hat{y}_t, y_t)}{\delta \hat{y}_t m} \in \mathbb{R}^m.$$

Knowing this and $\frac{\delta \hat{y}_t}{\delta V} = h_t \sigma_V'(V h_t)$ we can compute $\frac{\delta L}{\delta V}$ as follows:

$$\frac{\delta L}{\delta V} = \sum_{t=1}^{m} < \frac{\delta L}{\delta \hat{y}_t}, \frac{\delta \hat{y}_t}{\delta V} > \; = \sum_{t=1}^{m} \frac{\delta l(\hat{y}_t, y_t)}{\delta \hat{y}_t m} (h_t \sigma_V'(V h_t))^\top$$

Here we use the $< \cdot, \cdot >$ operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication. For higher dimensional tensors, we use the appropriate counterpart. The operator $< \cdot, \cdot >$ hides all the notation overhead.

Going backwards down the line, we can also compute W and U as follows:

$$\frac{\delta L}{\delta W} = \sum_{t=1}^{m} < \frac{\delta L}{\delta h_t}, \frac{\delta h_t}{\delta W} >= \sum_{t=1}^{m} \frac{\delta L}{\delta h_t} (h_{t-1} g_W'(U x_t + W h_{t-1}))^\top,$$

$$\frac{\delta L}{\delta U} = \sum_{t=1}^{m} < \frac{\delta L}{\delta h_t}, \frac{\delta h_t}{\delta U} >= \sum_{t=1}^{m} \frac{\delta L}{\delta h_t}(x_t g'_U(Ux_t + Wh_{t-1}))^\top.$$

Finally, we have to compute $\frac{\delta L}{\delta h_t}$. Observe that at the final time step m, the objective function L depends on the hidden state h_m only via $\hat{y}_m$. Therefore, we can easily find the gradient $\frac{\delta L}{\delta h_m}$ using the chain rule:

$$\frac{\delta L}{\delta h_m} =< \frac{\delta L}{\delta \hat{y}_m}, \frac{\delta \hat{y}_m}{\delta h_m} >= (V\sigma'_{h_m}(Vh_m))^\top \frac{\delta L}{\delta \hat{y}_m}.$$

It gets trickier for any time step $t < m$, where the objective function L depends on h_t via h_{t+1} and $\hat{y}_t$. According to the chain rule, the gradient of the hidden state $\frac{\delta L}{\delta h_t} \in \mathbb{R}^h$ at any time step $t < m$ can be recurrently computed as:

$$\frac{\delta L}{\delta h_t} =< \frac{\delta L}{\delta h_{t+1}}, \frac{\delta h_{t+1}}{\delta h_t} > + < \frac{\delta L}{\delta \hat{y}_t}, \frac{\delta \hat{y}_t}{\delta h_t} >= (Wg'_{h_{t+1}})^\top \frac{\delta L}{\delta h_{t+1}} + (V\sigma'_{h_t})^\top \frac{\delta L}{\delta \hat{y}_t}.$$

Now, as we can see it depends once again on $\frac{\delta L}{\delta h_{t+1}}$, which brings us to the following expansion for any time step $1 < t < m$:

$$\frac{\delta L}{\delta h_t} = \sum_{i=t}^{m} ((Wg'_{h_{i+1}})^\top)^{m-i}(V\sigma'_{h_i})^\top \frac{\delta L}{\delta \hat{y}_{m+t-i}}.$$

We now can observe, where does the vanishing gradient problem come from, since computing $\frac{\delta L}{\delta h_t}$ involves potentially very large powers of W. In it, eigenvalues smaller than 1 vanish and eigenvalues larger than 1 diverge. This is numerically unstable, which manifests itself in the form of vanishing and exploding gradients (rarely), which then makes the model insensitive to past inputs. One way to address this is issue is to clip the gradient to a smaller number whenever they explode [PMB12]. Other way is to initialize W as the identity matrix to avoid the vanishing gradients [LJH15] and use the Rectified Linear Units (ReLU) instead of the sigmoid function [Aga18]. In practice, this truncation can also be effected by detaching the gradient after a given number of time steps.

This architecture looks very tempting to include into our solution, since the inputs might have different forms and length. Every name, as well as any written word, is a sequence of characters, where each one is dependent on the previous character. However those dependencies might be "forgotten" for the long sequences. To get the best performance, we will use a more sophisticated sequence model called long short-term memory (LSTM) model [HS97][GSC00], which introduces is a sub-architecture for the hidden layer of an RNN.

3.5.4. Long Short-Term Memory Model (LSTM)

In the context of neural networks, Long Short-Term Memory (LSTM) models are analogous to standard recurrent neural networks, with the difference being that each *recurrent node* in the latter is replaced by a *memory cell* in the former. Memory cells in LSTM models contain an *internal state*, which is represented by a node that possesses a self-connected recurrent edge with a fixed weight of 1. This mechanism ensures that gradients can flow seamlessly across multiple time steps without encountering vanishing or exploding gradient problems.

The term "long short-term memory" stems from the intuition that simple recurrent neural networks possess *long-term memory* in the form of slowly changing weights that encode general information about the data, as well as *short-term memory* in the form of transient activations that pass from one node to the next. LSTM models introduce a new form of memory storage through the use of memory cells, which are composite units constructed from simpler nodes following a specific connectivity pattern that includes multiplicative nodes.

We will introduce, inspired by Zhang at. al.[ZLLS21], the idea of the memory cell.

Gated memory cell: Each memory cell is equipped with an *internal state* and a number of multiplicative gates that determine whether (i) a given input should impact the internal state (*the input gate*), (ii) the internal state should be flushed to (*the forget gate*), and (iii) the internal state of a given neuron should be allowed to impact the cell's output (*the output gate*).

This means that we have dedicated mechanisms for when a hidden state should be *updated* and also when it should be *reset*. These mechanisms are learned and they address the concerns listed above. For instance, if the first token is of great importance we will learn not to update the hidden state after the first observation. Likewise, we will learn to skip irrelevant temporary observations. Last, we will learn to reset the latent state whenever needed. We discuss this in detail below.

The data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Figure 3.5. Three fully connected layers with sigmoid activation functions compute the values of the input, forget, and output gates. As a result of the sigmoid activation, all values of the three gates are in the range of . Additionally, we require an input node, typically computed with a tanh activation function. Intuitively, the input gate determines how much of the input node's value should be added to the current memory cell internal state. The forget gate determines whether to keep the current value of the memory or flush it. And the output gate determines whether the memory cell should influence the output at the current time step.

We assume the previous input-output set-up from previous section. Then we define the gates at time step t as follows: the input gate is $I_t \in \mathbb{R}^h$, the forget gate is $F_t \in \mathbb{R}^h$ and the output gate is $O_t \in \mathbb{R}^h$:

$$
\begin{aligned}
I_t &= \sigma(x_t W_{xi} + h_{t-1} W_{hi}),\\
F_t &= \sigma(x_t W_{xf} + h_{t-1} W_{hf}),\\
O_t &= \sigma(x_t W_{xo} + h_{t-1} W_{ho}),
\end{aligned}
$$

where $W_{xi}, W_{xf}, W_{xo} \in \mathbb{R}^{n \times h}$ and $W_{hi}, W_{hf}, W_{ho} \in \mathbb{R}^{h \times h}$ are weights. Note, that as previously we aren't using any biases for notation simplicity. We use logistic sigmoid function (as introduced in previous section) to map the input values to the interval $(0, 1)$.

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the input node $\tilde{C} \in \mathbb{R}^h$. Its computation is similar to that of the three gates described above, but using a $tanh$ as activation function with a value range $(-1, 1)$. This leads to the following equation:

$$\tilde{C}_t = tanh(x_t W_{xc} + h_{t-1}).$$

In LSTMs, the input gate I_t governs, how much we take new data into account via $\tilde{C}_t$. And the forget gate F_t addresses. how much of the old cell internal state C_{t-1} we retain. Using the Hadamard (elementwise) $\odot$ product operator we arrive at the following update equation:

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t.$$

If the forget gate is always 1 and the input gate is always 0, the memory cell internal state C_{t-1} will remain constant forever, passing unchanged to each subsequent time step. However, input gates and forget gates give the model the flexibility to learn when to keep this value unchanged and when to perturb it in response to subsequent inputs. In practice, this design alleviates the vanishing gradient problem, resulting in models that are much easier to train, especially when facing datasets with long sequence lengths.

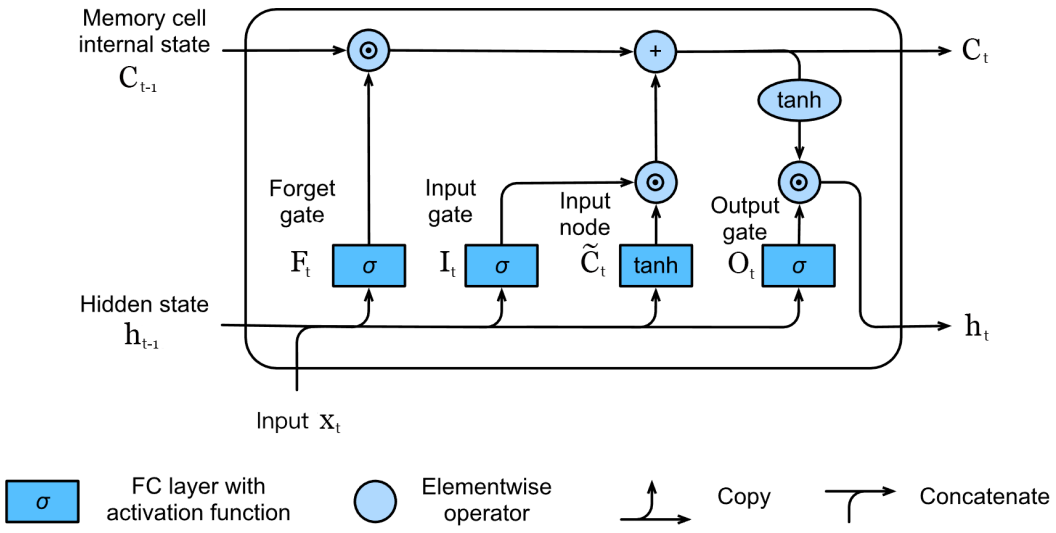


Figure 3.5.: The hidden state of a LSTM model.

3.5.5. Convolutional Neural Network

Now that we have learned how to handle variable-length input data in a recurrent manner, we direct our attention to Convolutional Neural Networks (CNNs) [LJB+95]. Traditionally, CNNs have been developed and primarily used in the field of computer vision [BB82], but nowadays practitioners apply them whenever possible. Increasingly, CNNs have emerged as credible competitors even for tasks involving one-dimensional sequences, such as audio [AHMJ+14], text [KGB14], and time series analysis [LJB+95], where recurrent neural networks are conventionally employed. Moreover, clever adaptations of CNNs have also facilitated their utilization for graph-structured data [KW16] and recommender systems.

Image data is represented as a two-dimensional grid of pixels, whether it is monochromatic or in color. Each pixel corresponds to one or multiple numerical values, respectively. One approach to handling such data is to flatten the images and utilize them as input vectors in a feedforward neural network structure. In this case, the spatial relation between pixels is disregarded. Since these networks are invariant to the order of features, similar results can be obtained regardless of whether the order corresponding to the spatial structure of pixels is preserved or if the columns of the design matrix are permuted before fitting the MLP's parameters. However, it is preferable to leverage our prior knowledge that nearby pixels are typically related to each other in order to build efficient models for learning from image data. CNNs, on the other hand, are a powerful family of neural networks explicitly designed for this purpose.

