MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Hyphenation using deep neural networks

TDK Thesis

Author
Gergely Dániel Németh

Advisor
Judit Ács

October 27, 2017

Contents

Kivonat

A szótagoló algoritmusok az elválasztás feladatának számítógépes megoldásai és legtöbbet dokumentumok tördelésekor használják, azonban a szavak elválasztásának hatása van a költészetben vagy akár a szövegfelolvasó, szövegfelismerő alkalmazások fejlesztésében is.

A mélytanulásos módszerek előretörésével megnőtt az igény a nyelvtechnológiai problémák gépi tanulás alapú megoldására is. Az online elérhető corpus adatbázisok mennyisége elősegíti, hogy a módszereket kipróbálhassuk olyan problémákon is, mint a szótagolás.

Az itt következő dolgozatban a szerző egy új fajta szótagoló algoritmust ismertet. A jelenleg használt szótagoló algoritmusok rövid ismertetője és három nyelvtechnológiában elterjedt neurális háló bemutatása után rátér ezek alkalmazására a szótagolás terén.

Abstract

Hyphenation algorithms are the computer based ways of syllabification and mostly used in typesetting, formatting documents but the hyphenation of words affects the poetry as well as text-to-speech and speech recognition algorithms.

The rise of the deep learning paradigms increased the demand for solving natural language processing problems by machine learning. The amount of online available corpora facilitates the use of these paradigms in problems like hyphenation.

In the following thesis, the author shows a new hyphenation algorithm. After a short summary of the currently used algorithms, the study describes three neural networks used in Natural Language Processing and shows a way of their application in the field of hyphenation.

Introduction

Hungarian children learn the way of syllabification in their early teens. The hyphenation rules are clearly defined [1], and after years of practice most people use it naturally and this is seen as part of the common knowledge.

In Hungarian, hyphenation depends mostly on the word itself and not from the surroundings. However, some words have different hyphenations based on the meaning which can only be derived from the context. For instance, *me-gint* (again) and *meg-int* (warn). Although it is a known issue, most hyphenation algorithms use only the word to insert the hyphens and defines exactly one hyphenation for a word.

Hyphenation algorithms

Commonly used hyphenation algorithms are based on the methods defined in the first version of TEX [12]. It is a pattern based hyphenation algorithm with thousands of manually chosen patterns. However its accuracy is good enough for the Research Institute for Linguistics of the Hungarian Academy of Sciences to collect its own patterns and use it in the online Hungarian hyphenation portal [16]. On this website[1] users can verify their syllabification.

Language corpus

In natural language processing (NLP) corpus is a collection of linguistic data. Although it is not only a group of words or sentences but linguistic information and statistics of the words, in this thesis only the words are used.

While we have enormous amount of words in corpora, it is dropped quite low in terms of pre-hyphenated words. One way to create hyphenated words is to use already available hyphenation algorithms.

[1] http://helyesiras.mta.hu/helyesiras/default/hyph

Chapter 1

Hyphenation algorithms

There are two main types of the common hyphenation algorithms: rule-based and dictionary-based. The world of open-source software *de facto* uses the TEX's hyphenation algorithm.

The following is a good summary of the early TEX hyphenation algorithm by Liang [15, page 3]:

> The original TEX hyphenation algorithm was designed by Prof. Knuth and the author [of his book] in the summer of 1977. It is essentially a rule-based algorithm, with three main types of rules: (1) suffix removal, (2) prefix removal, and (3) vowel-consonant-consonant-vowel (vccv) breaking. The latter rule states that when the pattern 'vowel-consonant-consonant-vowel' appears in a word, we can in most cases split between the consonants. There are also many special case rules; for example, "break vowel-q" or "break after ck". Finally a small exception dictionary (about 300 words) is used to handle particularly objectionable errors made by the above rules, and to hyphenate certain common words (e.g. pro-gram) that are not split by the rules. The complete algorithm is described in Appendix H of the old TEX manual.

This book of Liang was published in 1983 after the algorithm described in it had became the default hyphenation of the TEX82 version of TEX. The most important innovation of the new algorithm was that it used hyphenation *patterns* which are basically schemes that the program can look for in the words.

The latest version of TEX uses the Hunspell's hyphenation algorithm [17].This method is based on Liang's algorithm completed with *non-standard hyphenation extensions.*

The following sections show the basics of Liang's algorithm and the Hunspell [15, 17].

1.1 Liang's algorithm

The basic concepts of Liang's algorithm are the hyphenation patterns. The process of hyphenating the word `hyphenation` is the following:

First of all, the algorithm checks if the word is in the exception list. These are essentially hard-coded hyphenations of full words. *Hyphenation* word is not in the exception list.

Secondly, the algorithm inserts a dot in both end of the word. This will be used when the algorithm checks for patterns used only in the beginning or the ending of words.

```
.hyphenation.
```

The next step is the pattern matching. The patterns used in Liang's algorithm consist of characters and numbers. When it searches for matching patterns in a word, it skips the numbers and compares the letters with the word.

Hyphenation's patterns are the following: `hy3ph, he2n, hena4, hen5at, 1na, n2at, 1tio, 2io` [15, page 37]. Placing the patterns in the right position and inserting the numbers in the patterns between the letters we got Figure 1.1.[1]

```
. h y p h e n a t i o n .
  h y3p h
        h e2n
        h e n a4
        h e n5a t
           1n a
            n2a t
               1t i o
                 2i o
-------------------------
.0h0y3p0h0e2n5a4t2i0o0n0.
  h y-p h e n-a t i o n
```

Figure 1.1. *The hyphenation of 'hyphenation' by Liang's algorithm*

After matching all the patterns the algorithm inserts one number between every two letters of the word. If there was no pattern match with a number at that position, it is a 0, otherwise the maximum of the matching numbers.

The final step is that the algorithm hyphenates at odd numbers and does not hyphenate if it is even. Therefore, the hyphenation of *hyphenation*: `hy-phen-ation`.

1.1.1 Choosing patterns

The effectiveness of the algorithm depends on the choice of patterns. In 1982, Liang's goal was to achieve good error rate with relatively low use of disk space. The final program used around 4500 patterns, occupied 25K bytes of storage and found 89% of the hyphens in the dictionary it was tested on. The complete hyphenation dictionary of these words would be 500K bytes.

The choice of patterns was manual but a computer scientist in the time of machine learning says that it's a typical deep learning problem.

[1]The visualization method comes from Németh's article [17].

1.2 Hunspell

Hunspell's hyphenation algorithm is currently used in $\TeX$ and OpenOffice. It is based on Liang's work with Sojka's non-standard hyphenation extensions [20] and was published by Németh in 2006 [17].

The finding of non-standard hyphenations uses Liang's patterns and an extension to character replacement. For example the German word *Zucker* with the non-standard hyphenation `c1k/k=k` gets a new *k* letter before the hyphen: *Zuck-ker*.

The different languages use different non-standard patterns whose sizes and types vary significantly. Here we summarize the list of Hungarian non-standart hyphenations.

1.2.1 Hungarian non-standard hyphenation patterns

The Hungarian language uses simplified forms to represent its double digraph and trigraph consonants ($sz + sz \rightarrow ssz$, $dzs + dzs \rightarrow ddzs$, etc.), but when the word is hyphenated into two part around these letters it undoes the simplification (*sz-sz, dzs-dzs*). A classic example: *asszonnyal* → *asz-szony-nyal*.

Its difficulty comes from the fact that these digraph-simplifications can represent two letters (a monograph and a digraph) like *ggy* as g-gy in *meg-gyúj-tot-ta*. So the algorithm must know whether its two digraphs are simplified or not. Table 1.1. illustrates these extensions in the v20110815 version of the Magyar Ispell (Hunspell Hungarian pattern dictionary)[2].

