Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Diacritics Restoration for Slovak Texts Using Deep Neural Networks

Master thesis

2018
Marek Šuppa

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Diacritics Restoration for Slovak Texts Using Deep Neural Networks

Master thesis

Study program:	Informatics
Field of study:	2508 Informatics
Department:	Department of Informatics
Thesis advisor:	prof. Ing. Igor Farkaš, Dr.

Bratislava, 2018
Marek Šuppa

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Marek Šuppa
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Diacritics Restoration for Slovak Texts Using Deep Neural Networks

Annotation: Diacritics restoration is a natural language processing problem, whose importance is increasing, given the amount of text produced without diacritics on venues such as social networks. Recent successes of Deep Recurrent Neural Networks being used for a related task in Arabic suggest that these techniques might also be useful in the context of Slovak language. Validation of this hypothesis is the aim of this project.

Aim:
1. Examine the relevant literature on Natural Language Processing with special focus on applicable Deep Learning techniques, as well as the state of the art methods for diacritics restoration.
2. Assess the viability of a proposed model for diacritics restoration based on Deep Neural Networks by implementing and evaluating it on a dataset based on Slovak Wikipedia.

Literature:
Belinkov Y., Glass J. (2015). Arabic diacritization with recurrent neural networks. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.
Goldberg Y. (2015). A primer on neural network models for natural language processing. arXiv preprint:1510.00726.

Supervisor:	prof. Ing. Igor Farkaš, Dr.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	15.12.2016
Approved:	19.12.2016

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

I would like to thank everyone who might have in any way contributed to creation of this thesis, most notably prof. Farkaš for giving me a chance to experience what life in academia is really about in the end.

Abstract

Diacritics restoration is a natural language processing problem, whose importance is increasing, given the amount of text produced without diacritics on venues such as social networks. Recent successes of Deep Recurrent Neural Networks being used for a related task in Arabic suggest that these techniques might also be useful in the context of Slovak language. Validation of this hypothesis is the aim of this project.

Keywords: deep learning, natural language processing, diacritics restoration

Abstrakt

Reštaurácia diakritiky je problém v oblasti spracovania prirodzeného jazyka, ktorého doležitost rastie, vzhľadom na množstvo produkovaného textu bez diakritiky, napríklad na sociálnych sieťach. Nedávne úspechy hlbokých neurónových sietí, ktoré boli použité na podobnú úlohu v Arabčine naznačujú, že tieto techniky by mohli byť použiteľné aj v kontexte Slovenčiny. Validácia tejto hypotézy je cieľom tohto projektu.

Kľúčové slová: hlboké učenie, spracovanie prirodzeného jazyka, reštaurácia diakritiky

Contents

Introduction **1**

1 Diacritics Restoration **3**
1.1 Word-based Diacritics Restoration 4
1.1.1 Baseline for word-based diacritics restoration 5
1.1.2 N-gram models 6
1.1.3 Bayesian classifiers 8
1.1.4 Other models 9
1.2 Grapheme-based Diacritics Restoration 10
1.3 Diacritics Restoration using Deep Neural Networks 12

2 Deep Neural Networks for NLP **14**
2.1 Neural Networks 14
2.1.1 Biological Neural Networks 14
2.1.2 Modeling Neural Networks 15
2.1.3 Neural Networks as Graphs of Neurons 20
2.1.4 Modeling Neural Networks Mathematically 21
2.1.5 A Note on Representation Power 24
2.2 Feature Representation 24
2.2.1 One Hot Representation 25
2.2.2 Dense Vector Representation 26
2.2.3 Embedding Layers 26
2.3 Recurrent Neural Networks 28
2.3.1 Defining Recurrent Neural Networks 28
2.3.2 Simple RNN 30
2.3.3 LSTM 31
2.3.4 GRU 35
2.3.5 Independent RNN 37
2.4 Training of Neural Networks 37
2.4.1 Loss Functions 37
2.4.2 Variants of Gradient Descent 38

2.4.3 Adaptive Learning Rate Algorithms 40
2.4.4 Training RNNs 43
2.4.5 Other considerations 45

3 DNNs for Diacritics Restoration **47**
3.1 Existing Deep Neural Network Architectures for Diacritics Restoration 47
3.2 Our Proposed Model 48
3.2.1 Encoder architecture 48
3.2.2 Decoder architecture 48
3.2.3 Implementation 50
3.3 Evaluation 50
3.3.1 Datasets 50
3.3.2 Preprocessing, Training and Testing Regimes 51
3.3.3 Evaluation Metrics 52
3.3.4 Results 53
3.3.5 Analysis 56

Conclusions **59**

List of Figures

1.1 Diacritics in European languages with Latin based alphabets. [MN02] . 4
1.2 restoration of diacritics for the word "mbũri" (goat in the G̃ikũyũ language). Note that every character is considered in its own context. [DPWDS07] 11
1.3 An illustration of the topology of the network used in [BG15] 13

2.1 An illustration of a biological neuron [LKJ15] 15
2.2 An illustration of a mathematically modelled neuron [LKJ15] 16
2.3 An illustration of the output of the sigmoid activation function (y axis). [LKJ15] 17
2.4 An illustration of the output of the tanh activation function (y axis). [LKJ15] 18
2.5 An illustration of the output of the ReLU activation function (y axis). [LKJ15] 19
2.6 An illustration of a three-layer neural network comprised of two fully connected and one output layer. Note that the number of layers does not include the input layer. [LKJ15] 21
2.7 An illustration of a three-layer neural network comprised of two fully connected and one output layer. The input layer neurons are colored in green, the output layer neurons in red and the hidden layers, onto which a sigmoid non-linearity is applied, are colored in purple. [Gol15] 23