Let us develop an intuition for how convolutional networks are designed for image recognition. First, it is important to understand that we do not need to examine the entire picture to capture similarities or patterns. Instead, the idea is to focus our attention on distinct areas of the image, processing them one by one and comparing them to the desired output. To achieve this, our network should respond similarly to the same patch in the earliest layers, regardless of where it appears in the image. This principle is known as *translation invariance* (or translation equivariance). Furthermore, since we are not concerned with other regions of the image, we can disregard them during our calculations. Therefore, the earliest layers of the network should focus on local regions without considering the contents of distant regions. This is known as the *locality principle*. Eventually, these local representations can be aggregated to make predictions at the whole image level. As we progress through deeper layers, the network should be capable of capturing longer-range features of the image, analogous to higher-level vision in nature.

The concept of translation invariance [Zo88] implies that a shift in the input X should simply lead to a shift in the hidden representation H. This is only possible if the window through which we examine the image is position-independent and remains constant. We define this window as F and refer to it as a *filter*. Additionally, the earlier mentioned shift s should remain constant throughout the entire process. Therefore we end up with following representation for our hidden state H at each position (i, j) on the image:

$$H_{(i,j)} = s + \sum_a \sum_b F_{(a,b)} X_{(i-a,i-b)}, \tag{3.20}$$

where a, b are the offsets from i and j. Observe, that our filter F is working not only as a window, that we're looking through at the image, but also as a kernel, that is able to emphasize desired regions (e.g. the edges) and neglect others. This operation is denoted as *convolution*. Mathematically convolution between two functions $f, g : \mathbb{R}^d \to \mathbb{R}$ defined as follows [Rud73]:

$$(f * g)(x) = \int f(z) g(x - z) dz.$$

It means that we measure the overlap between f and g when one function is "flipped" and shifted by x. Whenever we have discrete objects, we are using the integral as a Riemann sum. For instance, for vectors from the set of square summable infinite dimensional vectors with index running over $\mathbb{Z}$ we obtain the following definition:

$$(f * g)(i) = \sum_a f(a)g(i-a).$$

For two-dimensional tensors, we have a corresponding sum with indices (a, b) for f and $(i-a, i-b)$ for g respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i-a, i-b).$$

This mathematical concept is the main inspiration for the convolutional neural network. We observe, that the filter F in (3.20) isn't given directly, rather we going to learn its weights during training.

3.6. Text-To-Speech conversion using Tacotron2

Text-to-speech (TTS) is a complex problem that involves several stages, such as linguistic feature extraction, duration modeling, acoustic feature prediction, and signal processing. These stages require extensive domain expertise and are trained independently, which can lead to compounding errors. In contrast, end-to-end TTS systems can be trained on text-audio pairs with minimal human annotation, thus alleviating the need for laborious feature engineering. Furthermore, end-to-end TTS systems can handle rich conditioning on various attributes and can be more robust to errors.

Thus, such a system would be a great choice to achieve our goal - representation of a name in a manner, that preserves pronunciation and, in the same time, converts our text input to a numerical one. For this task we are going to use Tacotron2 model [SPW+17].

The backbone of Tacotron2 is a seq2seq model with attention [BCB16]. Figure 3.6 depicts the model, which includes an encoder, an attention-based decoder, and a vocoder part. At a high-level, Tacotron2 takes characters as input and produces spectrogram frames, which are then converted to waveforms. We describe these components below.

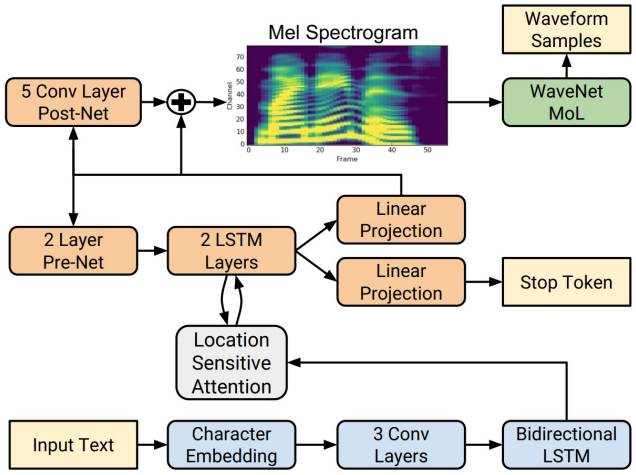


Figure 3.6.: Block diagram of the Tacotron 2 system architecture

First, we describe the encoder part. Then the input sequence will be $x = (c_1, c_2, ..., c_n)$, where c_i is a single character in a sequence for $i \in 1, 2, ..., n$. Afterwards the character sequence gets encoded into character embedding. Tacotron2 uses a learned 512-dimensional character embedding. Next the sequence gets concatenated into a matrix and used as an input for 3 consecutive convolution layers. Each layer contains 512 filters of size 5×1. This process model longer-term context (e.g., N-gram model) [JM08] in the input character sequence. The output of the final convolution layer is passed into a single bi-directional LSTM layer containing 512 hidden states (256 in each direction) to generate the encoded features. This process helps our model to catch short and long term context for every feature.

The encoder output is consumed by an attention network which summarizes the full encoded sequence as a fixed-length context vector for each decoder output step. Wang at. al [WSS+17a] use the location-sensitive attention from [CBS+15] to use cumulative attention weights from previous decoder time steps as an additional feature. This encourages the model to move forward consistently through the input, mitigating potential failure modes where some subsequences are repeated or ignored by the decoder.

Attention probabilities are computed after projecting inputs and location features to 128-dimensional hidden representations. Location features are computed using 32 1-D convolution filters of length 31.

The decoder is an autoregressive recurrent neural network which predicts a mel spectrogram from the encoded input sequence one frame at a time. The prediction from the previous time step is first passed through a small pre-net containing 2 fully connected layers of 256 hidden ReLU units. During testing, Wang at. al 2017 found that the pre-net acting as an information bottleneck was essential for learning attention. The pre-net output and attention context vector are concatenated and passed through a stack of 2 uni-directional LSTM layers with 1024 units. The concatenation of the LSTM output and the attention context vector is projected through a linear transform to predict the target spectrogram frame. Finally, the predicted mel spectrogram is passed through a 5-layer convolutional post-net which predicts a residual to add to the prediction to improve the overall reconstruction. Each post-net layer is comprised of 512 filters with shape 5×1 with batch normalization, followed by $tanh$ activations on all but the final layer.

We observe, that the architecture of the tacotron2 follows the desired procedure to create mel spectrograms, but backwards. Whereas we would gladly take a dataset of waveform audio-signal of spoken names for the given dictionary to transform them into mel spectrograms using STFT as described in Section 3.4, the introduced model does the opposite. It takes the written name as the input, predicts a mel spectrogram and transforms it into waveform using Griffin-Lim algorithm [GL84].

While Tacotron2 has different goals than are ours, namely achieving human-like speech mechanism, it also produces high quality mel spectrograms from the input text. Thus, it's the perfect tool for us to work with. It also produces slightly different spectrograms each times due to the dynamical implementation of the "stop token", which also contribute to our idea of the robust prediction models.

3.7. Conclusion

In this Chapter we formalized our problem and described several ideas on how to approach the desired solution. We formulated our main goal and showed, that it's possible to achieve it, having enough data. We then introduced central ideas, that will help us create feasible neural network architectures for the testing and outlined a modern state-of-art neural network called tacotron2, that will achieve the task of mel spectrogram generation. We will utilize all of this tools in the coming Chapter and build several architectures for classification for the purpose of fuzzy matching.

4. Experiments

As we've outlined in Chapter 2, there're effective methods for *Fuzzy Name Matching*, when comparing similar names one by one. Thus, we will focus on the task of the search reduction. In this Chapter, we will show experimentally, that the neural network approach to this task reduces the search time dramatically. We will also show, that this solution is scalable and can be applied for the preprocessing of CRM data. We start with the solution proposal in the Section 4.1. We then set the classification task and outline the main preparations for the training in Section 4.2. Consequently we show several proposed architectures and their performance during the training. Lastly, we analyse their performance on the validation data in Section 4.3. We conclude our work and discuss further development in Section 4.4.

4.1. Solution proposal

To reduce the search time using classical fuzzy matching algorithms, we transform the available CRM dictionary (RHS) in its respective set of mel spectrograms. Note, that although it's (somewhat) computationally complex task to generate mel spectrograms out of a large set of names, we only have to do this process once.

Assuming the RHS is always given, we can 'memorize' the given data in a way that enables us to classify each unvalidated name as one of the 'memorized' ones. To achieve this, we propose creating a CNN based neural network that can classify the given mel spectrogram if trained properly.

To proceed with this task, we first need to define the requirements for our solution architecture:

1. The neural network architecture should enable us to train weights for name classification using m labels.
2. The classification should be robust to spelling errors, where similar spelling should lead to similar classes.
3. The classification should also robust for pronunciation.
4. The correct predictions should have ether highest or have a high probability (according to Softmax normalization of the last layer).

To achieve the best performance it's not enough to encode only the given names from RHS as mel spectrograms. Cook 1999 [Coo99] asserts that "unlike native speakers, foreigners may not know the actual system of English, and will appear to use the wrong letter". The study concludes that English spelling errors can be classified into four main types of spelling errors: **omission**, **substitution**,**transposition** and **insertion** errors, which confirms with the ideas presented in Chapter 2 for Edit Distances. To preserve pronunciation and spelling robustness, we propose the data augmentation techniques, before we transform

them into their mel spectrogram representations. Thus, we train the neural network for spelling robustness using adversarial training. Finally, we create several network architectures to achieve the best results for the proposed problem.

4.2. Mel spectrogram classification

In Section 3.6 we've seen that it's possible to achieve an audio representation of any given text input. We use this tool to convert our RHS dictionary into a set of mel spectrograms and train the classification model on this set. In order to create such a model we will use a similar approach as [KSH12b]. It's architecture has proven it's efficiency in the image recognition over a large-scale image recognition tasks. It was trained upon ImageNet Dataset with over 22.000 labels.

We will split the whole training process into two steps: first, we're going to use some subset of the RHS dictionary in order to prove the concept; second, once we have a good idea of what kind of architecture is necessary, we scale up the models and increase the number of labels, that we want to classify.

Observe, that since the main goal of this theses is to create a solution, that should overcome the time-complexity problem as described in Section 2.5, we're not necessary want to build a classifier, that would classify arbitrary big number of labels perfectly. It's enough to decrease the number of labels of the validated dataset that we have to iterate over to a significantly smaller amount.

4.2.1. Models and datasets

Training Set for tacotron2.

We use the model pre-trained on an internal US English dataset [WSS$^+$17b], which contains 24.6 hours of speech from a single professional female speaker. All text in our datasets is spelled out. e.g., "16" is written as "sixteen", i.e., our model is trained on normalized text. Note, that since the training was performed by an English speaker, it might affect the mel spectrogram representation of the names that natively come from other languages. The affect comes in a sense of capturing only those misspellings, that an English speaker might do, which sometimes differ from those of a non-native English speaker.

Performance metrics.

As in the [KSH12b] we will use a mix of convolution and dense layers for classification task. During our experimental testing of the performance, we're interested in following metrics: number of required training epochs, validation and training accuracy, as well as a number of correct classifications, that fall below some threshold $t \in \mathbb{N}$ of labels on the validation dataset as well as the number of misclassified labels (beyond the threshold). This can be seen as a variation of the approximate dictionary searching from Definition 2.3.2. While exploring the performance of mixed convolution and feedforward model, we will try different activations, normalizations, dimensionality reduction and regularization techniques.