The meaning of the patterns showed in the `fröc5csen/cs=,4,1` example: The part before the slash character is the standard hyphenation pattern. After it comes the redefinition: `cs=,4,1` means that from the 4th character of the pattern in 1 letter long it changes the characters as *cs=* where the = represents the possible hyphen. In this case, the letter *c* will be replaced by the *cs=* so if it happens to be a hyphenation break, the algorithm will add a *s* character into the word.

Digraph	Example	No. of patterns
ccs	fröc5csen/cs=,4,1	96
ggy	meg3gyes/gy=,3,1	39
lly	gal5lya/ly=ly,3,3	20
nny	szen5nye./ny=ny,4,3	97
ssz	hos5szal./sz=,3,1	1727
tty	hat5tyú/ty=ty,3,3	18
zzs	z5zsel./zs=zs,1,3	4

Table 1.1. *The hyphenation of 'hyphenation' by Liang's algorithm*

[2]http://magyarispell.sourceforge.net/

1.2.2 Hyphenation errors of the Hunspell

Most of the hyphenation errors come from the fact that the Hunspell's creators wanted to create a typesetting algorithm so they decided to not hyphenate one letter long word parts at the begining and the ending of the words. However, when it comes to compound words, these one-letter parts can be in the middle of the word. There are some examples of hyphenation errors in Table 1.2.

Hyphenation by Hunspell:	**Correct hyphenation:**	**Error type:**
au-tó-val	a-u-tó-val	one-letter
szem-üveg-gel	szem-ü-veg-gel	one-letter
has-izom	has-i-zom	one-letter
messze	mesz-sze	no hyphen
fölül	föl-ül	no hyphen
top-ikok	to-pik-ok	wrong place[3]
vi-deó	vi-de-ó	one-letter
geo-dé-zia	ge-o-dé-zi-a	no hyphen
diszk-ri-mi-na-tív	disz-kri-mi-na-tív	wrong place

Table 1.2. *Hyphenation errors in Hunspell*

[3]This word is hyphenated wrongly even in the online hyphenation tool of the Research Institute for Linguistics.

Chapter 2

Neural networks

The following summary of neural networks is based on the *Deep learning in neural networks: An overview* by Schmidhuber [18].

2.1 Feedforward neural network

A feedforward neural network approximates any given function f as $y = f(x, T)$ where T represents those parameters with which the model can learn to achieve the best approximation. These networks are called feedforward because the information flows through the function from x to inner (hidden) parts and finally to y. There are no directed cycles or loops in the network.

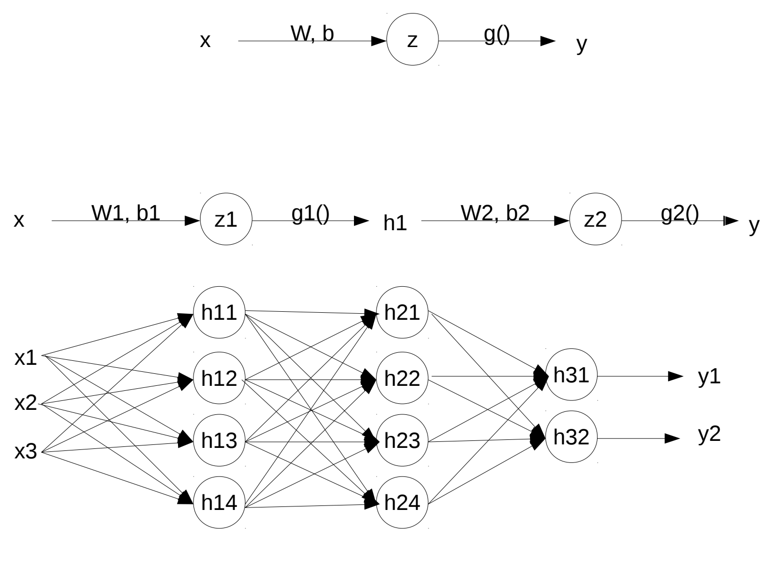

Figure 2.1. *Basics of Feedforward Neural Networks (F(F)NN)*

The simplest model is a single-layer perceptron which has a weight W and bias b, so for an input x can compute the $\hat{y} = Wx + b$ function where the learning method optimizes the weight W and the bias b. Later on researchers showed that adding a non-linear *activation* *function* to it can fasten the learning method and makes it usable in non-linear functions [4]. So from the $\hat{y} = Wx+b$ the function changed to $z = Wx+b$ and the prediction became $\hat{y} = g(z)$.

The deep neural network's name comes from that instead of a single-layer perceptron the output of the above equation $g(z)$ now called as h_1 is used as the input of the next layer and so for many-many layers. So for the first layer it became $z_1 = W_1x+b_1$ and $h_1 = g_1(z_1)$ and to the second layer: $z_2 = W_2h_1+b_2$ and $h_2 = g_2(z_2)$ and so on, until the last, nth layer where the a_n became the prediction $h_n = \hat{y}$. Figure 2.1. summarizes idea of feedforward networks.

A single iteration of training consists of a forward step where the model predicts $\hat{y}$, the evaluation step where the model compare the $\hat{y}$ and y with some type of gradient descent [7] and lastly a backpropagation step where the model updates the weights [8].

2.2 Convolutional neural network

Convolutional neural networks were introduced to solve image recognition problems [6]. A convolutional neural network has a filter (or kernel) which is sliding around the input (image) and multiplying the values in the filter (the weights of the filter) with the original input values (pixels). Summing up these values the network get a single number for every position of the filter. This will be the output of the layer. The size of the output depends on the filter size and the parameter *strides* which defines the steps of the filter's sliding.

Let a_{ij} be the cell (pixel) of the input (image) in the ith row and jth column and f_{ij} the cell of the filter, while h_{ij} the output. Thus the first cell of convolutional layer's output is

$$h_{11} = \sum_{x=1..k,y=1..l} a_{xy}f_{xy},$$

(where k and l are the height and width of the filter respectively), and assuming that the stride is 1, the h_{ij} is:

$$h_{ij} = \sum_{x=i..(i+k),y=j..(j+l)} a_{xy}f_{(x-i+1),(y-j+1)}.$$

Figure 2.2. illustrates a convolutional network with a $(3,3)$ kernel.

Kim, Jernite, Sontag and Rush showed a way of using 1 dimensional convolutional neural networks and LSTM networks in character sequences [10], where the filter size says that how many character should be included into the convolution.

2.3 Recurrent neural network

A recurrent neural network (RNN) is suited for modelling sequential phenomena. At each time step t, an RNN takes the input vector $x_t \in \Re^n$ and the hidden state vector $h_{t-1} \in \Re^m$

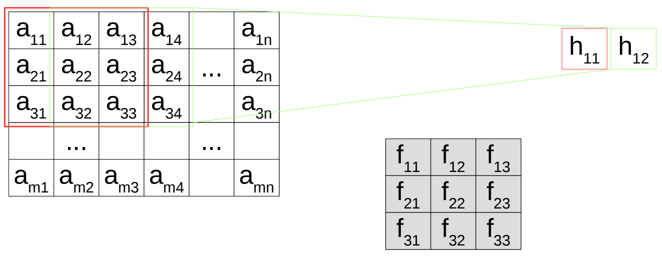

Figure 2.2. *Basics of Convolutional Neural Network (CNN)*

and produces the next hidden state h_t by applying the following recursive operation: $h_t = f(Wx_t + Uh_{t-1} + b)$. In theory, an RNN can store all information in h_t, however learning long-range dependences with it is difficult due to vanishing/exploding gradients [2].