2.8 An illustration of the comparison between different feature representations. In both cases the following information is encoded: *current word is "dog"; previous word is "the"; previous part-of-speech tag is 'DET'*. In case of sparse feature vector (a), each feature, as well as combination of features, is represented by a separate dimension. Feature values are binary and the dimensionality of such a vector is very high. In case of dense feature vector (b), the dimensionality is lower, as the resulting feature vector consists of concatenated embeddings of both words and part-of-speech tags and combinations of features are not explicitly encoded (except for specific positions of respective embedding vectors in the resulting feature vector). [Gol15] 25
2.9 An illustration of the RNN architecture. [Gol15] 28
2.10 An illustration of the "unrolled" RNN architecture. Note especially the parameters θ which are shared across different time steps. [Gol15] . . . 29
2.11 An illustration of the deep RNN architecture. [Gol15] 30
2.12 An illustration of the bidirectional RNN architecture applied on the sentence "the fox jumped .". Note that the illustration shows a single BiRNN layer with a separate RNN for both forward and backward directions of processing the input. [Gol15] 31
2.13 An illustration of the LSTM architecture in forward direction. Note that the red circles contain point-wise mathematical operations, yellow boxes contain non-linearities associated with Neural Network layers and black pathways signify the flow of information. [Ola15] 32
2.14 A visualization of the forget gate. [Ola15] 33
2.15 A visualization of the input gate. Note that the uppercase $\hat{C}$ in the picture is functionally the same as $\hat{\mathbf{c}}$ we used in our descriptions. [Ola15] 34
2.16 A visualization of the computation of the new memory state. Note that the uppercase $\hat{C}$ in the picture is functionally the same as $\hat{\mathbf{c}}$ we used in our descriptions. [Ola15] 34
2.17 A visualization of the computation of the new memory state. Note that the uppercase $\hat{C}$ in the picture is functionally the same as $\hat{\mathbf{c}}$ we used in our descriptions. [Ola15] 35
2.18 A visualization of the GRU architecture. . [Ola15] 36
2.19 A visualization of fluctuation during training using the SGD algorithm. [Com17] 39
2.20 A visualization of the concept of teacher forcing. During train time (left) the ground truth $\mathbf{y}^{(t-1)}$ is provided to the network as an input, whereas at test time (right), it receives its own past output $\mathbf{o}^{(t-1)}$ as input. [GBCB16] 44

3.1 An illustration of the topology of the network used in [BG15] 48
3.2 An illustration of the attention mechanism [BCB14]. 49
3.3 A visualization of the embedding layer learned as part of training the Diacritics Restoration model. 58

Introduction

Although it may not be obvious at the first sight, communicating via text is one of crucial parts of human life in modern society. Whether through SMS, instant messaging, social networks, email or combination thereof, we encode our thoughts in text-based messages, which are then consumed by the receiving party. Especially with the advent of always connected devices such as smartphones and tablets, this encoding happens quickly, often times in a rush, without an expectation for further corrections. It is therefore only natural that the process of encoding needs to be as efficient as possible.

While most European languages make use of diacritic marks (in fact as the authors of [KDDV17] observe, out of 36 European languages, English is the only one which does not contain diacritic marks), when it comes to efficient expressions of thoughts via textual messages as described in the previous paragraph, diacritics marks are often not used. Despite the fact that auto-correcting mechanisms alleviate the need for dealing with this problem, we can only hypothesize that this may be due to the widespread adoption on QWERTY/QUERTZ keyboards where typing diacritized characters poses an additional hurdle, and the fact that the receiving party is generally capable of extracting the encoded information even from text which contains only the ASCII equivalents[1] of characters with diacritic marks but is otherwise grammatically correct.

While the information extraction argument proposed in the previous paragraph may hold in case of personal communication, it certainly comes along as problematic when it comes to official communication: it is difficult to imagine that a document written in a language that makes use of diacritic marks would be considered serious if it did not contain a single diacritized character. Furthermore, most of the available Natural Language processing tools expect grammatically correct text as its input.

Thus, a need for an automatic solution for diacritics restoration arises. Sadly, to the best of our knowledge such system specifically for Slovak language does not exist yet. Given promising results that were achieved by applying Deep Learning techniques to this task, the aim of this thesis therefore is to

1. review related work with regards to Diacritics Restoration and Deep Neural Networks

[1] These are also sometimes referred to as "latinized" equivalents.

2. review recently proposed models for Diacritics Restoration that make use of Deep Learning

3. propose a Diacritics Restoration model for Slovak texts

4. evaluate the proposed model on texts from Slovak Wikipedia

This thesis is structured as follows. The topic of Diacritics Restoration is first discussed in Chapter 1, along with relevant related work in the area. In Chapter 2 we present an overview of Deep Learning methods and concepts with special focus on Natural Language Processing applications. This chapter provides the necessary background for the next one, in which we describe the relevant diacritics restoration models based on Deep Learning, propose our own alternative, evaluate it on two datasets comprised of Slovak texts and analyze its performance. The final chapter summarizes our conclusions, discusses possible shortcomings of our proposed solution as well as the used methodology, while providing an outlook for future work as well.

Chapter 1

Diacritics Restoration

Diacritics Restoration (sometimes also referred to as "diacritization", "automatic diacritization", "unicodification" or "deasciification") is usually defined as *"the task of inserting missing diacritics in text produced in languages that have diacritically marked characters in their orthography, but the diacritics are replaced with their corresponding Latinized grapheme"*[AH16]. This Latinization can be caused by various reasons. In most cases, they are either technical (such as for instance the lack of a specific keyboard and thus difficulty in producing correctly diacritized characters[CK11], technical requirements of transport protocols which are not able to work correctly with characters outside of the 8-bit ASCII character set, or difficulty in digitalizing resource-scarce languages by scanning pages and extracting characters from resulting images using a process generally called "Optical Character Recognition" (OCR) [DPWDS07]) or sociological (for example when the speed of transmitting a message is much more important than their grammatical correctness or aesthetic value, such as in the case of social media [üAE14]). Especially in the sociological case it can be easily seen that while humans are capable of interpreting latinized text and restore diacritics in real time while reading, the stored text that constituted the transmitted message (and which will most probably be used as input to other Natural Language Processing systems) misses a lot of morphological, lexical and phonological information [DPWDS07] [AH16]. Due to its indisputable importance in many real world applications of language technologies, Diacritics Restoration is an essential component of bigger Natural Language Processing systems whose end goal can range from Information Retrieval through Machine Translation to Corpora Acquisition [MN02].

Diacritics Restoration is generally considered to be a well studied problem and to some extend (especially in cases of specific languages for which appropriate lexicons exist) one that is already solved. It is not uncommon for published work to report accuracies that well exceed 90% [üAE14] [CK11].