Datasets.

As mentioned earlier, we train our models on two dictionaries of names separately. In order to train the model appropriately, we transform both dictionaries into the mel spectrogram datasets. First dataset RHS_1 will contain just 10^3 names, second one RHS_2 will be the full dataset of 10^4 names, where $RHS_1 \subseteq RHS_2$. We call this phase *Spectrogram generation*. In order to increase model robustness, we use Data Augmentation[1] during the spectrogram generation phase. Since we are looking for a robust way to represent not only character sequence, but also audio similarities, it won't be enough to address this issue by distorting

[1] a technique to artificially increase the training dataset and reduce overfitting by making a small changes to the original data [YXZ$^+$22]

spectrograms in the way, one would do for images. Rather we distort the name strings by generating misspellings and mistakes directly. This would affect the output spectrogram representations by completely changing some parts of it in comparison to original name, but preserving the common parts. Thus, it allows us to create more robust predictions.

In order to achieve this, we define several functions in Appendix B that simulate human-like mistakes, such as a misclick on a keyboard, permutations of letters, missing or additional letter and spacings between letters (e.g. "Abuhalil" -> "Abu Halil"). Additional augmentation is also provided by Tacotron2 generation process, generating slightly different mel spectrograms every time.

We generate 100 different mel spectrograms for each last name in the RHS_1 dataset to preserve the balance between labels. To prove the concept, we use the System's RAM to train our models. It will allow us to increase the training speed and try several neural network architectures. In order to achieve this, we bound our training set to have at max. 200.000 spectrograms. Once we confirm, that the predictions produce the desirable result, we increase the number of labels and lower the number of augmentations per name to 20 per name in the RHS_2 dataset. This way we keep the training dataset inside the established boundary. We will observe how this affect the quality of the predictions and show, that this approach is scalable even for more than 10.000 labels.

Note, since we use mel spectrogram name representations for classification task, the model doesn't require any heavy hardware support as the image classification does. The reason behind it is the number of channels, that is used to represent the spectrogram. An ordinary image consists of three channels, one for each color, whereas the spectrogram is a voice representation and consists of only one layer.

As mentioned in the Section 2.5, we assign each transformed name to the according label using label encoding [PVG+11]. During the training phase we split each RHS into two parts: *train* and *test* datasets. The training of the models will happen over the train dataset, whereas the test dataset will produce the insides, on how well the model generalizes on the new data. For the sake of consistency, we take 30 of 100 mel spectrograms from each label into the test dataset of RHS_1 and 6 of 20 mel spectrograms from each label of RHS_2. Hence, we split our data into having 70% of the RHS to serve as train and 30% to serve as test data. This way we will preserve, that each name is represented equally in both datasets.

Finally, we transform the LHS, defined in Section 2.5 into mel spectrogram dataset. This data will serve as the *validation* data. Note, that LHS was defined the same way as the RHS_1 and having exactly the same names inside. The only difference, in that the LHS consists additionaly of 4 misspelled names for each name in RHS_1 making it four times bigger. Although this mistakes were mostly created by LLM [Ope23], every misspelling was additionally supervised and proved to be unique. The exact implementation can be found in the code, outlined in Appendix B.4.

We then resize all transformed mel spectrograms for LHS and RHS into the same shape using bilinear interpolation. Empirically observed, that for our task it's enough to have resize each mel spectrogram to the 80×100 shape, since for most name representation it's just slightly smaller than the original ones. It is, of course, not the unique representation shape and can be reduced further for the space optimization purposes.

Software and hardware.

For the implementation we use Python programming language (v3.9) combined with the programming interface of Jupyther Notebook by Anaconda, Inc. Main framework for modeling is the open-source library pyTorch (1.13.1 + cu117) with GPU (Cuda) acceleration as well as other common libraries for data processing and plotting, e.g. Sklearn, Pandas, Numpy, Librosa and Plotly. The necessary libraries will be

listed inside of the requirements document and in the Appendix B.1. The hardware is represented by a single AMD Ryzen 7 3700X 8-Core Processor (3.6 GHz) with 16 GB RAM memory and a single NVIDIA GeForce RTX 3060 ti (8 GM VRAM) CUDA GPU unit.

4.2.2. Experimental Set-up

We train 8 models in total to find the optimal setup for mel spectrogram classification. Although we're using the same kind of architecture as one would use for image classification, the mel spectrograms still differ from ordinary images. The pattern recognition should rely on the signal differences only. Meaning, that there're no "direct" forms to recognize as one would do during regular image recognition task. That also means, that we're rather interested in the signal differences as smooth transition "edges" on the spectrogram. The spectrogram signal is time dependent, which also gives us a room to work with, i.e. using LSTM layer at some point to represent time dependency.

General hyperparameters for classification

Each model will be trained over 100 epochs with an early stopping mechanism. The training will be stopped, once there's no significant progress in validation loss observed or the training achieves 99.9% accuracy. We set $\epsilon = 0.01$ to stop the training, if the training accuracy achieves $1 - \epsilon$. We also save the best model for each of the following criteria: best validation loss, best validation accuracy and the last epoch model. Since we create multi-label classification model, we're going to use cross entropy loss as our general loss function. For the optimization process we rely on AdamW algorithm [LH17] with the learning rate $l = 0.001$ for RHS_1 and $l = 0.0001$ for RHS_2. During mel spectrogram generation process we create equal number of samples for each name, which allows us to create balanced training dataset. Also we introduce the training regulation mechanism, that should reduce our learning rate by factor $\alpha = 0.1$, if we get on plateau for more than k epochs. We summarize all the settings in the Table 4.1.

Setting	Value	
	RHS_1	RHS_2
Max. Epoch	100	100
Early stopping	99.9% accuracy	99.9% accuracy
Learning rate	0.001	0.0001
Reduce-on-plateau	$\alpha = 0.1$ ($k = 3$)	$\alpha = 0.1$ ($k = 3$)
Weights init	random	Xavier/He
Batch size	32	32
Optimizer	AdamW	AdamW

Table 4.1.: Training settings

4.2.3. Training acceleration techniques

In order to accelerate the training process of a neural network, we outline several additional techniques, that we used in our experiments. Most of them were employed by manually testing each of these techniques, since there's no general solution on how to use them in general. Rather, there exist some thumb rules, that

help one to improve the training process. Still, they should be approached individually and depend mostly on the exact problem and physical limitations of the hardware.

Batch size

A batch (or minibatch) refers to a subset of the training data that is processed together during each iteration of the training process. Instead of updating the neural network's weights based on individual training examples, the weights are updated based on the average gradient computed from a batch of examples.

Training neural networks using batches offers several advantages:

- *Efficiency*: Processing a large dataset one example at a time is computationally expensive. By processing examples in batches, parallelization is utilized, taking advantage of the GPU and speeding up the training process.
- *Smooth Gradient Estimates*: The gradient of the loss function with respect to the weights is estimated by computing the average (stochastic) gradient over the samples in the batch. This averaging helps to smooth out the noisy gradients that may arise from individual examples, leading to more stable and accurate updates to the model's weights. Thus, the size of the batch contributes to avoiding suboptimal local minima if small enough and to not avoiding the global minimum if large enough. This information is rarely directly available. In our theses we rely on the experiments and memory usage limitations, to assign the batch size.
- *Memory Usage*: Neural networks often require substantial memory to store the activations and gradients of the intermediate layers. By using smaller batches, memory requirements can be reduced, allowing for larger models or datasets to be processed within the available resources.

For all our models, we set batch size to be equal to $N = 32$. Experimentally, it's almost the largest batch size allowed by the underlying hardware, that we can utilize.

Batch normalization (BatchNorm)

At a high level, BatchNorm is a technique that aims to improve the training of neural networks by stabilizing the distributions of layer inputs. This is achieved by introducing additional network layers that control the first two moments (mean and variance) of these distributions.

For a layer with d-dimensional input $x = (x^{(1)}, \ldots, x^{(d)})$, we will normalize each dimension $x^{(k)}$ as

$$x_b^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

where the expectation and variance are computed over the training dataset. As shown in [LBBH98], such normalization speeds up convergence, even when the features are not decorrelated.

Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, Sergey Ioffe and Christian Szegedy [IS15] make sure that the transformation inserted in the network can *represent the identity transform*. They introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}$ and $\beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

These parameters are learned along with the original model parameters and restore the representation power of the network. Indeed, by setting $\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathbb{E}[x^{(k)}]$, we could recover the original activations if that was the optimal thing to do.

The BatchNorm's success, as well as its original motivation, relates to so-called *internal covariate shift* (ICS)([Shi00]). Informally, ICS refers to the change in the distribution of layer inputs caused by updates to the preceding layers. It is conjectured that such continual change negatively impacts training. The goal of BatchNorm is to reduce ICS and thus remedy this effect [STIM19].

Indeed, during our experimental phase, BatchNorm showed quite an improvement of the training process and allowed us to improve the training process.

MaxPooling

Max Pooling is a popular technique used in CNNs to downsample feature maps. It works by dividing the input image into a set of non-overlapping rectangular windows and computing the maximum value for each window. This reduces the spatial dimensions of the feature map while retaining the most important features. Max Pooling is commonly used after convolutional layers to extract robust features from the input image [HZRS15a].

In our case, we're going to use mel spectrograms that represent 80 frequencies over 100 time-frames. One convolutional layer with 100 filters expresses $100 * 100 * 80 = 800.000$ parameters. Since we want to use them for classification, at some point we're going to project them into a fully connected layer by flattening. We observe that already at this (small) scale this task will require some major computational power. Instead, we reduce dimensionality by extracting the most important features using MaxPooling between the convolutional layers.

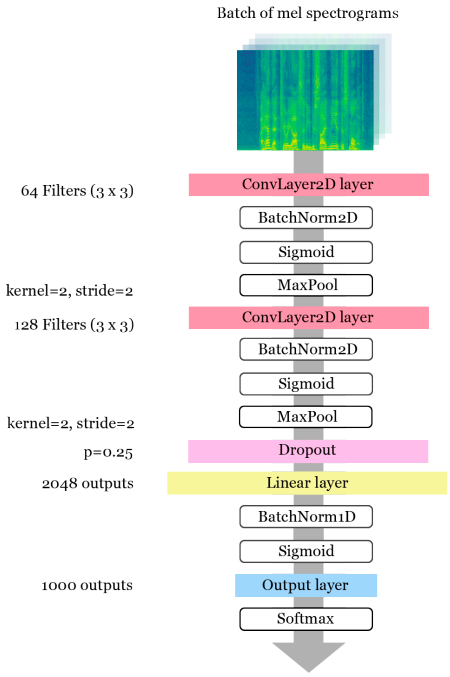
Weights Initialization

Weight initialization plays a crucial role in training neural networks effectively. Proper initialization helps in achieving faster convergence, avoiding vanishing or exploding gradients, and promoting stable and efficient learning.

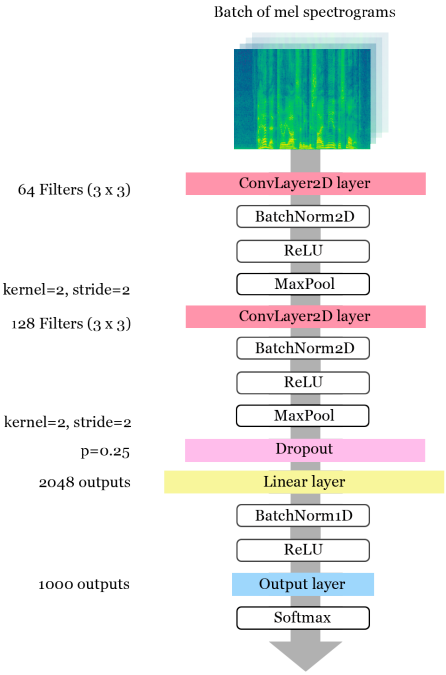
Xavier Glorot and Yoshua Bengio [GB10] explore the challenges faced in training deep neural networks and proposes the *Xavier initialization method*. They find that the logistic sigmoid activation is unsuited for deep networks with random initialization because of its mean value, which can drive especially the top hidden layer into saturation.