2.3.1 Long short-term memory

Long short-term memory (LSTM) [9] addresses the problem of learning long range dependencies by augmenting the RNN with a memory cell vector $c_t \in \Re^n$ at each time step. Concretely, one step of an LSTM takes as input x_t , h_{t-1}, c_{t-1} and produces h_t , c_t via the following intermediate calculations:

$$i_t = \sigma(W^i x_t + U^i h_{t-1} + b_i)$$

$$f_t = \sigma(W^f x_t + U^f h_{t-1} + b_f)$$

$$o_t = \sigma(W^o x_t + U^o h_{t-1} + b_o)$$

$$g_t = \tanh(W^g x_t + U^g h_{t-1} + b_g)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

Here $\sigma(\cdot)$ and $\tanh(\cdot)$ are the element-wise sigmoid and hyperbolic tangent functions, $\odot$ is the element-wise multiplication operator, and i_t, f_t, o_t are referred to as *input*, *forget*, and *output* gates. At $t = 1$, h_0 and c_0 are initialized to zero vectors. Parameters of the LSTM are W^j , U^j, b^j for $j \in i, f, o, g$. See the visualisation of an LSTM unit in Figure 2.3.[1]

Bidirectional recurrent neural networks are based on the principle to split the neurons of a regular RNN into two directions, one for positive time direction (forward states), and another for negative time direction (backward states) [19]. In terms of characters in a word it means that the letters can affect their surroundings on both sides.

[1]Visualisation and more : http://colah.github.io/posts/2015-08-Understanding-LSTMs/

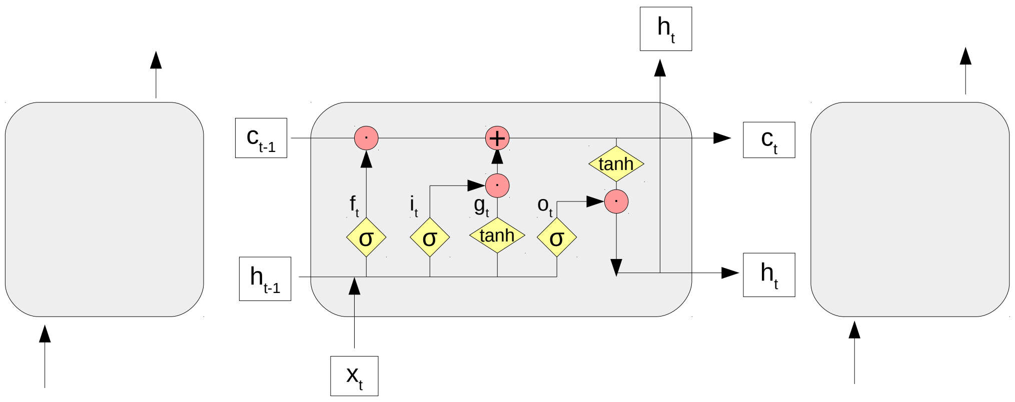

Figure 2.3. *Basics of Long short-term memory (LSTM)*

2.4 Neural network APIs

When it comes to developing deep learning systems, recently the use of neural network APIs are rising both in industrial and scientific field. Its main reasons are the quickness of designing and the quickness of adaptation. The active community behind these APIs makes sure that the systems are up-to-date with the recent scientific results.

The most important APIs: TensorFlow[2], Torch7[3], Keras[4]. The programmings of this thesis are based on Keras[3].

2.5 Neural networks for hyphenation

There is a paper about a similar task: A Norwegian backpropagation neural network based hyphenation algorithm [14], however it was tested only on a very small database of words.

[2]TensorFlow: https://www.tensorflow.org/
[3]Torch7: http://torch.ch/
[4]Keras: https://keras.io/

Chapter 3

Data preprocessing

In the following chapter I show the preprocessing methods I had used before I started the training. Data sizes mentioned here are generated by using the first 100 000 rows of the Hungarian Webcorpus's frequency list file.

Table 3.1. illustrates the preprocessing steps and the amount of dropped words.

	Origin	Cleaning	Non-standard	Special chars	Long words	Final
Words	100000	16322	1115	646	7	81910
% of the origin	100	16.32	1.11	0.65	0.01	81.91
% of the previous	-	16.32	1.33	0.78	0.01	
Words after	-	83678	82563	81917	81910	

Table 3.1. *Data preprocessing*

3.1 Hungarian Webcorpus

Hungarian Webcorpus is the largest Hungarian language corpus with over 1.48 billion words and it is available in its entirety under a permissive Open Content license [13]. Because it is collected from websites with a .hu domain, it contains not only Hungarian words but from many different languages too. However it has over a billion words, for the most of this study we will use only a smaller portion of it (100 000 words).

3.2 Cleaning

In the first stage of the data preprocessing I used the following cleaning methods:

All lowercase Setting the letters in lower case form.

Filtering special characters Deleting the following punctuation characters from the texts (the *python* function `string.punctuation` defines them environment dependent so it can be different elsewhere):

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

Filtering numbers Deleting the numbers from the texts.

After the data cleaning I deleted the multiple occurrences (mostly caused by the upper case of sentence starting words) and the empty data (caused by deleting standing numbers). The remaining data size was 83678 words.

3.3 Filtering

Some of the models I designed are not suitable for all the words in the cleaned data.

3.3.1 Non-standard hyphenation with character addition

There are non-standard hyphenations where a character addition occurs (see more in 1.2.1 section). The following table shows these additions in the Hungarian language (Table 3.2.). Since the models use character tagging that may interfere, I removed them from the database. Another way of solving the problem would be to only remove these additions. Note that in the final model if we try to predict a word with this type of non-standard hyphenation it may find the place of the hyphen but cannot solve the addition.

There were 1115 words with this type of hyphenation in the dataset. There are words like *asszonnyal* which contains multiple of non-standard hyphenation so the total amount was 1117 and I used number when I calculated the frequencies.

Before	After	Number	Frequency(%)
ccs	cs-cs	21	1.88
ggy	gy-gy	21	1.88
lly	ly-ly	37	3.31
nny	ny-ny	210	18.80
ssz	sz-sz	813	72.78
tty	ty-ty	13	1.16
zzs	zs-zs	2	0.02

Table 3.2. *Hungarian non-standard hyphenation*

3.3.2 Special characters

My models use an One-hot encoding for the Hungarian character set, which is:

a á b c d e é f g h i í j k l m n o ó ö ő p q r s t u ú ü ű v w x y z

In the previous step, we cleaned the punctuation characters from the data, but there are remaining letters like ô. With filtering these words we lost another 646 data row. Table 3.3. illustrates frequency of these special characters among the 646 words. Note that one word could contain multiple non-Hungarian characters.

Char	No.	Frequency(%)	Char	No.	Frequency(%)
ô	319	49.38	ń	13	2.01
ä	61	9.44	§	13	2.01
ă	37	5.73	â	9	1.39
ĺ	29	4.49	ľ	9	1.39
ď	28	4.33	ż	9	1.39
î	26	4.02	a̧	8	1.23
ß	25	3.87	a̧	8	1.23
ë	18	2.79	ř	8	1.23
ȩ	18	2.79	u̇	8	1.23
ň	17	2.63	ž	8	1.23
ě	15	2.32	š	6	0.93
č	15	2.32	ć	6	0.93
ç	14	2.17	ý	5	0.77
ŕ	14	2.17	Other	69	10.68

Table 3.3. *Non-Hungarian characters in the data set*

3.4 Long word cut and padding

The CNN and LSTM models use fixed size words as input so I had to pad the shorter words with padding and skip the longer words. The FFNN model needs pre- and post-word padding too so I defined them as ˆ and $ respectively. $ was chosen as the length standardizing filling character and I filled the words' end with it.