As the authors in [AFNR12] note, however, while the diacritization problem can

Language	Diacritics	Language	Diacritics
Albanian	ç ë	Italian	à é è í ì ï ó ò ú ù
Basque	ñ ü	Lower Sorbian	ć č ě ł ń ŕ ś š ź ž
Breton	â ê ñ ù ü	Maltese	ċ ġ ħ ż
Catalan	à ç è é í ï l· ò ó ú ü	Norwegian	å æ ø
Czech	á č d' é ě í ň ó ř š t' ú ú ý ž	Polish	a˛ ć e˛ ł ń ó ś ź ż
Danish	å æ ø	Portuguese	â ã ç ê ó ô õ ü
Dutch	á à â ä é è ê ë i í ì î ï ó ò ô ö ú ù û ü	Romanian	â ă î ş ţ
English	none	Sami	á ï č d- ń n¸ š t- ž
Estonian	ä č õ ö š ü ž	Serbo-Croatian	ć č d- š ž
Faroese	á æ d- í ó øú ý	Slovak	á ä č d' é í Í ñ ó ô ŕ š t' ú ý ž
Finnish	ä å ö š ž	Slovene	č š ž
French	à â æ ç è é ê ë î ï ô œ ù û ÿ	Spanish	á é í ó ú ü ñ
Gaelic	á é í ó ú	Swedish	ä å ö
German	ä ö ü ß	Turkish	ç ğ i ı ö ş ü
Hungarian	á é í ó ö ő ú ü ű	Upper Sorbian	ć č ě ł ń ó ř š ž
Icelandic	á æ ð é í ó ö ú ý þ	Welsh	â ê î ô û ŵ ŷ

Figure 1.1: Diacritics in European languages with Latin based alphabets. [MN02]

be trivially solved when there is only one correct possibility in a dictionary for a given word, there can be other more interesting cases in which the error gives rise to some ambiguity (that is, for instance, that there are two correct sentences with different meanings). In such case, it is difficult to solve this problem automatically, since a non-trivial level of understanding of the composition of language and its inner-workings (which seems to be a natural attribute of most humans) is required. It this therefore obvious that models that provide very good results at tasks like language modeling are then directly usable for this task.

Number of all words	1208949
Number of unique words	899702
Number of all 'clean' words	856286
Words without alternations	515245
LexDif score	1.05

Table 1.1: A sample of the statistics report for a subset of Slovak Wikipedia dump. It shows that since the number of words with alternations is considerably big, the problem of Diacritics Restoration for Slovak language is far from trivial.

1.1 Word-based Diacritics Restoration

As noted above, most of the published work on diacritics restoration considers a word to be an atomic unit of information which is then processed further. One of the first works in this area (which already reported around 90% accuracy with a solid dictionary) [Yar99] made use of this concept and utilized both N-gram models, Bayesian classifiers and also a combination of both in the form of decision lists. Since then, many other

methods have been proposed and successfully tested, most of which rely on dictionaries and select among ambiguous words based on the context provided by other words that surround them.

In the following subsections we introduce some of these methods along with the results their respective authors report in the published works in order for the reader to better asses their viability and usefulness.

1.1.1 Baseline for word-based diacritics restoration

As any other class of tasks that rely on statistical inference, in the case of word-based diacritics restoration it is necessary to establish a simple baseline which is though to be a standard every new (and presumably better, at least in some sense) method will be compared against. The comparison with this baseline also allows one to comment on the size of the improvement which was introduced by any new model.

An investigation in literature shows that the majority of tokens (or words in this context) that are considered in the case of word-based diacritics restoration exhibit only one variation in accented vs. non-accented version of the token. In other words, in most cases there is no diacritics to be recovered. Moreover, when there is ambiguity among restorable words, one of them is normally used more often and is thus though to be dominant. This not only holds for European languages like Spanish and French [Yar99], but also for languages like Māori[CK11].

Thus, the baseline algorithm for word-based diacritics restoration is usually formulated as follows:

> Given a dictionary D, select the most frequently occurring variant of word w as the diacritized word.

Despite its simplicity, this baseline can achieve very solid results, when the metric of overall accuracy is used. In [Yar99] the authors report 97,6% mean accuracy for French and 98,7% mean accuracy for Spanish. Similarly, in [CK11] the authors report 97,11% accuracy for Māori. It needs to be noted, however, that while the accuracy of these models may seem high, they still make errors. And when they do, it is exactly in the ambiguous cases which should be considered most important by a model that is trying to restore diacritics.

Furthermore, the note above suggests, that plain accuracy might not be the best metric of assessing the quality of a diacritics restoration model in the word-based context. Many other metrics are used as a result, most notably the diacritics restoration error (error in words for which some diacritics was to be restored) and others, which will be discussed in more detail in the following sections.

1.1.2 N-gram models

The concept of a N-gram model is a very simple, yet quite powerful one. All one needs to do is to recall the counts of each N words that were present in the training dataset. At prediction time the word with the highest stored frequency given the words that surround it will be chosen.

While there are a few technical details to be considered (such as the predicted word, which can be located at different places: in between, at the end or even in front of the words that constitute its context), this model is very straightforward to implement and can lead to very good results, especially with well polished corpora which it can be trained on. In the most general sense, the model is trying to model the probability $P(w|c)$ where w is the word currently considered and c is the context, which can be defined in various ways ad described above. While the frequentest assumption described above might not always hold, in practice it provides an acceptable approximation of the true distribution.

More formally, we can define an n-gram language model as one that gives an approximate probability score for a word sequence $\boldsymbol{W} = w_0 w_1 \cdots w_m$. This probability can be obtained by considering local scores using $n-1$ Markov chains over the word sequence[Hif12], that is:

$$P(\boldsymbol{W}) = \prod_{i=1}^{m} P(w_i | w_{i-n+1}^{i-1})$$

In the special case of a bigram (that is, 2-gram), we get

$$P(\boldsymbol{W}) = \prod_{i=1}^{m} P(w_i | w_{i-1})$$

where $P(w_i|w_{i-1})$ can be estimated as

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

where $c(w_i, w_{i-1})$ is the count of w_i and w_{i-1} being spotted next to each other (in that order) and $c(w_{i-1})$ is the number of times the word w_{i-1} occurred in the text.

Inference (prediction of a proper diacritization) using this algorithm works as follows: all possible diacritizations are generated, and then the formula above is used to score them. More formally, the problem of restoring diacritics using n-gram models can be defined as a problem of choosing a word sequence $\boldsymbol{H}$ with the maximum model score, given the sentence without diacritics, that is:

$$\boldsymbol{H} = \arg max_{\boldsymbol{H}} P(\boldsymbol{H}|\boldsymbol{W})$$

One of the biggest issues with this method is that given its frequentest nature, it may cause numerical issues in many places, since $c(w_{i-1}))$ and $c(w_i, w_{i-1})$ may often times be zero and thus cause the probability they out to estimate to either be zero, or in the worst case trigger a division by zero error. The authors in [Hif12] deal with these issues by utilizing a method called smoothing, which has dramatically improved their results (word error rate reduction from 61% to 9.2%).