Kaiming He, et al [HZRS15b] introduces the *He initialization method*, which is designed specifically for rectified linear units (ReLU) activations. It demonstrates the benefits of using He initialization for improving the training of deep convolutional neural networks.

We will especially use both of this methods to improve the training process when increasing the number of labels and decreasing the number of samples per label. We discovered experimentally, that without weights initialization, e.g. the architectures with ReLU activations often land directly on plateau and show no decrease in the loss function or show worse training characteristics than otherwise.



(a) SCNN var.1 with Sigmoid activations



(b) SCNN var.1 with ReLU activations

Figure 4.1.: SCNN architectures for RHS_1

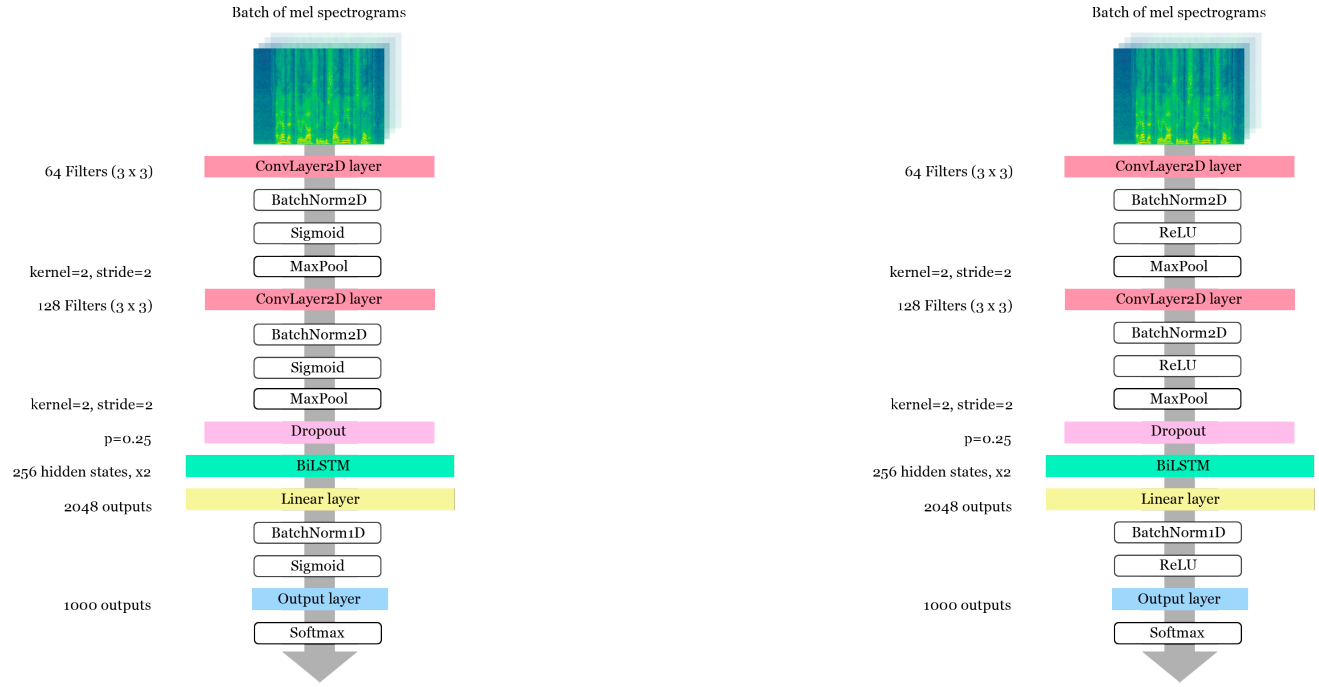
4.2.4. Model architectures and training process

In this Section we declare the possible classification architectures and observe how well they perform during the training. We then analyse the performance of each architecture on the validation dataset.

Since we already recognized general similarities between an image and a mel spectrogram classification, we will rely on the structures, that are similar to the one proposed in [KSH12b]. The general idea is to use several convolution layers, which output we flatten and then feed to a regular feedforward network. The number of the hidden layers as well as the number of parameters (neurons), as outlined in the Table 4.2, is variable and obtained experimentally. In fact, testing deeper architectures provided even better results. However, we will stick with the defined architectures, since this number of neurons and layers catches enough information to make good predictions, while remaining relatively small.

Since there's no unique answer to what the best architecture should be for our task, we start by creating two simple architectures, as shown in Figure 4.1. Both architectures created for classification of the RHS_1 set of names. We call the CNN based networks for spectrogram classification SCNN (Spectrogram Classification Neural Network).

Both architectures are identical in structure and differ only in choice of activation functions. It will help us identify the best activations for our further exploratory analysis of the SCNNs. In order to somewhat decrease the number of parameters, we employ two MaxPool layers after each activation. As we proceed, we modify the architectures by adjusting complexity, regularization and hyperparameters in order to achieve the best performance for our task.



(a) SCNN var.2 with Sigmoid activations

(b) SCNN var.2 with ReLU activations

Figure 4.2.: SCNN architectures for RHS_1 with LSTM

Our second architecture should also be able to capture time-related features, from the underlying spectrograms. In order to extract the time dimension features, we use 2 stacked bidirectional LSTM layers, as shown in Figure 4.2. As in case of sequence prediction, we want to capture both backward and forward dependencies. We also add a dropout $p = 0.25$ into the first BiLSTM layer for regularization purposes. This will also help us to reduce the number of training parameters, since the number of features, that will be fed to the feedforward layers, dramatically decreases from Number of filters $\times$ Kernel dimensionality to simply the number of the BiLSTM hidden states. Note, that before we can feed the output data from the last convolution layer into the BiLSTM layers, we have to make the data sequential. Instead of flattening all the layers into one single long vector, we use the adaptive average pooling[2] along the frequency dimension. This way we get an average frequency for each time step.

We will see, that this approach produces already quite good results of RHS_1. Now, in order to classify more labels, we have to extend our SCNNs for RHS_2. We use the previous architecture and extend it to fit the purpose by increasing the number of neurons in the feedforward part. Since now there're ten times more labels to classify, we assume that we also have to capture more fine relations. Hence, we add one more convolution layer to be able to extract more features from the underlying training set and increase the number of hidden states.

Lastly, we try to extract more insights from the spectrogram structure. To keep the model light we use the same BiLSTM layers twice. We use the adaptive average pooling along the frequency and along the time dimension as it can be seen from Figure (A.6). This way we get two sequences that we put through the BiLSTM layers and then concat the output, with the idea to extract more valuable insights, that can

[2]similar to MaxPooling. It allows us to resize any given input to the required shape by taking the averages instead of max values

characterize the spectrograms better. Note, that in the implementation of the forward pass this way, we do implement this step after the convolutional layers.

All the variants listed above follow the same structure. We list all created architectures in the Table 4.2. In this table we can observe some other minor structural dissimilarities between the architecture. We use following notation: HS stands for Hidden States, $p = 0.25$ stands for 25% probability to drop out a node, $(3 \times 3) \times 64$ means that the convolutional layer consists of 64 filters with kernel $K \in \mathbb{R}^{3\times3}$.

Structure	var. 1	var. 2	var. 3	var. 4
Convolution layers	$(3 \times 3) \times 64$ $(3 \times 3) \times 128$ -	$(3 \times 3) \times 64$ $(3 \times 3) \times 128$ -	$(5 \times 5) \times 64$ $(3 \times 3) \times 128$ $(3 \times 3) \times 256$	$(5 \times 5) \times 64$ $(3 \times 3) \times 128$ $(3 \times 3) \times 256$
BiLSTM layers	- -	256 HS ($p = 0.25$) 256 HS	512 HS ($p = 0.25$) 512 HS	256 HS ($p = 0.25$) 256 HS
Feedforward layers	In: 64.000 Hid: 2.048 Out: 1.000	In: 256 Hid: 2.048 Out: 1.000	In: 512 Hid: 12.800 Out: 10.000	In: 512 Hid: 12.800 Out: 10.000
Supporting techniques				
Normalization	BatchNorm	BatchNorm	BatchNorm	BatchNorm
Regularization	Dropout ($p = 0.25$)	Dropout ($p = 0.25$)	Dropout ($p = 0.25$)	Dropout ($p = 0.25$)
Dim. reduction	MaxPool (2×2)	MaxPool (2×2)	MaxPool (2×2)	MaxPool (2×2)

Table 4.2.: Overview of the SCNN architecture variants

We summarize the training process and Table 4.3[3]. In this table we summarize the best models for each architecture, defined by the lowest loss and highest accuracy on the test data. We also list the epoch, at which the model was observed during the training. Note, that none of the introduced models are getting to the 100th epoch and converge fairly fast to the maximum training accuracy of 99.9%, which is our stopping criterium. We observe, that the models with the lowest loss and highest accuracy are SCNN var.2 (ReLU) for RHS_1 and SCNN var.3(Sigma) for RHS_3. We also notice, that all of the provided models do not differ by more than 5% of the test loss.

Moreover, the models, that were trained on the RHS_2 are producing significantly lower accuracy, than those that were trained on the RHS_1. The main reason behind it, is that we reduced the amount of augmented spectrograms for each label 5 times in RHS_2 in comparison to RHS_1 (from 100 to 20) in order to reduce the space complexity while increasing the number of classification labels. This effect is the direct consequence of the Corollary 3.2.0.1. Nevertheless, as we will see in Section 4.3, all models produce a high quality predictions during testing over the validation dataset by avoiding the idea of the exact matching.

[3]the exact training process can be retrieved from the training logs - Appendix B.2

4.3. Results

As outlined in Section 4.2, our main goal for the classification task is to decrease the number of possible candidates, by providing a reasonable subset of the names to work on. We then are able to produce the matching results by applying any of the classical metrics from Chapter 2, to distillate the most accurate matches from the subset. Clearly, if it was possible, we would solemnly rely on the predictions produced by the classification network. It is however rarely possible on practice to create such a perfect classificator.

Let $m = |LHS|$, which in our case is the length of our validation dataset - 4.000, and $n = |RHS|$, which on our case is either 1.000 for RHS_1 or 10.000 for RHS_2. We examine the generalization power of the listed architectures, by analysing the predictions $\hat{y} \in \mathbb{R}^{m \times n}$ against the actual labels of the validation dataset $y \in \mathbb{N}^n$. Note, that for each label y_i we get the vector of probabilities $\hat{y}_i \in \mathbb{R}^n$ for $i \in \{0, 1, \cdots, m-1\}$. We define an index map $I : \mathbb{R} \times \mathbb{R}^n \to \mathbb{N}$, $I(x, y) = i$, that maps the given value of the vector to its position in that vector and 0 otherwise. If we have a match, it should show the highest probability at the i-th position $I(\max(\hat{y}_i), \hat{y}_i) = y_i$, indicating the right prediction. We relax this assumption by taking a threshold $t \leq n$, which allows us to claim the match if the right label is located within the given threshold in a sorted $\hat{y}_i^*$ in a descending way, i.e. $\hat{y}^*_{i,j+1} \leq \hat{y}^*_{i,j}$ for $j \in \{0, 1, \cdots, n-1\}$. We say that the match for the label i is found, if $y_i \in \{I(\hat{y}^*_{i,0}, \hat{y}_i), \cdots, I(\hat{y}^*_{i,t}, \hat{y}_i)\}$, where $t = 0$ indicates the exact matching. We define the number of mismatches under the given threshold t by

$$N_{miss} = m - \Sigma_{i=0}^{m-1} \mathbb{I}_{\{y_i \in \{I(\hat{y}^*_{i,0}, \hat{y}_i), \cdots, I(\hat{y}^*_{i,t}, \hat{y}_i)\}\}},$$

where $\mathbb{I}$ is the indicator function. We use N_{miss} in Table 4.4 to identify the number of mismatches for a given model. If the model doesn't produce any mismatches, we write the maximum of the all positions under the given threshold, where the labels were identified.