The fixed size of the words was 30. There were only 7 words out of the range and all of them were gibberish (Table 3.4.).

Word	Possible origin
mailtomaiserszechenyinkzsasulinethu	mailto hyperlink
httpdelphiszechenyinkzsasulinethu	HTTP hyperlink
ftpftpszechenyinkzsasulinethudelphi	FTP hyperlink
httpwwwegyismertszerverhuképgif	HTTP hyperlink
orgapachecatalinacorestandardpipeline	?
standardpipelinevalvecontextinvokenext	?
httpwwwsomeunknownplacenetmypicturegif	HTTP hyperlink

Table 3.4. *Words out of the fixed length*

Chapter 4

Machine Learning based Hungarian hyphenation algorithms

4.1 Character classification

These models are based on character classification. The letters in the words are inserted into groups according to their position between the hyphens. I defined two classification patterns but later I only used the simpler one.

BM The BM classification uses two classes:

- B: In the beginning of the syllables.
- M: Every other letter.

The word *leopárd* (leopard) hyphenated as `le-o-párd` and tagged as `BMBBMMM`.

BMES

- B: In the beginning of the syllables.
- M: The middle of the syllables. Hyphens are neither before nor after this character.
- E: Ending character, a hyphen is placed after this.
- S: Single character: it is between hyphens.

The `le-o-párd` tagged as: `BESBMME`

4.2 Feedforward neural network

In the feedforward neural network we want to decide to which class a specific character belongs. To do it we use the character and its surroundings coded in one-hot as the input layer of a fully connected network. The output is the class.

The network is summarized in Figure 4.1. The blue boxes are the letters used to define the tag of L, in a five window length method it represents the five letter ^^*LEO* in one-hot encoded way, then flatten them to make input for the FFNN network. The outputs of the network are two numbers: the probability of B and M. The process chooses the larger one, B. The orange boxes are for the letter E and the green ones are for the D.

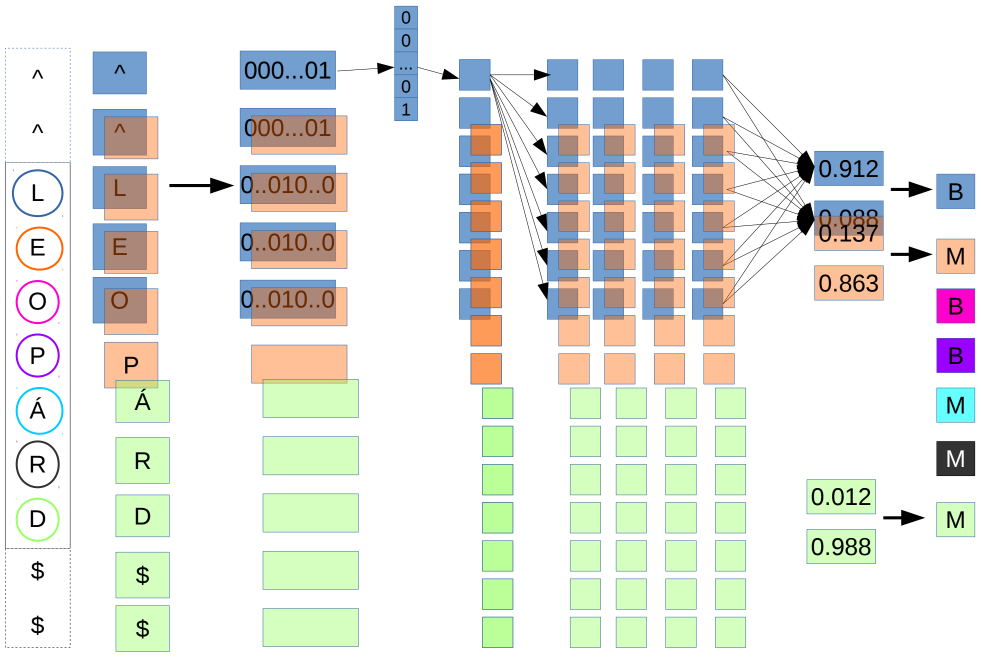

Figure 4.1. *Feedforward neural network*

4.2.1 Data preparation

In the following I show the methods of creating the training data by using an example. This example is the 5-length windowed BM classified preparation of the *o* character of the *leopárd* word. Figure 4.2. summarizes the data preparation.

Learning from the surroundings (Windowing)

The 5-length window means that we use 5 characters around the *o* letter to classify it. The value 5 means that we will use 2 letters before the actual character, itself and 2 letters after it. Implementing it on *leopárd* the windows are from *^^l eo* to *ár d $$* where the window of letter *o* is: *le o pá*.

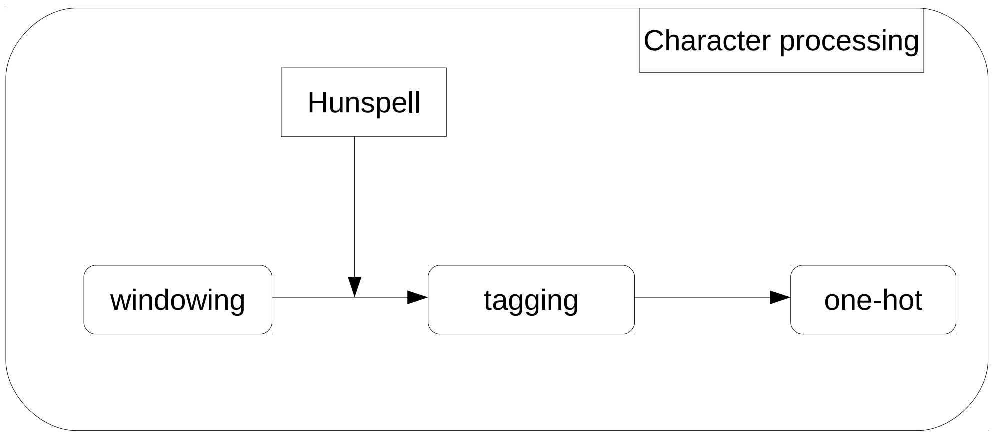

Figure 4.2. *Character preprocessing*

Note that if the windows are out of the word's range we add ^ character at the beginning of the word and $ character at the end of it. We can see the paddings in Figure 4.1.

Labelling

To determine the labels of the training data I used the Hunspell's hyphenation algorithm. In this example it is the **BM** labelling, so the class of the letter *o* is B.

One-hot encoding

So far we have worked with characters but Keras trains on numbers so we have to represent the words as numbers. An effective way of doing so is the one-hot encoding. It means that we define a 37 length array for each letter in the word: each element of the array means one letter of the Hungarian characters (35 letters) or the beginning or ending characters. One-hot means that there is only one 1 in the array, the others are 0. For the letter *a* it is the first one and the character *o* it is the 18th.

Thus we have 5 characters in a window, we have $(5, 37)$ shaped two-dimensional array.

The same encoding is used on the labels, B is the 0 and M is the 1.

Flattening

The final step before the training is flattening. From the two-dimensional array we create a reshaped one-dimensional. This means simply putting the characters after each other. So if we had the $(5, 37)$ array, now we have the $5 \cdot 37 = 185$ long 0,1 sequence with only 5 ones in it. And this is the training data.

In summary, from the *leopá* window we got a 185 long $0, 1$ as the training input and $[1, 0]$ as the training output.

4.2.2 The model

The neural network is a fully connected feedforward neural network. In the hyperparameter optimization along the window length I changed the number of hidden layers and the units in each hidden layer. The last layer has a softmax activation to approximate the values as probabilities. Figure 4.3. is the Keras visualization of the layers.