While modeling words directly may be the easiest way of setting up the model, upon deeper consideration, one may decide to use a framework normally used in computational linguistics for Part of Speech tagging, and model only the distinctions necessary to resolve the major accent ambiguities. In the case of Spanish for instance, this would be *-o/-ó* or *-ara/-ará* for instance. Compared to the task of Part of Speech tagging, one big advantage of this formulation of the task is that it does not require annotated corpus data, which are usually hard to come by[Yar99].

This method would therefore transform the Spanish sentence

la posición anunció oficilamente que

into the following list of "words":

la/LA posicion/-IÓN anuncio/-Ó oficilamente/-MENTE que/QUE

Note that the forward slashes above were only added for better readability. A model trained on this transformed data would then be used exactly as we described above. Using this method, the authors in [Yar99] were able to improve the accuracy from 57% (baseline) to as much as 97,8% in some cases.

Evaluation Metrics

As we mentioned earlier, simple accuracy many times does not fully capture the complexity of the considered task, and is therefore quite hard to interpret correctly, especially when comparing two models of similar quality with their accuracies being well over 90%. Some of the biggest issues with this metric could already be visible in the context of the following baseline algorithm. Suppose that we have a dataset in which one out of each 10 words has an ambiguous form. We train a model on this dataset, which achieves 88% accuracy on the test set. Upon manual inspection of its errors we notice that it associated more weight with forms which were used in a less ambiguous way in the training set, hence inducing some bias. We thus arrive at a strange conclusion: we have a model that tries to actually restore diacritics but its performance is worse than the baseline model that only copies its input to the output, as such a model would like achieve accuracy close to 90%. It is therefore quite obvious that a better metric is necessary.

An alternative set of metrics that might be considered are `precision` and `recall` which are defined as follows:

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

where tp is the number of true positives, fn is the number of false negatives and fp is the number of false positives. These two metrics can be combined into a single number metric called F1-score by taking their harmonic mean, that is:

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

In practice the F1 score is regarded as much more robust measure of relevance than accuracy, mostly due to the fact that it does not only take into the account the mistakes made by a given model, but also the way in which these mistakes were made (i.e. whether they were false positives or false negatives).

These metrics (most notably the F1 score) were used by the authors in [UBP+08] where they utilized them in order to evaluate their unigram, bigram and trigram models, although in a different context (character level diacritics restoration).

1.1.3 Bayesian classifiers

Bayesian classifier are a class of classifiers which has been used for many practical tasks of computational linguistics, such as word-sense disambiguation, authorship identification or person-place classification of proper nouns [Yar99]. They have been shown to be well suited for handling longer contexts, which was problematic with the models discussed above. In this section, we will discuss a specific member of this class of classifiers: the Naive Bayes classifier.

Despite its name and naivete involved in its design, the Naive Bayes classifier is widely used in various classification tasks and its use is not at all limited to the field of Natural Language Processing. In the case of diacritics restoration, it was most notably used for the Māori language [CK11].

It is based on the application of the Bayes' theorem combined with an independence assumption, which is assumed between its input features. Given class c and dependent feature vectors x_1 through x_n, the Bayes' theorem provides the following:

$$P(c|x_1, \cdots, x_n) = \frac{P(c)P(x_1, \cdots, x_n|y)}{P(x_1, \cdots, x_n)}$$

Using the naive assumption described above, which states that for every i

$$P(x_i|c_i, x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n) = P(x_i|c)$$

we obtain

$$P(c|x_1, \cdots, x_n) = \frac{P(c)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \cdots, x_n)}$$

Since we are considering a classification task and the denominator is constant, given the input, we can simplify this into a separate classification rule:

$$c = \arg\max_c P(c)\prod_{i=1}^{n} P(x_i|c)$$

Since the considered probabilities are normally quite small and might pose issues in computation, the following form is often used:

$$c = \arg\max_c \log P(c)\sum_{i=1}^{n} \log P(x_i|c)$$

The probability $P(x_i|c)$ is usually estimated in the same frequentest way as we already described above. The other probability necessary $P(c)$ is usually simply estimated as

$$P(c) = \frac{N_c}{N}$$

Where N_c is the number of words in a given class c in the training dataset, whereas N is the number of all training examples. In order to fight against possible zero estimates, the authors in [CK11] apply Laplacian smoothing to their probability estimates.

The final piece of information necessary before these models can be considered in the classification task is the features that will be used at the time of inference. The authors in [CK11] use n-grams that either precede or follow the target word. Moreover, they introduce a convenient notation in order to better identify them. For instance (-1, 3) means a trigram preceding the target word, (1, 2) indicates a bigram following the target word and (-1, 1), (1, 1) describes a monogram on either side of the target word. The authors of the aforementioned work went as far, as to combine five of these features. Their final feature set therefore consists of (-1, 3), (-2, 2), (1, 2), (-1, 4), (-2, 4). This combination of features, utilizing the Naive Bayes classifier, was able to achieve 99,01% accuracy, which was a substantial improvement over the baseline (97,11%).

1.1.4 Other models

While the previously described models constitute the backbone of diacritics restoration techniques at word level, there are many other that can and were used for this task in

the context of various languages. For instance, many previously used modes utilized Part of Speech tagging [SD01], but due to unavailability of solid Part of Speech taggers and a push for end-to-end[1] solutions, these models were not frequently used in the past few years. Moreover, other models utilize Conditional Random Fields [üAE14], specifically tuned and constructed language models [AFNR12], and translation modeling combined with language modeling [LEF16].

1.2 Grapheme-based Diacritics Restoration

As we described in the previous section, the word-based diacritics restoration is a task that is not thought of to be very difficult and at the same time provides very good results, even with quite simple models.

However, as noted in [MN02], most of these methods fall short when

1. Dictionaries in electronic forms are not available for the target language, or only dictionaries of small sizes are publicly available. A specific situation happens when the dictionary itself lacks diacritics, in that case its usefulness in the context of diacritics restoration is very limited.

2. Morphological and/or syntactic analysis tools (such as Part of Speech tagger) are not publicly available for the target language.

3. Size of the dataset on which the diacritics restoration model is to be trained is small.

It is therefore obvious that word-based approach is in many ways limited. The authors in [MN02] therefore proposed a model that utilizes information on the grapheme (character), rather than word level. The basic premise is as follows: the local graphemic context encodes enough information to solve this disambiguation problem [DPWDS07]. Such a problem can then be formulated as a machine learning task and solved using well known machine learning tools.

In order to provide some more context, let us consider the differences between rules learned by models that work on the word level, compared to those that consider grapheme (character) as the most atomic unit of input. While on the word level the model can for instance learn a rule like

" posicion should change to posición when it is a noun"

the grapheme-based model will learn something on the order of

"c followed by an i and preceded by whitespace should change to č"

[1] text-to-text, without any middle man, may be more appropriate

We can see that the latter rules are much more general: they do not require a specific word to be seen and will most probably require much less space in order to be stored.