Model	epoch	Best loss		epoch	Best acc		Num. labels
		Loss	Matches, %		Loss	Matches, %	
SCNN var.1 (ReLU)	24	19,5296	87,84	25	20,0292	87,93	1000,00
SCNN var.1 (Sigm)	20	17,8222	87,80	20	17,8222	87,80	1000,00
SCNN var.2 (ReLU)	17	**14,8561**	89,85	23	15,1312	90,45	1000,00
SCNN var.2 (Sigm)	35	19,9553	87,07	35	19,9553	87,07	1000,00
SCNN var.3 (ReLU)	17	126,5268	57,73	30	133,2071	60,3	10000,00
SCNN var.3 (Sigm)	24	**84,4647**	61,53	24	84,4647	61,53	10000,00
SCNN var.4 (ReLU)	18	137,9175	54,23	24	143,4971	56,02	10000,00
SCNN var.4 (Sigm)	28	97,8059	58,25	29	98,5025	58,29	10000,00

Table 4.3.: Training results

While giving the threshold and building the best cluster for each model under the given validation dataset is one approach to extract all possible matches, we can generalize it for any given data. Let's observe the distribution of the matches produced by SCNN var.1 (Sigm)[4] on the validation dataset (Figure 4.3). We see, that almost all matches were found instantly on the 0-th positions, i.e. they had the highest probability[5]. It tells us, that the accuracy of the model is very high and it generalizes well for the unseen data. However,

[4] for validation purposes we always use the model, that showed the lowest loss during the training
[5] The labels, that weren't matched assigned to the -1 position.

the highest probability wasn't assigned to all matches. But still they are high enough, so we can find them at the consecutive positions. Thus, our predictions are ordered in such a way, that the most likely matches positioned near to the top.

Judging by the histogram pattern, we can estimate the underlying distribution of the predictions. Since they all are produced by the same model, we can assume that the probabilities of $\hat{y}$ are independent and identically distributed. In our case we can visually assume, that the underlying distribution is the exponential distribution. Hence, we can estimate it by calculating first two moments from the sampled data. Particularly, the expected value of the underlying distribution will show us, at which position the potential match could be found:

$$\mathbb{E}[y_i] = \frac{1}{\lambda},$$

where λ is obtained using Maximum Likelihood Estimator. Hence, the dictionary search complexity using the SCNN reduces to $O(m \cdot \frac{1}{\lambda})$, which dramatically reduces the initial complexity $O(m \cdot n)$ observed in the Section 2.5 previously. In the Figure 4.4 we can see the how the probability density fits the aggregates of the positions of produced matches.

In order to obtain the reduced number of comparisons between LHS and RHS, we can use the approximate dictionary search from Definition 2.3.2 over the dictionary of names ordered according to the predictions of the outlined SCNN. We then can either restrict ourselves to the cluster of names created by threshold t with some expected error $< 1\%$, if the RHS is too big to handle, or use the whole RHS to conduct approximate searching. Ordering the dictionary allows us to reduce time complexity to the feasible magnitude.

In the Table 4.4 we can observe, that the number of mismatches for exact classification is still quite high in comparison to the classical approaches, it decreases very quick inside the threshold of $t = 300$. The amount of mismatched records inside the threshold already outperforms any of the introduced algorithms from the Chapter 2. It confirms our conclusion, that the main advantage of this approach is the ability to order the RHS in a way, that reduces the time complexity of the approximate dictionary search. All values in this table are average values obtained by repeating the experiments 10 times for each model. The uncertainty lies in the Tacotron2 implementation, that produces slightly different spectrograms each time. Nevertheless, our trained models are trained to be robust to such changes and produce consistent results in each run, having the slight deviation of ± 3 mismatches of what is listed in the table. The reason for it is the dynamical implementation of the "stop token" prediction, that may differ each time for the same output

Model	N_{miss}			$\mathbb{E}[X] = \frac{1}{\lambda}$	Num. labels
	$t = 0$	$t = 300$	$t = 1000$		
SCNN var.1 (ReLU)	357 (8.93%)	max. 120	max. 120	0.522	1000,00
SCNN var.1 (Sigm)	354 (8.85%)	max. 271	max. 271	0.705	1000,00
SCNN var.2 (ReLU)	281 (7.03%)	max. 162	max. 162	0.4542	1000,00
SCNN var.2 (Sigm)	368 (9.2%)	max. 152	max. 152	0.6105	1000,00
SCNN var.3 (ReLU)	1101 (27.52%)	12 (0.3%)	8.2602	10000,00	
SCNN var.3 (Sigm)	1042 (26.05%)	9 (0.23%)	7.451	10000,00	
SCNN var.4 (ReLU)	1135 (28.38%)	32 (0.8%)	9 (0.23%)	12.3912	10000,00
SCNN var.4 (Sigm)	1099 (27.48%)	35 (0.88%)	13 (0.33%)	9.1352	10000,00

Table 4.4.: Validation performance

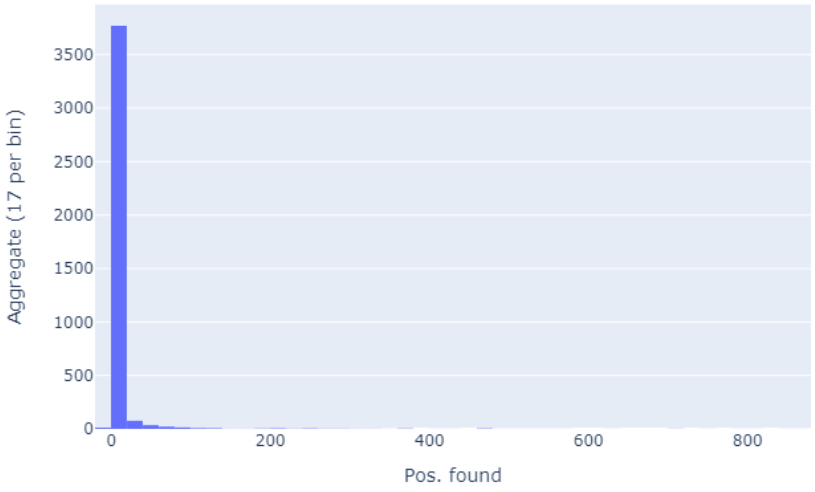


Figure 4.3.: SCNN var.3 with Sigmoid activations. Distribution of matches for $t = 1000$

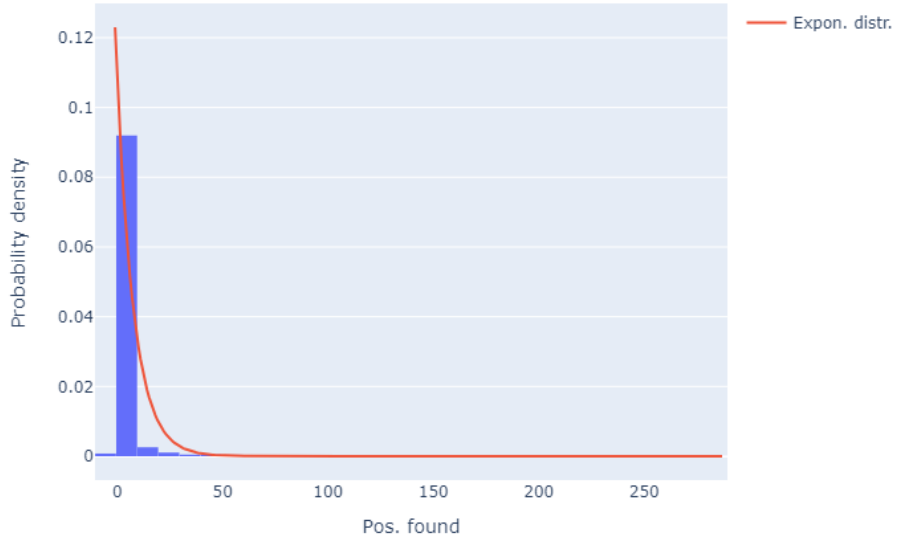


Figure 4.4.: SCNN var.3 with Sigmoid activations. Exponential distribution fit for $t = 300$ and $\lambda = 8.26$

[SPW+17]. The best performing model - SCNN var.3 (Sigm) - shows 4.55 times less mismatches as the best performing phonetic algorithm - Double Metaphone. Analysing the mismatches we also observe, that the most of them were made on the word, which letters exceed English alphabet and aren't transformed well by Tacotron2, e.g. original - 'Hyrkkoo' vs. misspelled: 'Hyrkkö' or original - 'Hruzik' vs. misspelled - 'Hruzík'. Other mismatches are often happen for the names, whose representation differs from the original

one by more than one mistake. All of this issues could be addressed in the further work, by fine-tuning the Tacotron2 learning and increasing the augmentation process.

Additionally, we notice that SCNN var.4 architecture doesn't show the expected performance increase comparing to the SCNN var.3, which leaves us with the lightest and best performant architecture for our cause. Nevertheless this approach also delivers quite reliable results and should be investigated further, e.g. by implementing two stacks of BiLSTM layers - one for each average pooling sequence, instead of one. Other ideas that wouldn't tested yet imply using of LSTM layers on other stages of the given architectures and using more deep feedforward architecture.

4.4. Conclusion and Future Work

In this theses, we introduced the pervasive problem faced by businesses across various fields – fuzzy name matching in data management. We highlighted the significance of valid and reliable data for effective utilization of Customer Relationship Management (CRM) systems and the challenges posed by errors, misspellings, and incomplete information in matching datasets. The goal of this thesis was to develop a mapping solution for fuzzy name matching using neural network approach, create an effective solution architecture, and perform the matching task efficiently.

We discussed the common metrics and phonetic algorithms used to address the problem. These techniques provided a foundation for understanding and measuring the similarity between character strings without relying on contextual or semantic information. By exploring these established methods, we gained valuable insights into the existing approaches and benchmarked their performance. Throughout the theses, we used IMDb dataset of names for demonstration purposes as a valid source of real names.

Consequently, we focused on setting up the theoretical ground and outlining the neural network architectures employed in our experiments. Neural networks have demonstrated remarkable capabilities in handling complex tasks, including natural language processing and pattern recognition. By leveraging the power of neural networks, we aimed to tackle the challenges of fuzzy name matching and improve the accuracy and efficiency of the matching process.

Lastly, we presented the development and training of eight neural network architectures specifically designed for fuzzy name matching. Through meticulous experimentation and fine-tuning, we sought to optimize the performance of these architectures and validate their effectiveness in mapping misspelled names to their valid counterparts. The results obtained from these experiments showcased the novel nature of our solution and its ability to outperform known techniques for approximate dictionary search in terms of both time complexity and accuracy.