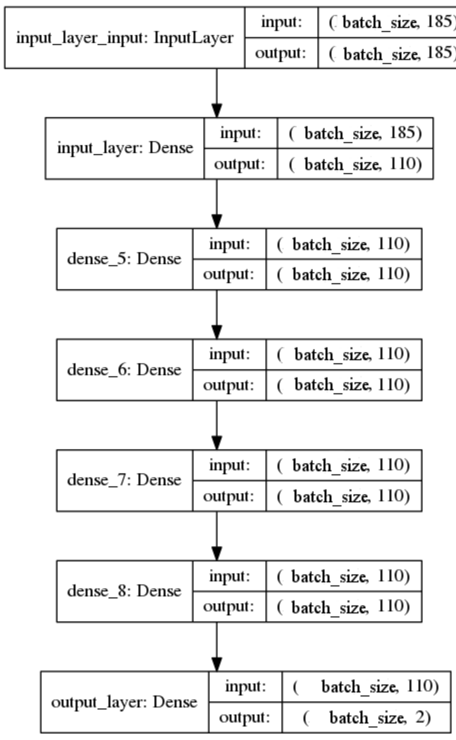

Figure 4.3. *The FFNN model*

In the training process I used *sigmoid* activation function and *adam* optimizer [11].

4.3 Train-, validation-, test data

In machine learning, it is a well-known paradigm to separate the training, validation and test data. While the neural network fits on a training data, its performance during the training is measured on a different, validation data. If we use early stopping, validation data is used to halt the training when it stopped improving on the validation data. The third part is the test data. It has not been used in the training method thus it provides an independent evaluation data. In the trainings I used the common 70-20-10 separation rate.

Figure 4.4. shows the following 3 methods of splitting the data words into 3 separate sets.

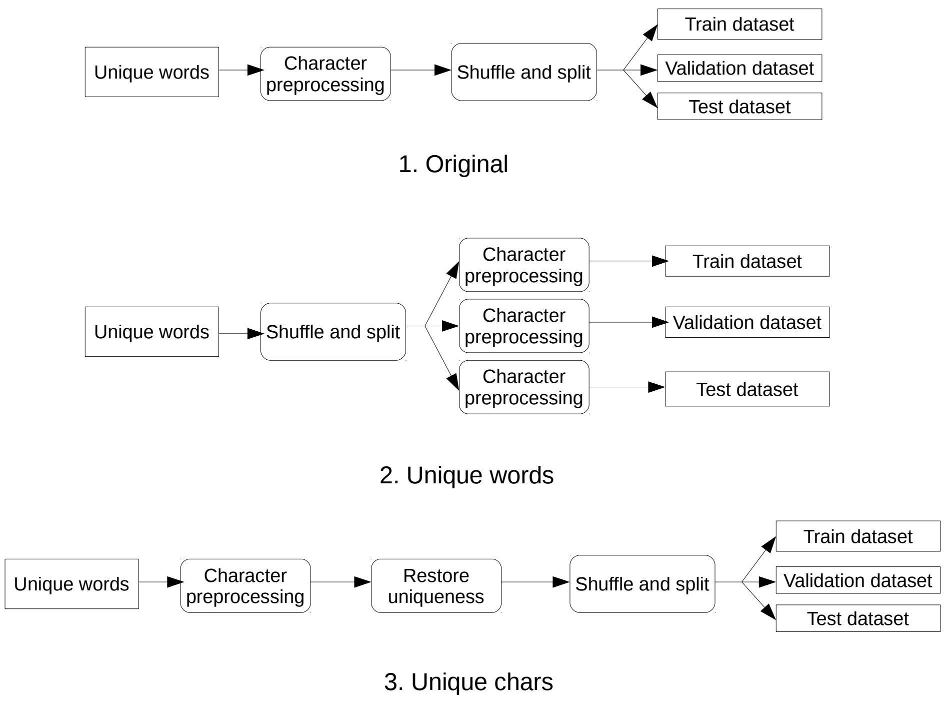

Figure 4.4. *The 3 methods of splitting train-validation-test data.*

4.3.1 Original method

In the first method I followed these steps: from the unique words of the dataset with the previously described way I created the train-ready data. Then I shuffled it and split into 3 parts. However it has two major problems.

4.3.2 Unique words

Firstly, when we shuffle the windows *leopá* and *eopár* they can be in different sets. The second method prevents it. Here I separate the words first, so every word is in one of the 3 datasets. So there are unique and separated words in the sets. But later we would create windows thus there are windows appearing multiple times, and in different datasets.

4.3.3 Unique characters

The third method focuses on the uniqueness of the {training window, label} pairs. However if a training window has an example when it is labelled with B and one with M and one of it went to the training set while the other was in the test set, it always makes an error at the evaluation.

4.3.4 Conclusion

While the third method is the closest to an ideal setting, in a real situation it is quite unlikely to get an entirely new {training window, label} pair. However it is common to get a new word.

When we compare the FFNN network with CNN and LSTM ones, we separate the words first because the other two model uses it.

4.4 Convolutional neural network

4.4.1 Data preparation

For the CNN and LSTM networks the data preparation steps are the same. First, using the Hunspell hyphenation to define the labels, then filling the words to a fixed size with padding (the filling label is **M**) and finally using the one-hot encoding method (Figure 4.5.).

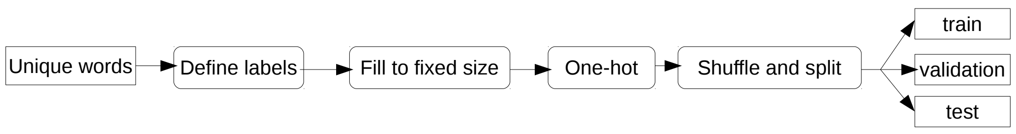

Figure 4.5. *CNN and LSTM data preparation*

4.4.2 The model

The model has two parts. The first one is the convolutional network and the second is a feedforward network. The convolutional layers have a stride one so it convolves all the characters. I optimized the kernel size (it is the same dimension as the window-length in the FFNN model), filter and hidden layer numbers. I used the Keras' built-in padding to prevent problems at the endings of the word. The model currently uses ReLU activation functions.

The feedforward part is a softmax layer for every character to get back the BM one-hot probabilities.

Figure 4.6. summarizes the CNN model.

4.5 Long short-term memory

The LSTM network uses the same inputs as the CNN. I used biLSTM and optimized the unit and hidden layer numbers. At the output of the LSTM I inserted a feedforward network just as in the CNN one. The model summary of the LSTM network is in Figure 4.7.