Left	Left	Left	Left	Left	Focus	Right	Right	Right	Right	Right	Class
-	-	-	-	-	**m**	b	u	r	i	-	m
-	-	-	-	m	**b**	u	r	i	-	-	b
-	-	-	m	b	**u**	r	i	-	-	-	ũ
-	-	m	b	u	**r**	i	-	-	-	-	r
-	m	b	u	r	**i**	-	-	-	-	-	i

Figure 1.2: restoration of diacritics for the word "mbũri" (goat in the G̃ikũyũ language). Note that every character is considered in its own context. [DPWDS07]

Since the input space is different compared to word-based models, the features used to train models in this context will be different too. We will illustrate this on the example of restoration of diacritics for the word "mbũri" (goat in the G̃ikũyũ language).

As we can see in Figure 1.3, for every class (grapheme, for instance "ũ") that is to be predicted (the last column), there are eleven features. One of them is the character we are currently changing (the middle column, for instance the character "u"), the left and the right context. In general, we say that the model has a context window of N when the features constitute N characters from the left and N character from the right. The authors of [MN02] did an exploration on what window size would be best suited for this task and in the case of Romanian found that the bigger the window, the better, with $N = 5$ being the optimum in this case. However, at some point the improvement plateaus and increasing the window size brings diminishing returns. This therefore suggests that window size is a hyperparameter we need to optimize our models over. Moreover, the authors of this seminal work also investigated the relationship between the size of the training dataset and the performance of the model. While the obvious hypothesis has been confirmed (the more data the better – regardless of the size in this case), the authors were able to show (again, in case of Romanian) that even when using very small amounts of data (on the order of 100 examples) the proposed model is already able to beat the baseline, which suggest that it generalizes very well.

The authors of [DPWDS07] tested this model's ability to generalize on resource-scarce languages. While for many Indo-Euroean languages a suitable textual data is not publicly available, in case of many African languages considered in [DPWDS07] a suitable dataset (at least in terms of size) is not available at all. In order to better asses the difficulty of the diacritics restoration task, the authors also introduced so called **LexDif** – "lexical diffusion" metric. To compute this value, a list of all unique

words is first extracted. Every word in this list is then latinized (its diacritics are removed). As one would expect, multiple unique words will be latinized into the same form. The value of **LexDif** is then computed by dividing the number of unique words by the number of unique latinized words. This number therefore indicates the average number of alternations that are to be disambiguated as part of the diacritics restoration task – the higher the number, the more difficult the task, since we can only predict single alternative for a given latinized word form.

Given the input features mentioned above, the models can be trained using many different methods. In [MN02], the authors mention that they used an instance based learning algorithm called TiMBL, while also comparing its performance with a specific algorithm for training decision trees. While the decision trees provided similar results and could also provide higher interpretability of the created models (by examining its nodes/leaves), given its significantly higher running times, they decided against using it. In [DPWDS07] the authors noted that although other machine learning algorithms like Maximum Entropy Learning or Support Vector Machines (SVMs) are typically able to outperform memory based learning on many Natural Language Processing tasks, in this case they were not able to conclude that that would be the case. A similar conclusion was reached by the authors of [KDDV17], where in the context of Lithuanian text a Conditional Random Field classifier has been outperformed by a standard count-based language model.

This does come as a surprise, since in the latest work on the topic to date [AH16] test logistic regresion, SVMs and also Random Forests, which can be though of as of a generalization of the C4.5 algorithm using boosting techniques. In this work the authors were able to confirm the conclusions from earlier work on generalization and window size, while also creating a model that is both very fast and very accurate, even in context of very short messages from the social media.

1.3 Diacritics Restoration using Deep Neural Networks

Since Deep Neural Networks and Deep Learning in general has been taking the field of Machine Learning by a storm, it was only a matter of time when the field of Diacritics Restoration gets hit. The first work on this topic was [RASRR14], which utilized a model called Deep Belief Network, to solve the classification task we described above. While this model managed to achieve state-of-the-art performance at the time, it was quickly surpassed by other works of similar type, such as for instance [BG15] and [AGAS+15], both of which made use of a much more robust technique called Long-Short

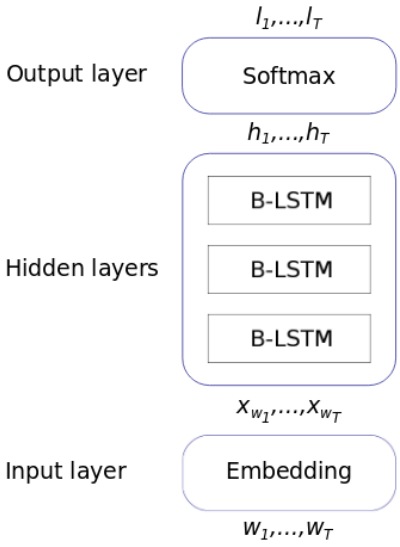

Figure 1.3: An illustration of the topology of the network used in [BG15]

Term memory (LSTM) and also designed the model to work end-to-end[2]. However, since then many alternations of the LSTM model were proposed, which suggests that there is still room for improvement, which is also suggested in the conclusion of the aforementioned work. This was indeed realized in [Náp17], where the author tested various architectures on both the word and character level, and evaluated them on Czech texts.

Further discussion on the topic of Diacritics Restoration using Deep Neural Networks along with our contribution to this are can be found in Chapter 3.

[2]By that we here mean that text comes in on one side and goes out on the other, which is not necessarily the case in other models that achieve near state of the art results by utilizing morphological and other language-specific tools.

Chapter 2

Deep Neural Networks for Natural Language Processing

In this chapter we provide an introduction to Neural Network models, with special focus on their deep variants, and their usage in Natural Language Processing applications. Our goal here is not to provide a comprehensive survey, but rather to provide background for the forthcoming discussions on the proposed models that are central to the presented work.

2.1 Neural Networks

Although they are no longer primarily viewed as such, Neural Networks as a computational model has been introduced with the goal of modeling biological neural processes that take place in the brain. The model itself has proven to be useful when applied to vast amount of practical problems. While its further developments no longer took inspiration just from its biologically inspired past, for the sake of completeness, we provide a high-level overview of the biological system these networks were first tasked with modeling.