Based on the comprehensive analysis and evaluation conducted throughout our research, it can be concluded that our proposed solution for fuzzy name matching represents a significant advancement in the field. By combining the insights gained from traditional metrics and phonetic algorithms with the capabilities of modern neural networks, we have successfully addressed the challenges associated with matching large datasets of character strings without contextual information or semantic sense. We established, that the solution powered by neural network architecture can be almost 5 times more precise than the best introduced phonetic algorithm - Double Metaphone.

The presented solution offers a superior level of accuracy, enabling businesses across diverse fields to manage their data more effectively and make informed decisions based on reliable information. Moreover, the demonstrated improvements in time complexity provide a practical advantage, allowing for efficient processing of large volumes of data within reasonable timeframes. This solution comes however with a cost of computational power, such as Cloud Computing, that will allow the training of larger datasets with more labels.

The achievements of this research highlight the potential of combining traditional methods with cutting-edge technologies to tackle complex data management challenges. However, it is important to acknowledge that the field of fuzzy name matching continues to evolve, and there are opportunities for further research and enhancements. Future work could explore additional neural network architectures, refine the training process and investigate the application of this solution in specific industry contexts involving large amount of labels.

One limiting aspect of our experiments with SCNNs is that we used relatively small models. The development of this solution should include the investigation in the context of training big number of labels for classification. As well as other methods should be investigated, such as pre-clustering the data and introduce labelling by cluster rather than by name, or training several models on the disjoint subsets of the RHS.

Bibliography

[ACC+17] Sercan Ömer Arik, Mike Chrzanowski, Adam Coates, Greg Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Jonathan Raiman, Shubho Sengupta, and Mohammad Shoeybi. Deep voice: Real-time neural text-to-speech. *CoRR*, abs/1702.07825, 2017.

[Aga18] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[AHMJ+14] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545, 2014.

[BB82] D.H. Ballard and C.M. Brown. *Computer Vision*. Prentice-Hall, 1982.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[BGJM16] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016.

[Bil04] Mikhail Bilenko. Learnable similarity functions and their applications to clustering and record linkage. pages 981–982, 01 2004.

[BJM06] Peter L. Bartlett, Michael I. Jordan, and Jon D. McAuliffe. Convexity, classification, and risk bounds. *Journal of the American Statistical Association*, 101:138–156, 2006.

[BK73] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, apr 1973.

[Boy11] Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16, 05 2011.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[Bui22] Minh Duc Bui. Improving cross-lingual representations by distillingmultilingual encoders into monolingual components, 2022.

[BYN98] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207)*, pages 14–22, 1998.

[CBS+15] Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, KyungHyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. *CoRR*, abs/1506.07503, 2015.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 91–100, New York, NY, USA, 2004. Association for Computing Machinery.

[Col09] P Collins. Definately* the most misspelled word in the english language (*it should be definitely). *The Daily Record*, June 2009.

[Coo99] Vivian Cook. Going beyond the native speaker in language teaching. *TESOL Quarterly*, 33(2):185–209, 1999.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CW08] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 160–167, New York, NY, USA, 2008. Association for Computing Machinery.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, mar 1964.

[Dau92] Ingrid Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.

[DH11] John Duchi and Elad Hazan. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 07 2011.

[Fre07] Kimmo Fredriksson. Engineering efficient metric indexes. *Pattern Recogn. Lett.*, 28(1):75–84, jan 2007.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[GL84] D. Griffin and Jae Lim. Signal estimation from modified short-time fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(2):236–243, 1984.

[Grö01] K. Gröchenig. *Foundations of Time-Frequency Analysis*. Applied and Numerical Harmonic Analysis. Birkhäuser Boston, 2001.

[GSC00] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 10 2000.

[Har54] Zellig S. Harris. Distributional structure. *WORD*, 10(2-3):146–162, 1954.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer New York, 2009.

[HZRS15a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[HZRS15b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[JM08] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 2. 02 2008.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KGB14] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014.

[Knu98] D.E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998.

[KSH12a] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[KSH12b] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. 25, 2012.

[KSKW15] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 957–966, Lille, France, 07–09 Jul 2015. PMLR.

[KW16] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[KZS+15] Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Skip-thought vectors. *CoRR*, abs/1506.06726, 2015.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[LBS+16] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *CoRR*, abs/1603.01360, 2016.

[Lei11] J.W. Leis. *Digital Signal Processing Using MATLAB for Students and Researchers*. Wiley, 2011.

[LH17] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[LJB+95] Yann Lecun, Larry Jackel, L. Bottou, A. Brunot, Corinna Cortes, John Denker, Harris Drucker, Isabelle Guyon, Urs Muller, E. Sackinger, Patrice Simard, and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. 01 1995.

[LJH15] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *CoRR*, abs/1504.00941, 2015.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[McM07] D. McMahon. *Quantum Computing Explained*. IEEE Press. Wiley, 2007.

[Mel96] B. Melichar. String matching with k differences by finite automata. In *Proceedings of 13th International Conference on Pattern Recognition*, volume 2, pages 256–260 vol.2, 1996.

[MS04] Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[Mye99] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, may 1999.

[MZ93] S.G. Mallat and Zhifeng Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[Nav01] Gonzalo Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31, 2001.

[NST05] Gonzalo Navarro, Erkki Sutinen, and Jorma Tarhio. Indexing text with approximate q-grams. *Journal of Discrete Algorithms*, 3(2):157–175, 2005. Combinatorial Pattern Matching (CPM) Special Issue.

[Ope23] OpenAI. [large language model]. *https://chat.openai.com*, March 2023.

[O'S87] D. O'Shaughnessy. *Speech Communication: Human and Machine, p.150. (4.2)*. Addison-Wesley series in electrical engineering. Addison-Wesley Publishing Company, 1987. p.150. (4.2).

[Pet86] James L. Peterson. A note on undetected typing errors. *Commun. ACM*, 29(7):633–637, jul 1986.

[Phi00] Lawrence Philips. The double metaphone search algorithm. *C/C++ Users J.*, 18(6):38–43, jun 2000.

[PMB12] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[Pra13] K.M.M. Prabhu. *Window Functions and Their Applications in Signal Processing*. 10 2013.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RJ07] P Rajkovic and D Jankovic. Adaptation and application of daitch-mokotoff soundex algorithm on serbian names. In *XVII Conference on Applied Mathematics*, volume 12, 2007.

[RKK19] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *CoRR*, abs/1904.09237, 2019.

[RMC16] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

[RO05] Luís M. S. Russo and Arlindo L. Oliveira. An efficient algorithm for generating super condensed neighborhoods. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Combinatorial Pattern Matching*, pages 104–115, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Rud73] W. Rudin. *Functional Analysis*. Higher mathematics series. McGraw-Hill, 1973.

[SDJ09] Ervin Sejdic, Igor Djurovic, and Jin Jiang. Time–frequency feature representation using energy concentration: An overview of recent advances. *Digital Signal Processing*, 19:153–183, 01 2009.

[Sel74] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26(4):787–793, 1974.

[Shi00] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227–244, 2000.

[SPW+17] Jonathan Shen, Ruoming Pang, Ron J. Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, R. J. Skerry-Ryan, Rif A. Saurous, Yannis Agiomyrgiannakis, and Yonghui Wu. Natural TTS synthesis by conditioning wavenet on mel spectrogram predictions. *CoRR*, abs/1712.05884, 2017.

[SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, 2014.

[STIM19] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2019.

[Stu07] Bob Sturm. Stéphane mallat: A wavelet tour of signal processing, 2nd edition. *Computer Music Journal - COMPUT MUSIC J*, 31:83–85, 09 2007.

[SVN37] S. S. Stevens, John E. Volkmann, and Edwin B. Newman. A scale for the measurement of the psychological magnitude pitch. *Journal of the Acoustical Society of America*, 8:185–190, 1937.

[The19a] Inc. TheMathWorks. Short-time fft, 2019.

[The19b] Inc. TheMathWorks. Short-time fourier transform, 2019.

[Ukk85] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132–137, 1985.

[UP20] Stefan Ulbrich and Marc Pfetsch. Optimization methodsfor machine learning. *Lecture Notes*, pages 9–42, 2020.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, jan 1974.

[WM92] Sun Wu and Udi Manber. Fast text searching: Allowing errors. *Commun. ACM*, 35(10):83–91, oct 1992.

[WSS+17a] Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: A fully end-to-end text-to-speech synthesis model. *CoRR*, abs/1703.10135, 2017.

[WSS+17b] Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: A fully end-to-end text-to-speech synthesis model. *CoRR*, abs/1703.10135, 2017.

[XBK+15] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015.

[YXZ+22] Suorong Yang, Weikang Xiao, Mengcheng Zhang, Suhan Guo, Jian Zhao, and Furao Shen. Image data augmentation for deep learning: A survey, 2022.

[Zal06] Maria Zaliznyak. Telefonica's dispute over an accented domain in chile. *https://www.multilingual-search.com/telefonica´s-dispute-over-an-accented-domain-in-chile/30/09/2006/*, 2006.

[ZLCO22] Zhenxun Zhuang, Mingrui Liu, Ashok Cutkosky, and Francesco Orabona. Understanding adamw through proximal methods and scale-freeness. *CoRR*, abs/2202.00089, 2022.

[ZLLS21] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[Zo88] W. Zhang and others. Shift-invariant pattern recognition neural network and its optical architecture. *Proceedings of annual conference of the Japan Society of Applied Physics*, 1988.