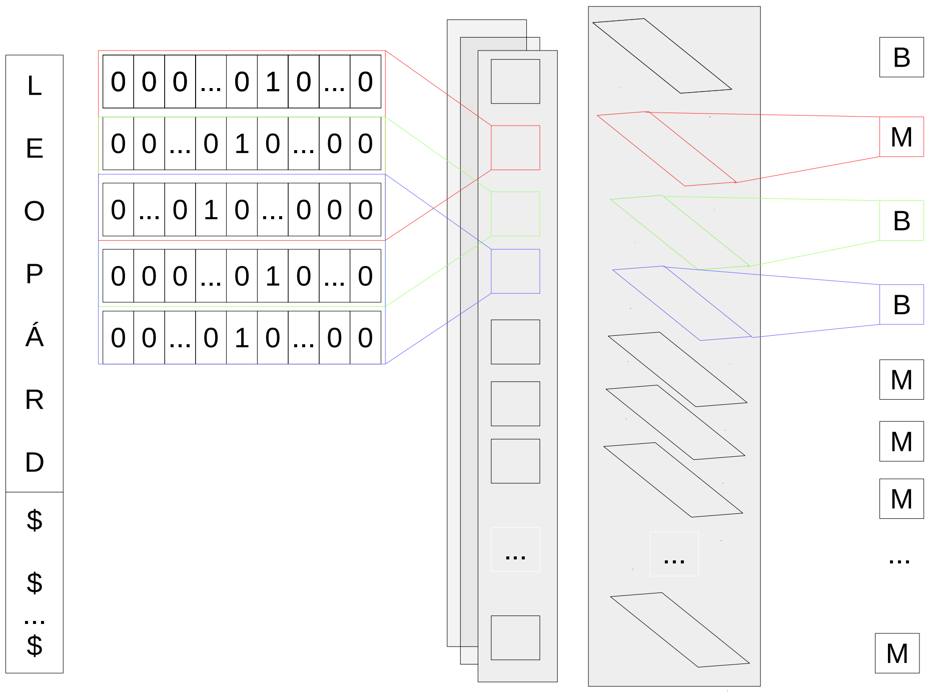

Figure 4.6. *CNN model summary*

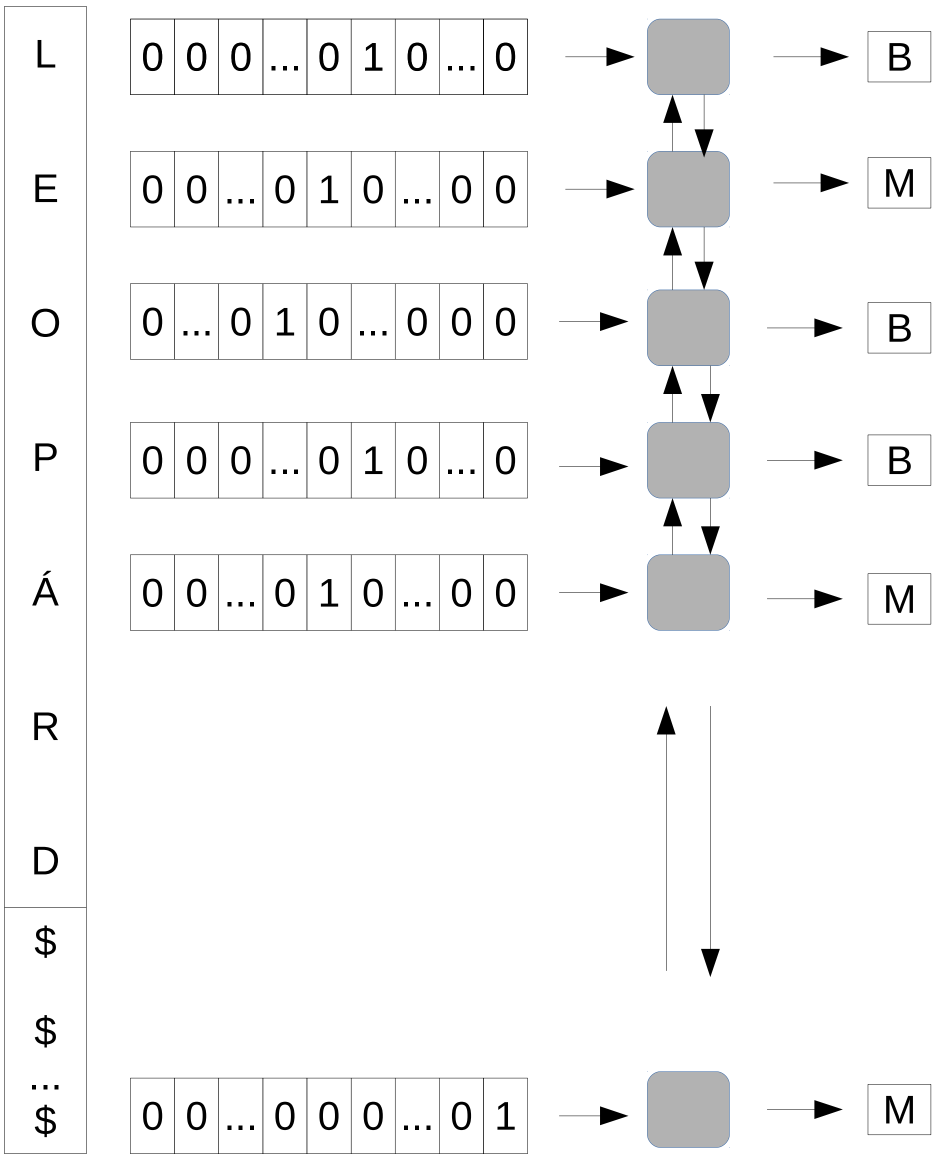

Figure 4.7. *Long Short-Term Memory network*

Chapter 5

Results

5.1 Evaluation values

Val_loss is the validation loss of the Keras model.

True positive sample is what has a **B** label in both the prediction of the model and the target (the Hunspell's hyphenation).

True negative sample has a **M** label as prediction and as target too.

False positive sample is predicted as **B** by the model but expected **M** by the dataset.

False negative is the opposite of false positive: the prediction is **M** and the target is **B**.

Precision is the rate of correct values among all positive results $P = t_p/(t_p + f_p)$.

Recall is the rate of predicted **B** values compared to all that should have been **B**, $R = t_p/(t_p + f_n)$.

F-Score is the harmonic mean of precision and recall: $F = 2\frac{P \cdot R}{P+R}$

Word accuracy is the ratio of correctly labelled words among the test data.

5.2 Feedforward neural network

I optimized around the window length, the layer number and the hidden unit numbers(Table 5.1.). The window-length was tested between three and eleven character. The layer number and hidden unit numbers were only optimized for five and seven window-length and between 3 and 7 for the layer number and 80 to 130 for hidden units.

Window length	Num_layer	Num_hidden	Epochs	Val_loss
3	5	60	303	0.0916
3	5	70	303	0.0916
3	5	80	229	0.0916
3	5	90	252	0.0914
3	5	100	196	0.0928
3	5	110	249	0.0913
5	5	60	222	0.0406
5	5	70	202	0.0397
5	5	80	222	0.0421
5	5	90	214	0.0399
5	5	100	198	0.0395
5	5	110	190	0.0400
7	5	60	133	0.0385
7	5	70	131	0.0425
7	5	80	147	0.0443
7	5	90	149	0.0387
7	5	100	137	0.0381
7	5	110	136	0.0393
9	5	60	121	0.0390
9	5	70	119	0.0432
9	5	80	125	0.0413
9	5	90	118	0.0393
9	5	100	116	0.0383
9	5	110	126	0.0397
11	5	60	112	0.0454
11	5	70	111	0.0408
11	5	80	119	0.0457
11	5	90	99	0.0436
11	5	100	103	0.0401
11	5	110	98	0.0412

Table 5.1. *FFNN window length optimization*

5.3 Convolutional neural network

First I optimized the numbers of hidden units and the kernel size. After a few run, the optimum of the kernel size showed to be around 10 character so I optimized the layer number and the numbers of hidden units only for 6-10 kernel size it (Table 5.2.).

Num_layers	Num_hidden	Kernel size	F-score	Word accuracy
1	512	6	98.63%	93.30%
1	512	8	98.77%	94.07%
1	512	10	98.71%	93.70%
1	1024	6	98.78%	94.10%
1	1024	8	98.79%	94.08%
1	1024	10	98.86%	94.52%
1	2048	6	98.75%	93.85%
1	2048	8	98.85%	94.42%
1	2048	10	98.84%	94.40%
2	512	6	98.92%	94.68%
2	512	8	98.84%	94.41%
2	512	10	98.79%	94.23%
2	1024	6	98.96%	94.96%
2	1024	8	99.01%	95.28%
2	1024	10	98.90%	94.92%
2	2048	6	99.03%	95.25%
2	2048	8	98.99%	95.14%
2	2048	10	98.92%	94.91%
3	512	6	98.77%	94.10%
3	512	8	98.68%	93.71%
3	512	10	98.39%	92.14%
3	1024	6	98.88%	94.54%
3	1024	8	98.81%	94.24%
3	1024	10	98.64%	93.82%
3	2048	6	98.92%	94.64%
3	2048	8	98.76%	94.18%

Table 5.2. *CNN hyper-parameter optimization*

5.4 Long short-term memory

The LSTM network was the most sensitive to the change of the LSTM layer number (Table 5.3.).