2.1.1 Biological Neural Networks

As the name suggests, biologically inspired Neural Networks consist of neurons, which are the basic building blocks of computation inside the brain. Around 86 billion of them can be found in the human nervous system, connected with 10^{14} to 10^{15} synapses. Each neuron receives input signal from its dendrites and passes forward its output signal along a single axon. This axon later branches out and connects to other dendrites of other neurons using synapses. When viewed computationally, the signal that travels from the axons is multiplied by the synaptic strength of a given synapse. Synaptic strength can then act as a learnable parameter that makes a single synapse act in

either excitory or inhibitory way, depending on whether a given synaptic strength (also referred to as weight) is positive or negative. If the sum of signal received by a single neuron reaches a certain threshold, this neuron can fire by sending a spike of signal along its axon. In the computational view, the precise timing of this spike is considered not to play a role, passing of information is mediated just by the frequency of firing. The firing rate of a single neuron is then modeled by a so called activation function, the output of which should represent the frequency of the output spikes along its axon. The most usual choice for an activation function has historically been the sigmoid function. It takes a real valued input (in this case a sum of neuron's input transformed by synaptic strengths of its input synapses) and produces a single real valued output from the range of 0 to 1. An illustration of this model can be seen in Figure 2.1.

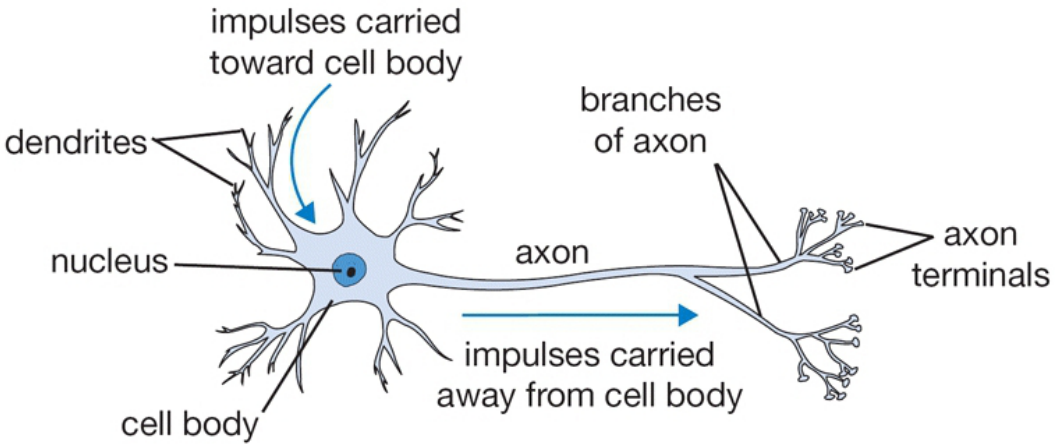

Figure 2.1: An illustration of a biological neuron [LKJ15]

Before proceeding onto a discussion on technical details of the described computational model, let us first stress out that the model itself is very coarse. Modern neuroscience recognizes many different types of neurons, dendrites are much more complex than just a simple weight and the timing of spikes has been shown to be important, which suggests that approximating information passing with just frequency may be inadequate. The computational model of neurons as described in detail in the upcoming sections is therefore only loosely inspired by the biological processes that take part in the brain. For further review we encourage the reader to consult [LH05] and [BHR14].

2.1.2 Modeling Neural Networks

Building upon the biological inspiration described in the previous section, we first propose a mathematical model of a single neuron and then extend it to a so called feed-forward neural network.

Modeling a Single Neuron

We denote an input signal that is flowing to a given neuron along its i-th axon as x_i in mathematical notation. Furthermore, the i-th synapse's weight is denoted as w_i. The sum of the input to a given neuron can thus be expressed as $\sum_i w_i x_i$. A bias term b is then usually added to this sum as well. Finally, a non-linear activation function f is applied on the resulting sum. The of $f(\sum_i w_i x_i + b)$ is thus the output of a given neuron's computation. This model is visually illustrated in Figure 2.2.

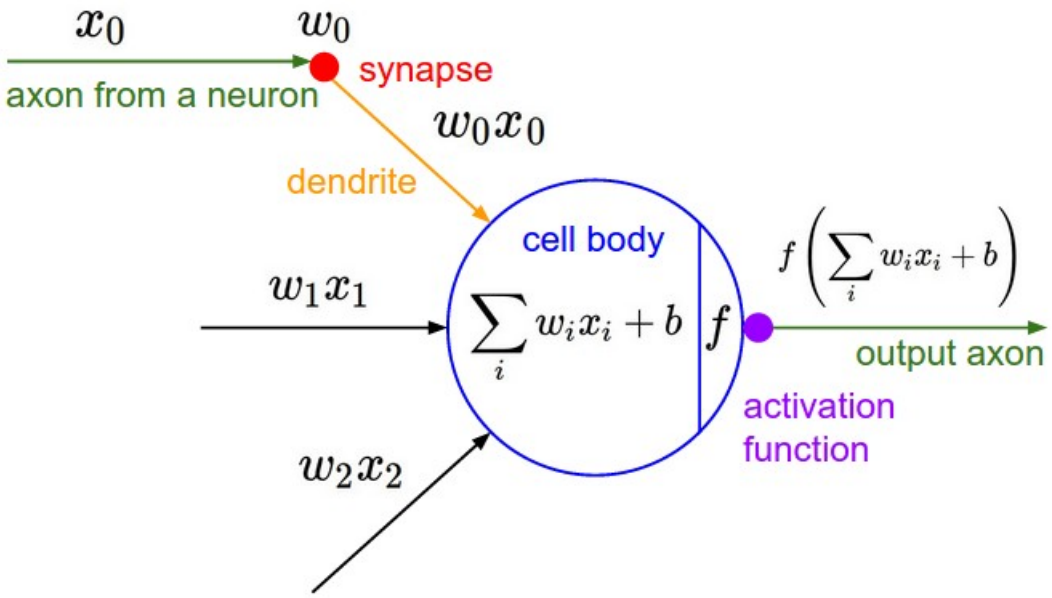

Figure 2.2: An illustration of a mathematically modelled neuron [LKJ15]

Despite its simplicity, this model is very general, which seems to be a great extend the reason for its success. Let us for instance consider that $f(\sum_i w_i x_i + b)$ can be easily interpreted as a binary classifier that would model $P(y_i = 1|x_i; w)$. Since the probability of both classes needs to sum to one, the probability for the other class would be $P(y_i = 0|x_i; w) = 1 - P(y_i = 1|x_i; w)$. This interpretation along with so called cross-entropy loss would lead to a binary Softmax classifier, which is also known as logistic regression. In a very similar manner, one may decide to make use of the max-margin hinge loss, the optimization of which would yield a binary Support Vector Machine. These models also usually make use of regularization terms in their loss functions, which improve their generalization capabilities by ensuring that no input dimension alone greatly influences the overall output. In our biological view this could be interpreted as gradual forgetting, since it would encourage the weights w_i to stay low at each parameter update.