A. Tables and Figures

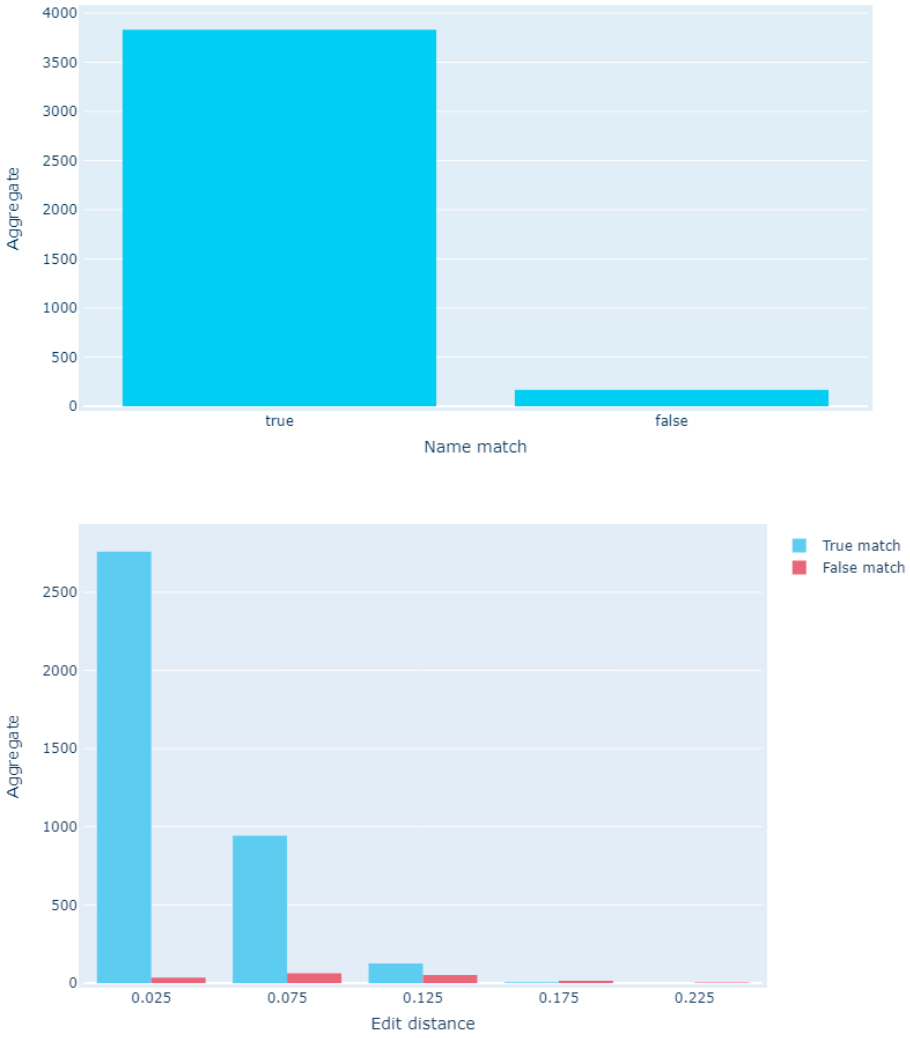
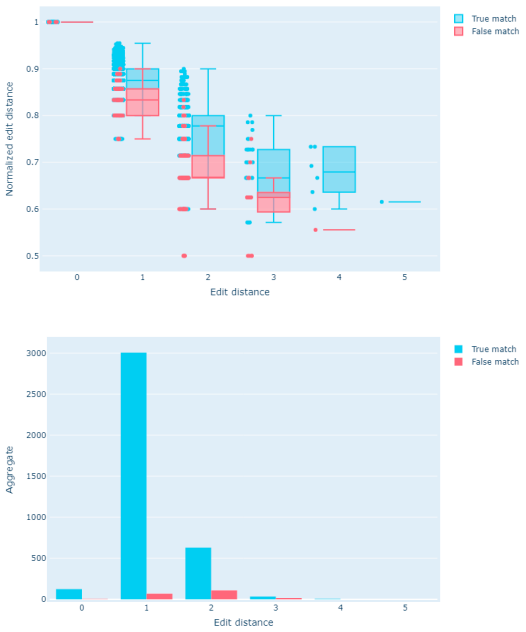
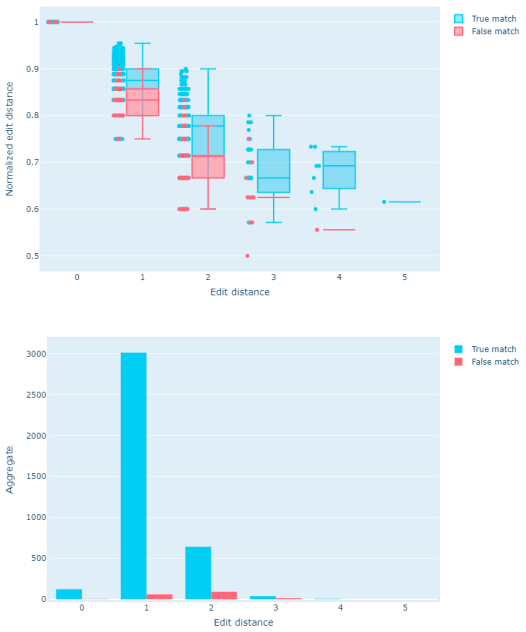