5.5 The 3 model

Table 5.4. illustrates a rerun of the models with highest F-score. The hyper-parameters: **FFNN**: 3 layers of 130 units and a window length of 7. **CNN**: 2 CNN layers with 1024 hidden units and a kernel size of 8. **LSTM**: 2 layer of 128 units.

Num_layers	Num_hidden	Precision	Recall	F-score	Word accuracy
1	8	50.33%	75.61%	60.43%	0.72%
1	16	50.32%	75.67%	60.45%	0.72%
1	32	50.33%	75.64%	60.44%	0.72%
1	64	50.33%	75.63%	60.44%	0.72%
1	128	50.34%	75.64%	60.45%	0.72%
1	256	50.33%	75.64%	60.44%	0.72%
2	8	42.55%	75.41%	54.41%	0.00%
2	16	97.09%	99.05%	98.06%	91.73%
2	32	95.95%	98.59%	97.25%	88.71%
2	64	97.52%	98.93%	98.22%	92.63%
2	128	97.99%	99.18%	98.58%	93.90%
2	256	97.77%	99.22%	98.49%	93.37%
3	8	54.23%	99.29%	70.15%	11.71%
3	16	64.65%	97.48%	77.74%	29.34%
3	32	99.94%	31.43%	47.82%	7.54%
3	64	54.95%	99.74%	70.86%	11.70%
3	128	60.53%	98.87%	75.09%	11.57%
3	256	55.52%	98.60%	71.04%	12.62%
3	512	43.75%	99.78%	60.83%	0.59%

Table 5.3. *LSTM hyper-parameter optimization*

Model	Kernel	Layers	Hidden	Epochs	Precision	Recall	F-score	Word accuracy
FFNN	7	3	150	103	98.17%	99.11%	98.64%	93.57%
CNN	8	2	1024	12	98.37%	99.27%	98.81%	94.45%
LSTM	-	2	128	66	97.68%	99.16%	98.42%	93.13%

Table 5.4. *The three model that performed the best*

Chapter 6

Evaluation

In the 5.5. section I showed 3 models trained in the same data. The following error analysis is based on those models. Figure 6.1. represents the models' overall performance on the 8191 test words. As we can see, if we uses all three models with the majority rule, 94.3% of the words can be hyphenated correctly (however the CNN network itself achieves a better performance with 94.45%).

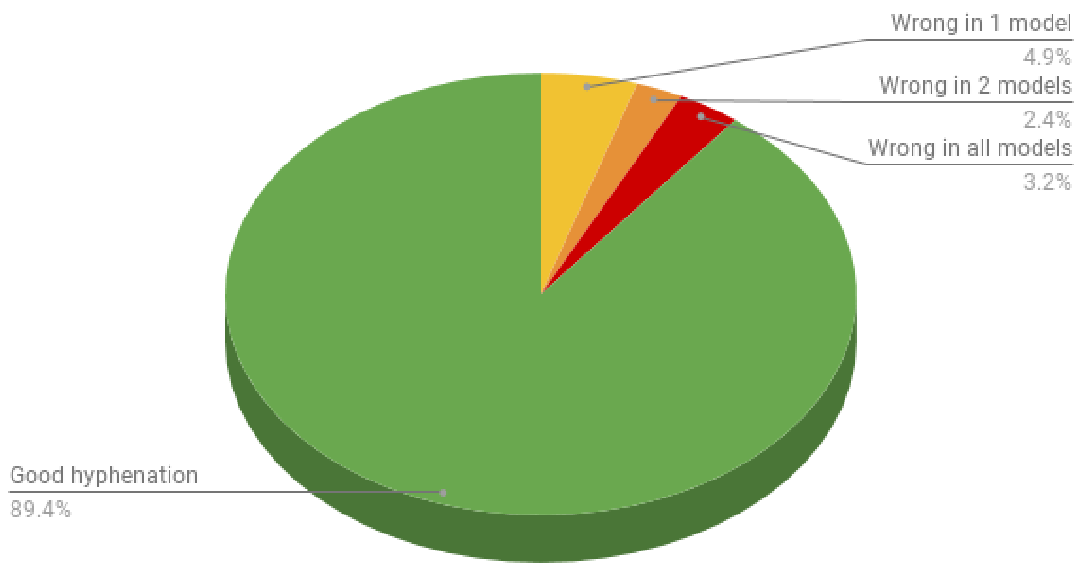

Figure 6.1. *Model performance*

6.1 Error categorization

After the rerun, I manually tested the errors and grouped them into categories. These categories are:

Non-hungarian word is which I recognized as a word but not Hungarian.

Compound word

Non-hyphenated part has a hyphen missing because of Hunspell's typesetting goals.

Wrong target Hunspell misses the hyphenation.

Not (a) word are mostly mistyped Hungarian words like *elol.*

Other words not filling the above categories.

6.1. Table shows category distribution among all the words (perfectly predicted words as well as missed ones). Note that one word can be in multiple categories.

Category	Distribution	Example (Hunspell's hyphenation)
Non-Hungarian word	11	obsta-c-les
Compound word	21	ak-ció-film
Non-hyphenated part	8	ak-ció-film
Wrong target	1	diszk-ri-mi-na-tív
Not word	4	el-er-ni
Others	59	

Table 6.1. *Error categories among 100 randomly chosen words*

6.2 Model performance

I collected all the errors of the three models and manually inserted 200 words into the error categories. Figure 6.2. shows the category distribution of the errors for each model and among the errors.

As we can see, over half of the errors are caused by using a Hungarian hyphenation algorithm on a non-Hungarian word. While Table 6.1. says that only 11% of the words are non-Hungarian, half of the errors can be tracked back to them. If we compare it with the data of Table 5.4. where we saw that only 6% of the words are hyphenated badly we can conclude that half of this 6% is caused by non-Hungarian words so 3% of every word is miss-hyphenated non-Hungarian word. From this point of view the 11% contribution of all non-Hungarian words says that 27% of all non-Hungarian words in the test set are hyphenated wrongly.

In the next figure (6.3.), I visualized the three models' performance in each categories. So in this figure we can see the percentage of missed words by the models, where the 100% is the amount of wrong word in the category. The last column shows the 3 models' distribution of hyphenation misses. As we can see, the Hunspell's typesetting error (*Non-hyphenated part*) had the worst effect on the LSTM network.

Note that there was only one word in the *wrong target* category.