In conclusion, the general nature of this model provides a simple yet effective framework for expressing many Machine Learning models in a unified fashion. In the next section we discuss one of its key parts: the so called non-linearities or activation func-

tions.

Activation Functions

Although the activation functions described in the following paragraphs differ, in principle they all perform the same function: they take a single real number as an input and performs a mathematical operation on this input. Note that as their name suggests, the described functions are generally non-linear.

Sigmoid The sigmoid activation function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

. Its interpretation is fairly simple: it takes a real number and "squashes" it between 0 and 1. As we can see directly from the definition and also in Figure 2.3, this squashing is most severe around the extremes: for very large negative numbers the function would output 0, whereas for very large positive numbers it would output 1.

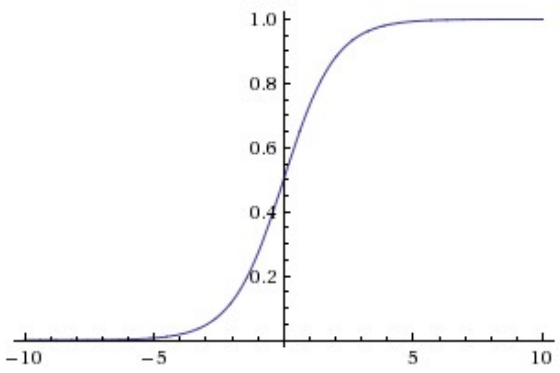

Figure 2.3: An illustration of the output of the sigmoid activation function (y axis). [LKJ15]

As we mentioned before in Section 2.1.1, this function has been historically a very common choice of non-linearity for neural networks, mostly because it can be viewed as directly mapping to the concept of firing rate of a neuron. With this activation function the neuron can model a wide range of firing rates: from not firing at all (0) to firing at maximum capacity (1).

In recent years, however, this non-linearity has fallen out of favour. The most important reason is that it saturates gradients at both extremes – the gradient there would be very close to zero. The result is that almost no signal will flow through a neuron in this case during training, since it effectively "kills" the gradient. It should also be noted that the exponential function that is used in this non-linearity is quite computationally expensive.

Tanh The tanh activation function is defined as

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

. It takes an input and transforms it into the range [-1, 1]. We also note that the the tanh activation function is just a rescaled sigmoid, since $tanh(x) = 2\sigma(2x) - 1$. As such, it suffers from many of the same issues as the sigmoid activation function. Still, it is quite frequently used in NLP applications.

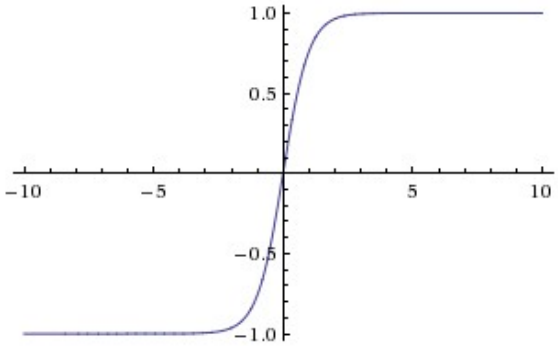

Figure 2.4: An illustration of the output of the tanh activation function (y axis). [LKJ15]

Hard tanh In order to address the shortcomings of the tanh activation function, a so called "hard-tanh" activation function has been proposed. This function is simpler and also faster to compute, as it only approximates the tanh function. It is defined as

$$hardtanh(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & otherwise \end{cases}$$

Rectifier Linear Unit (ReLU) The ReLU non-linearity has become very popular in the past few years, especially in the context of Computer Vision. It is simply defined as

$$relu(x) = max(0, x)$$

As we can see from the definition, this function is very easy to compute – it can be implemented by essentially just thresholding any activation input at zero. It has been found empirically that using this non-linearity has the potential to greatly accelerate the convergence of training of networks that make use of this non-linearity – in [KSH12], which is considered to be one of the most famous examples of this phenomena, it was

found to increase the convergence rate six fold when compared to networks that used sigmoid or tanh activation functions.

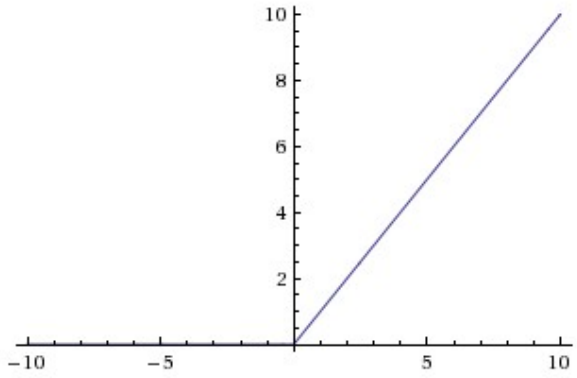

Figure 2.5: An illustration of the output of the ReLU activation function (y axis). [LKJ15]

One of the major downsides of the ReLU non-linearity is that nodes that make use of it can also "die". A very large update of the weights associated with this node can make it so that the node will not respond to any of the data points in the (training) dataset. This effectively makes the node "dead" as its weights lose their capacity to be updated again during training.

Leaky ReLU This non-linearity attempts to fix the problem of ReLU described above by ensuring that any node that uses this non-linearity is capable of responding to any input it receives, albeit in a very small manner. It is defined as

$$lrelu(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$$

where α is a small constant. While this non-linearity has been successfully used in many published works, the results of its usage are not consistent across input domains or considered problems.

Maxout The Maxout non-linearity is an example of an activation function that does not follow the general form of $f(w^T x + b)$, but rather tries to generalize both ReLU and Leaky ReLU into a single activation function. It is defined as

$$maxout(x) = max(w_1^T x + b_1, w_2^T x + b_2)$$

Note that in this formulation both ReLu and Leaky ReLU are a special case of this non-linearity: in case of ReLU $w_1 = 0$ and $b_1 = 0$ for instance. Its disadvantage is the number of weights, which need to be doubled in case of this non-linearity.