Figure A.1.: Distribution of matches using Jaro-Winkler Similarity

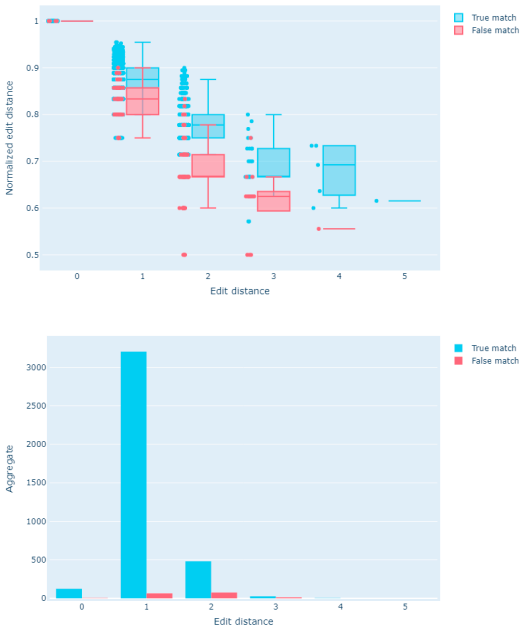


(a) Matching by regular distance

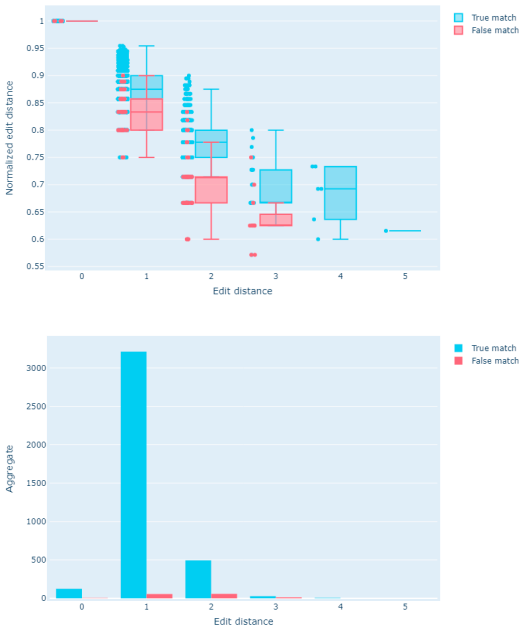


(b) Matching by normalized distance, Eq. 2.5

Figure A.2.: Distribution of matches using Levenshtein Distance

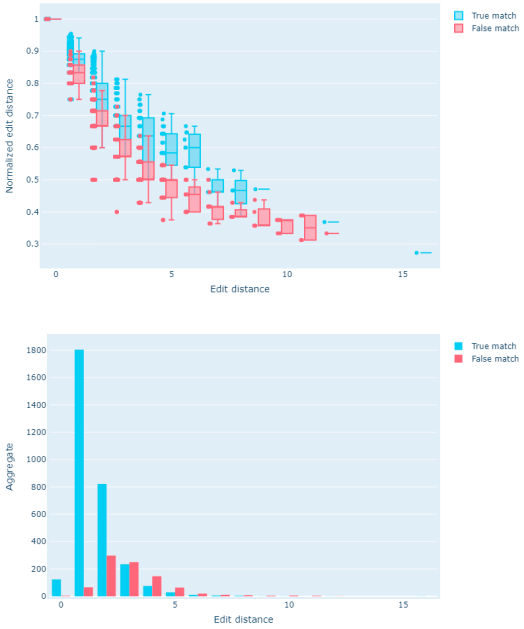


(a) Matching by regular distance

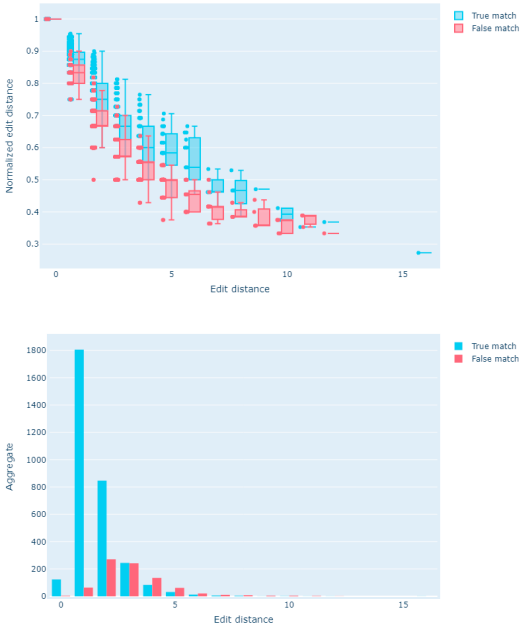


(b) Matching by normalized distance, Eq. 2.5

Figure A.3.: Distribution of matches using Damerau-Levenshtein Distance

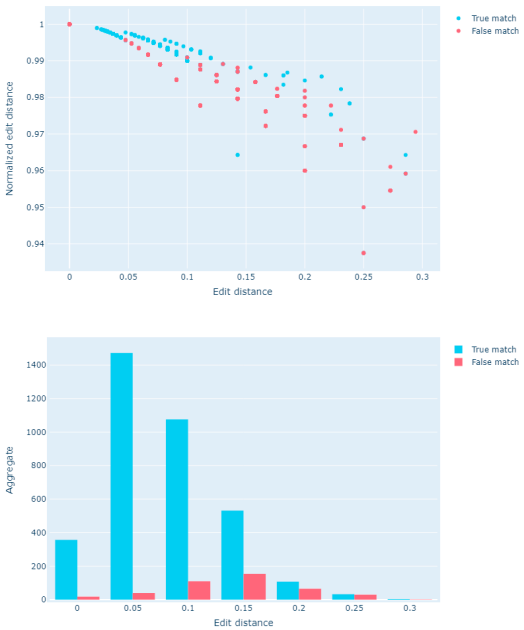


(a) Matching by regular distance

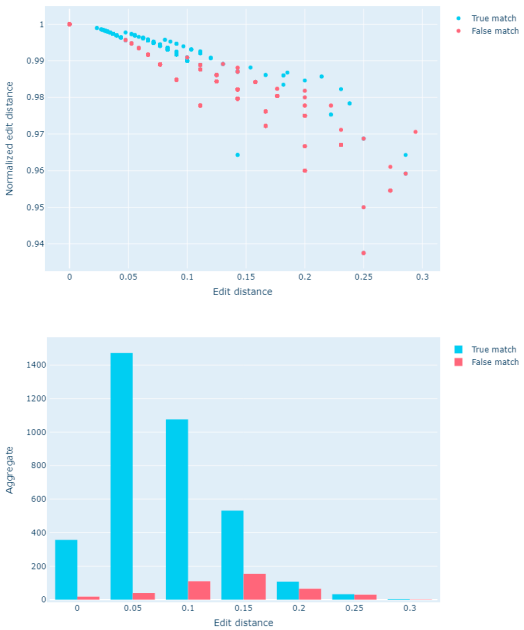


(b) Matching by normalized distance, Eq. 2.5

Figure A.4.: Distribution of matches using Hamming Distance



(a) Matching by regular distance



(b) Matching by modified distance, 2.5.1

Figure A.5.: Distribution of matches using Jaccard Similarity

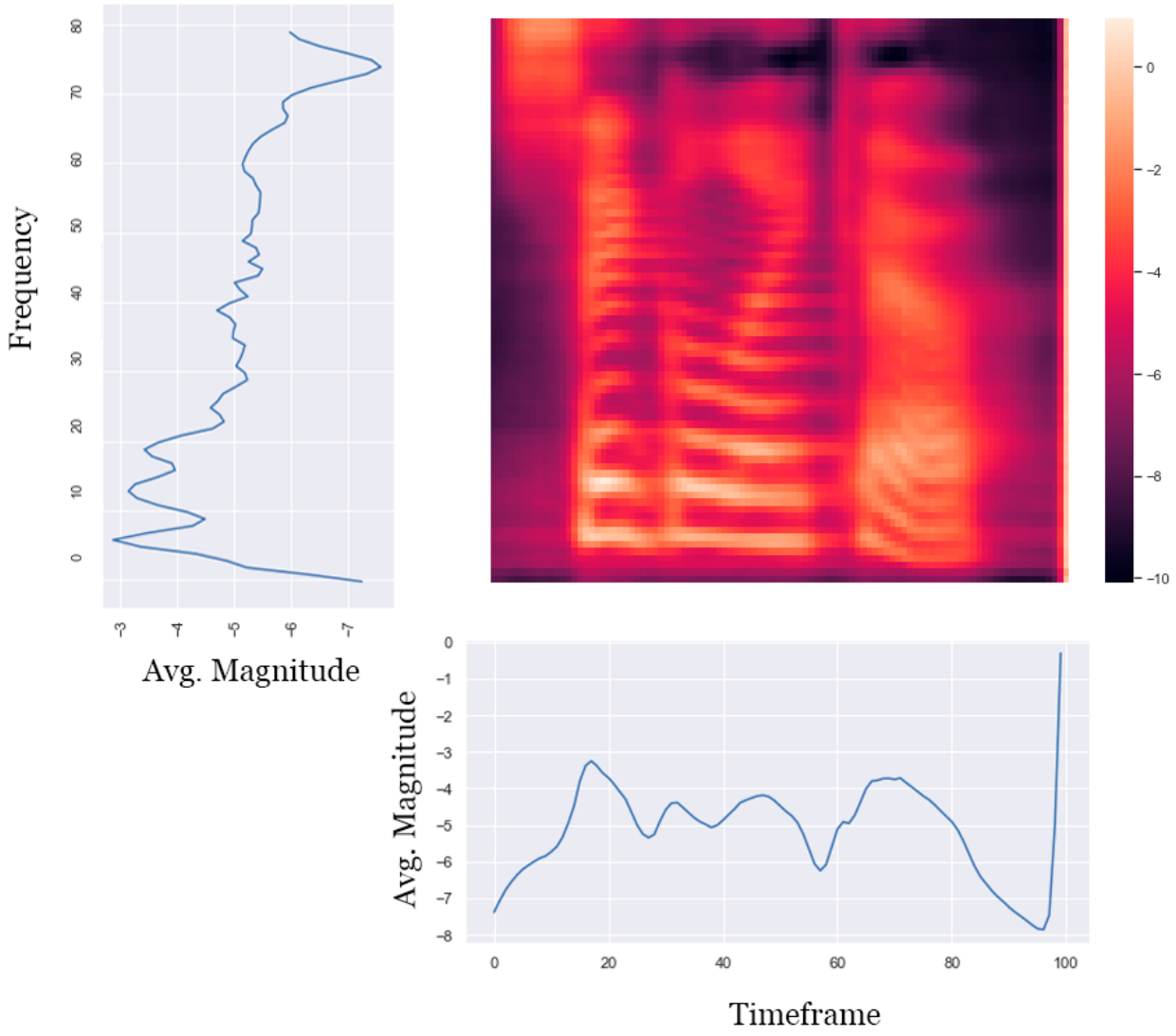


Figure A.6.: Average Pooling technique on the mel spectrogram along axis

Table A.1.: Supervised similarity learning: Active labelling of Record Linkage by Dedupe.io

First Name	Second Name	Label
Tremblay-Beauvaiss	Tremblay-Beauvais	y
Krzywicka-Kaindell	Krzywicka-Kaindel	y
Lakshiminarayana	Lakshiminarayanan	y
Wereschtschakoo	Wereschtschako	y
Wereschtschakow	Wereschtschako	y
Kabosuu	Kabosu	y
Tiopezz	Tiopez	y
Tremblay-Beauvaisse	Tremblay-Beauvais	y
Ahlstrin-Muniecck	Ahlstrin-Muniec	y
Druvr	Druv	y
Ekiss	Ekis	y
Ohovenk	Ohoven	y
Bedoeva	Bedoev	y
Lakshiminarayan	Lakshiminarayanan	y
Kallipolitisss	Kallipolitis	y
Futtersonne	Futterson	y
Veeragavanm	Veeragavan	y
Zulkarniany	Zulkarnian	y
Macarayg	Macara	n
Macaraigg	Macara	n
Krzywica-Kaindel	Krzy	n
Young-ian	Young-Ian	y
Dundjekar	Dundgekar	y
Ciofas	Ciofassa	u
Sukariya	Sukari	u
Llanodosaas	Llanodosa	y
Teig	Teighe	n
Bylo	Bylotas	n
Matt	Mattuicci	n
Mov	Movva	n
Bur	Burres	n
Bur	Burduj	n
Mykra-Nieminen	Mykrä-Nieminen	y
Muller-Morelli	Müller-Morelli	y
Foure	Foureau	u
Waukeee	Wauke	y
Batistellaa	Batistel	n
Meisner	Meisnerová	n
Sukariyah	Sukari	n
Halicekk	Halic	n
Haliceck	Halic	n
Lahrzaoui	Lahrz	n
Zajacownaa	Zajacówna	y
Weichnerieder	Leichner	n
Saparovskyi	Saporovsky	y
Ostafinsk	Ostapinski	u

B. Program Code/Resources

The source code used during this thesis is available on Github under the link https://github.com/burik193/Master_thesis and instructions on how to run the code. We implement the whole process in three phases: *spectrogram generaion*, *model training* and *model validation*. Each part is represented in code as a Jupyther Notebook. The code is commented and grouped in the easy-to-use frameworks.

We define several supporting modules, that we will use during all the phases. Main module is utils.py. It defines all the global functions, that we will need:

Here we list the created functions and their specific functionality:

- Function "**drop_nulls**": Drops all columns from a tensor, that have no (low) variance. Often, when creating batches, Tacotron2 produces blank columns in the mel spectrogram to fit the overall number of timeframes.
- Function "**get_matrix**": Converts a list representation of a keyboard layout into a numpy matrix. List representation matrix is a list of character sequences iterating with spaces, that imitate the keyboard. The function by default uses English keyboard.
- Function "**get_nearest**": Gets one nearest character to a given key in a matrix representing a keyboard layout randomly. This function imitates a human behaviour of misclicking the key on a keyboard.
- Function "**common_mistakes**": Generates common mistakes (or variations) of a given name. Following mistakes are supported - misclicks, permutations, missing letter, additional space, additional letter. The number of generated mistakes depends on the length of a name.
- Function "**get_spectrogram**": Applies STFT on a given audio signal and produces a spectrogram.
- Function "**plot_spectrogram**": Creates a heatmap of the given spectrogram
- Class "**Tacotron2_modified**": Overrides the method "**encode batch**" of the original Tacotron2 implementation by correcting the problem of unsorted list of names. Original class doesn't work correctly, if the batch of names isn't sorted by length.
- Class "**EarlyStopper**": A simple class, that can be used for early stopping during the training with specified *patience* (num. epochs) and *delta* parameters.

Other supporting modules include model_parser.py and record_linker.py.

First one is utilized in the Model Validation Notebook. It provides a simple interface to load the desired model architecture and its weights. Note, that this module create a new local environment variable. If this behaviour is undesired, one has to define the architecture manually and then load the trained weights. Both can be found in the provided "models" folder.

Second one is utilized during the benchmarking in Section 2.5. It employs the Learnable Similarity Metric Algorithm by Dedupe.io. One can manually pre-train this model to test the concept. Otherwise, it's utilized in the Chapter 2. Benchmarking.ipynb.

The entire code and a PDF version of this thesis is also contained in the USB stick attached to this thesis.

B.1. Spectrogram generation Code

We start by creating a pipeline for mel spectrogram generation. The corresponding Notebook called Chapter 4. Spectrogram generation.ipynb.

First, we import the necessary libraries and declare the global settings, that will help us throughout the Notebook.

The Notebook proceeds with a section, where one can experiment with the given tools: Tacotron2 - mel spectrogram generation and Tacotron2 - HIFIGAN Vocoder.

Consequently, we proceed with creating of the RHS. We start with import of the original data by IMDb, Inc and proceed with the preprocessing, that was outlined in the Section 4.2.1. We then apply data augmentation techniques, to create distorted names from the original ones to increase model robustness.

We conclude the Notebook by establishing mel spectrogram generation pipeline, that iterates over the prepared set of names batchwise and creates and stores their respective mel spectrograms representations in form of torch tensors. The corresponding labels are stored in the separate file.

B.2. Model Training Code

We proceed with Notebook, that serves us a main ground for model architecture creation and training. The corresponding Notebook called Chapter 4. Model Training.ipynb.

We begin with importing the necessary libraries and declare the global settings, that will help us throughout the Notebook as previously.

We proceed by importing the created mel spectrograms and corresponding labels, as well as a name-label dictionary. After the necessary data is imported we put it into a TensorDataset - a mechanism provided by pyTorch for a simple splitting and iteration over the spectrogram-label pairs batchwise. We then declare the hyperparamters for the training and split the TensorDataset into train and test parts.

In the next step we initialize all the presented architectures and encourage to create new ones. After choosing the desired architecture we initialize the loss function, optimization algorithm, optionally - initialization weights and a scheduler, that will dynamically reduce the learning rate if we hit the plateau during the training.

Lastly, we present the possibility to make quick tests with the trained model in the "Instant Evaluation Logic" Section.

All produced models are stored and will be provided together with the training logs. Training logs will give the possibility to inspect the training process more carefully using Tensorboard.

B.3. Model Validation Code

In the last Notebook, we create a framework to test the desired model. The corresponding Notebook called Chapter 4. Model Validation.ipynb.

We start by initializing necessary imports and global settings. We then chose the RHS, that we want to work on, since they contain different labels. I.e. the 1.000 labels from RHS_1 do not correspond to the same names from RHS_2, even though RHS_1 is the subset of RHS_2. The reason behind it lies in the simplifying approach to the Spectrogram Generation. Since it was used twice, we create the labels twice for two RHS. This distinction is not necessary and depends solemnly on the personal choice.

We then import the validation data. Note, that the validation data itself was produced using ChatGPT3.5 and verified manually row-by-row. One can reassure, that there're no repeating mistakes and every name and misspelling across the dataset are unique. We turn the dataset into a dictionary, where every misspelling correspond to the same label and transform them into the set of mel spectrograms, imitating the LHS.

Afterwards we initialize the desired trained model. The list of the models can be found in the Table 4.3. Now, the initialized model can be used for predictions using the LHS. We then have a possibility to analyse the outputs and the consecutive cells, including different parameters for threshold t and fitting the desired distribution to the matching samples.

Lastly, we provide with the framework to analyse the trained weights from the underlying model.

B.4. Benchmarking Code

We also present the framework for benchmarking the algorithms introduced in the Chapter 2. The corresponding Notebook called Chapter 2. Benchmarking.ipynb.

Here we introduce two major global functions: "**find_candidates**" and "**find_duplicates**". The first function implements the greedy search. Its inputs are the LHS and RHS as well as the Edit Distance, that should be employed during the search phase. The function implements a double loop and iterates over the LHS, comparing each name to each name from the RHS, while saving the lowest found distance for each name. Its output is a Pandas DataFrame, that contains in each row one LHS name and one found RHS name with the distance between then. Apart from that, the DataFrame also contains information about the labels of both names. This creates for us an opportunity to check the quality of the match. If both labels are the same, then the match was correct. We consequently provide a framework to analyse the quality of matches for each metric visually.

Apart from metrics, we also include every presented phonetic algorithm. Since they rely on exact matches, we use the inner join of two dictionaries to produce matches. This approach however produces also a large number of false positives due to ambiguity of name representations.

Lastly, we present the learnable similarity metric, which was trained beforehand.

We summarize all the results visually in the last section.

Package	Version
conda	22.9.0
cuda	11.7.1
ipykernel	6.4.1
ipython	7.29.0
jellyfish	0.11.2
librosa	0.9.2
numpy	1.19.5
torch	1.13.1 + cu117
torchaudio	0.13.1 + cu117
torchvision	0.13.1
pandas	1.3.4
pathlib	1.0.1
plotly	5.13.0
scipy	1.7.1
speechbrain	0.5.13
tensorflow	2.11.0
tensorboard	2.11.2

Table B.1.: List of the main packages used during the implementation