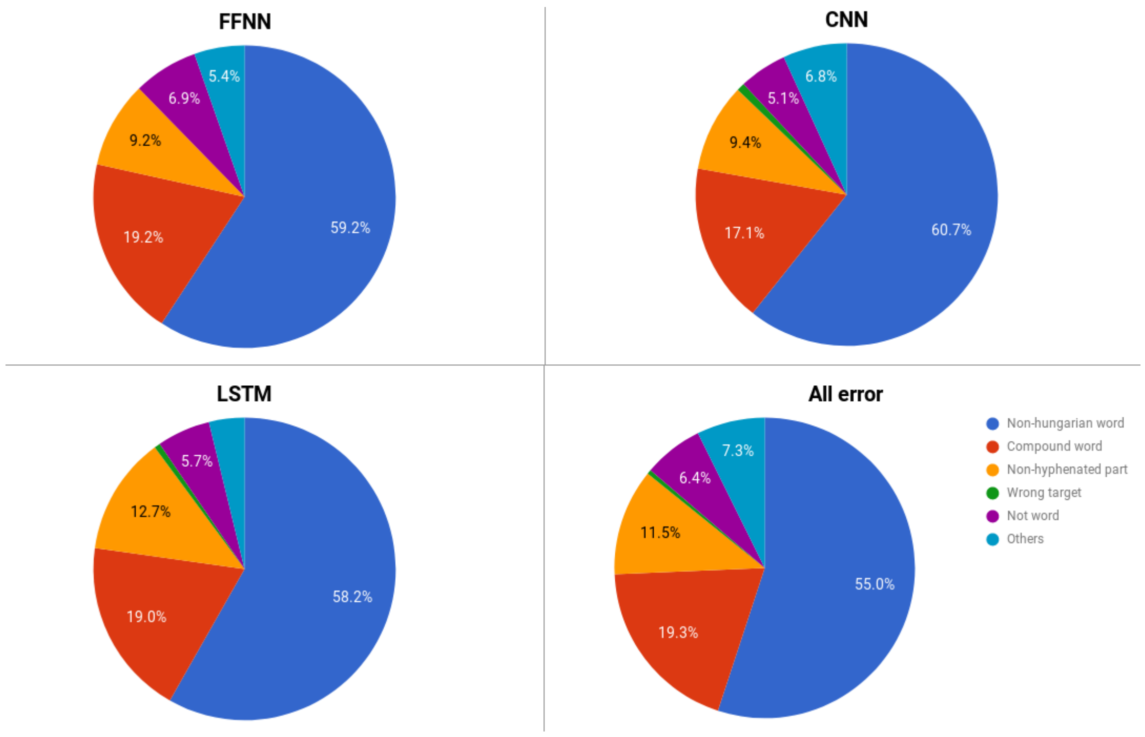

Figure 6.2. *Missed hyphenations by error categories*

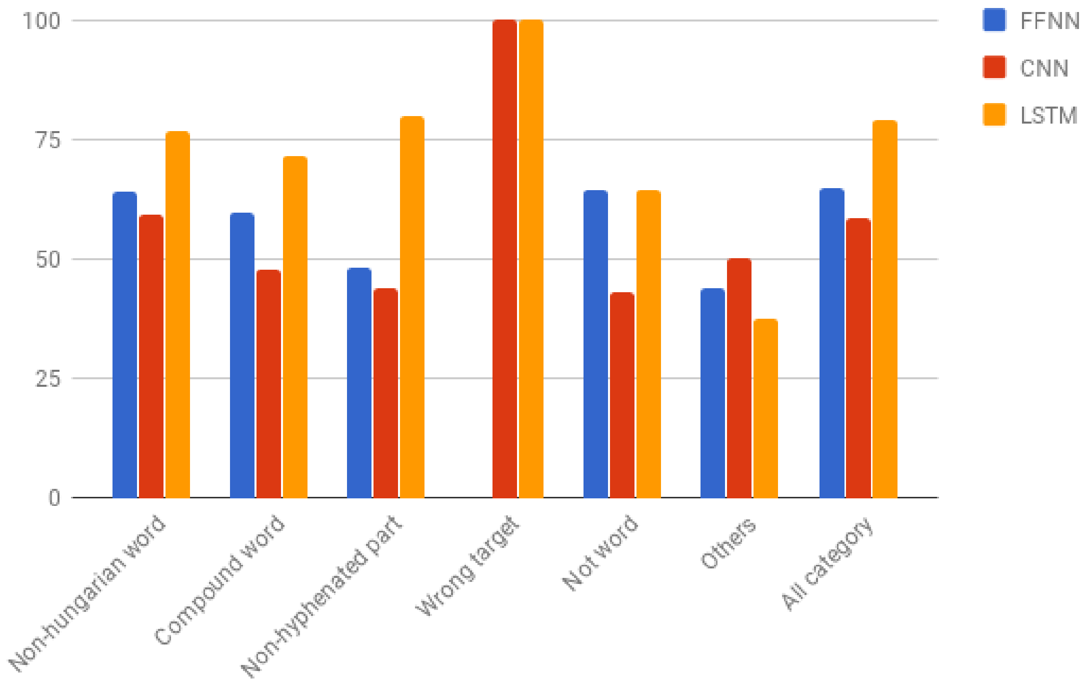

Figure 6.3. *Model performance by categories*

6.3 Further research

There are many ways to improve the hyphenation algorithm. In the future I am intent to do two things.

6.3.1 Multilingual hyphenation algorithm

As we saw, the hyphenation algorithm tends to make a mistake in non-Hungarian words. In Hunspell, the writer has to chose the language of the hyphenation but I believe that the deep learning based algorithm can achieve it in the same network with a mixed dataset of multilingual words.

6.3.2 Non-standard hyphenation

The algorithm described above cannot handle the non-standard hyphenations because it is a character tagging algorithm but I think that other models can solve this problem too.

Conclusion

In this thesis, a deep learning based hyphenation algorithm was introduced. The thesis presents the currently used hyphenation algorithms. Three neural network models were described: the Feedforward Neural Network (FFNN), the Convolutional Neural Network (CNN) and the Long Short-Term Memory network (LSTM).

The paper shows a method for words to be used in the networks and implements the three models for the hyphenation problem. The networks described in the paper achieve around 98% precision, over 99% recall and about 98.5% F-score in terms of inserting hyphens between characters in Hungarian words. Hyphenation of whole Hungarian words is 95% accurate.

The thesis evaluates the errors of the networks by categorizing them into error groups and adumbrate a possible way of improving the accuracy with multi-lingual hyphenation.

List of Figures

List of Tables

Bibliography

[1] Magyar Tudományos Akadémia and Maďarsko Budapešt'. *A magyar helyesírás szabályai.* Akadémiai kiadó, 1959.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[3] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[4] Scott E Fahlman. An empirical study of learning speed in back-propagation networks. 1988.

[5] Bernd Fritzke and Christof Nasahl. A neural network that learns to do hyphenation. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 960–vol. IEEE, 1991.

[6] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[7] Jacques Hadamard. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*, volume 33. Imprimerie nationale, 1908.

[8] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

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

[10] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016.

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

[12] Donald Ervin Knuth. *TEX and METAFONT: New directions in typesetting.* American Mathematical Society, 1979.

[13] András Kornai, Péter Halácsy, Viktor Nagy, Csaba Oravecz, Viktor Trón, and Dániel Varga. Web-based frequency dictionaries for medium density languages. In *Proceedings of the 2nd International Workshop on Web as Corpus*, pages 1–8. Association for Computational Linguistics, 2006.

[14] Terje Kristensen. A neural network approach to hyphenating norwegian. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 2, pages 148–153. IEEE, 2000.

[15] Franklin Mark Liang. *Word hyphenation by computer.* Department of Computer Science, Stanford University, 1983.

[16] Márton Miháltz, Péter Hussami, Zsófia Ludányi, Iván Mittelholtz, Ágoston Nagy, Csaba Oravecz, Tibor Pintér, and Dávid Takács. Helyesírás. hu. 2013.

[17] László Németh. Automatic non-standard hyphenation in openoffice. org. *COMMUNICATIONS OF THE TEX USERS GROUP TUGBOAT EDITOR BARBARA BEETON PROCEEDINGS EDITOR KARL BERRY*, page 32, 2006.

[18] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[19] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[20] Petr Sojka et al. Notes on compound word hyphenation in tex. *TUGboat*, 16(3):290–296, 1995.