Choosing the Right Non-Linearity To the best of knowledge of the author of this document, there is currently no fundamental theory on what activation function to choose for which use case of a given network. While the practitioners do provide valuable rules of thumb, such as "ReLU units work better than tanh, and tanh works better than sigmoid" [Gol15], they are also quick to point out that in specific cases even more exotic functions, such as for instance the Cube activation function (that is $cube(x) = x^3$) or the tanh cube activation function (defined as $tanhcube(x) = tanh(x)^3$), have been found more effective in certain situations (see for instance [CM14] and [PGC15]). On the other hand, we have non-linearities such as Maxout with interesting properties, but increases in terms of model size.

The answer to the question "which non-linearity should one choose" therefore seems to be tightly connected with the problem one desires to solve. Looking at the recently published literature in the field however, it seems that starting with ReLU and gradually moving towards Leaky ReLU or Maxout if the number of "dead" neurons is of concern seems like a solid choice. When it comes to NLP applications specifically, the tanh non-linearity seems to be worth trying out as well. Note that the biological plausibility of activation functions other than sigmoid remains yet to be determined – they were generally inspired by the need for solving an engineering problem rather than the need for a better model of a biological process.

2.1.3 Neural Networks as Graphs of Neurons

As their name suggests, Neural Networks are modeled as a collection of neurons that comprise a network. This network has further properties, namely that it is a direct acyclic graph. Such a property ensures that information flows in a single way throughout the network and that an infinite look in the forward pass through the network will not occur. While in principle Neural Networks may not need any other property, for ease of implementation they are usually organized in a number of layers, each of which may have a specific function. In order to describe this concept in more concrete terms, let us introduce one of the most common Neural Network layer: a fully-connected layer, which is also known as a dense or affine layer.

Fully-connected Layer

A fully-connected layer is simply a layer that has pairwise connections from each of the previous layer's neuron to each of its neuron (hence the "fully-connected" in its name), while the neurons on the same layer are not connected to one another. An illustration of an architecture that makes use of this layer can be seen in Figure 2.6. Note that the fully-connected layers in this figure are labeled as "hidden". The notion behind this label is that it is the part of the network that the end user of the network does not

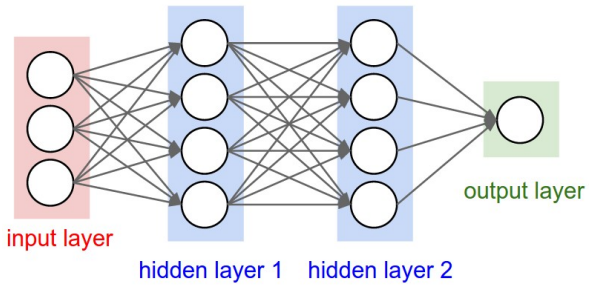

Figure 2.6: An illustration of a three-layer neural network comprised of two fully connected and one output layer. Note that the number of layers does not include the input layer. [LKJ15]

have a direct access to. For a more in-depth discussion on what the hidden layers of a fully-connected feed-forward network represent, see [TP89].

A network with the architecture described in Figure 2.6 is also sometimes referred to as "Artificial Neural Network" or "Multi Layer Perceptron".

Output Layer

The output layer is essentially a special case of a fully-connected layer, which does not use an activation function[1]. The reason is fairly straightforward: the output values are usually interpreted as either real valued outputs in the case of regression or class scores in case of classification.

2.1.4 Modeling Neural Networks Mathematically

While the biologically inspired view of Neural Networks allows us to discuss many related high level concepts without going into too much technical detail, it also prevents us from providing a clear picture of how would a model like this be implemented, which is a crucial prerequisite for further discussions. In this section we therefore provide a description of previously introduced concept in mathematical notation which make it easy to extend them in many ways and also provide a direct starting point for their implementation.

Perceptron

The simplest type of a Neural Network is a perceptron. In terms of our previous discussion, this is a neural network that only contains an input and an output layer. In mathematical notation this can be expressed as

[1] Alternatively, we can say that it uses the linear identity activation function, that is $id(x) = x$.

$$NN_{perceptron}(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$$

where $\mathbf{x} \in \mathbb{R}^{d_{in}}$, $\mathbf{b} \in \mathbb{R}^{d_{out}}$, $\mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}$ and d_{in} denotes the input dimensionality, while d_{out} denotes the output dimensionality. The $\mathbf{W}$ and $\mathbf{b}$ denote the weight matrix and bias vector, respectively.

Multi Layer Perceptron

The simple network above can then be extended to a Multi Layer Perceptron by adding a new layer which also makes use of a non-linear activation function. Its definition is as follows:

$$NN_{MLP}(\mathbf{x}) = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^{out} + \mathbf{b}^{out}$$

where $\mathbf{x} \in \mathbb{R}^{d_{in}}$, $\mathbf{b}^1 \in \mathbb{R}^{d_1}$, $\mathbf{b}^{out} \in \mathbb{R}^{d_{out}}$, $\mathbf{W}^1 \in \mathbb{R}^{d_{in} \times d_1}$, $\mathbf{W}^{out} \in \mathbb{R}^{d_1 \times d_{out}}$ and d_{in} denotes the input dimensionality, while d_{out} denotes the output dimensionality. The $\mathbf{W}^1$, $\mathbf{W}^{out}$ and $\mathbf{b}^1$, $\mathbf{b}^{out}$ denote the weight matrices and bias vectors on the hidden (1) and the output (out) layers respectively. The g function is a non-linear function, which is applied in element-wise fashion and allows the model to represent complex functions[2].

In a very similar way we can also extend this network with another hidden layer:

$$NN_{MLP2}(\mathbf{x}) = g^2(g^1(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2)\mathbf{W}^{out} + \mathbf{b}^{out}$$

For clarity, we may also decide to lay out this network in a layer-by-layer fashion:

$$\begin{aligned} \mathbf{h}^1 &= g^1(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1) \\ \mathbf{h}^2 &= g^2(\mathbf{h}^1\mathbf{W}^2 + \mathbf{b}^2) \\ NN_{MLP2}(\mathbf{x}) = \mathbf{y} &= \mathbf{h}^2\mathbf{W}^{out} + \mathbf{b}^{out} \end{aligned}$$

Each of the aforementioned lines would be called a layer of a neural network. Note that in the first two layers a non-linearity is applied after the linear transformation, while the output layer does not contain a non-linearity. Note that is consistent with our definition of the output layer in Section 2.1.3. Furthermore, it also directly represents the neural network visualized in Figure 2.7, provided the sigmoid non-linearity is used in place of both g^1 and g^2.

Given this layer representation, one of the differences between the visualization and the mathematical representation is that the visualization omits the bias terms. This

[2]Without using it, the model would only be able to model linear transformations, as a sequence of linear transformations in the end results in just a linear transformation.

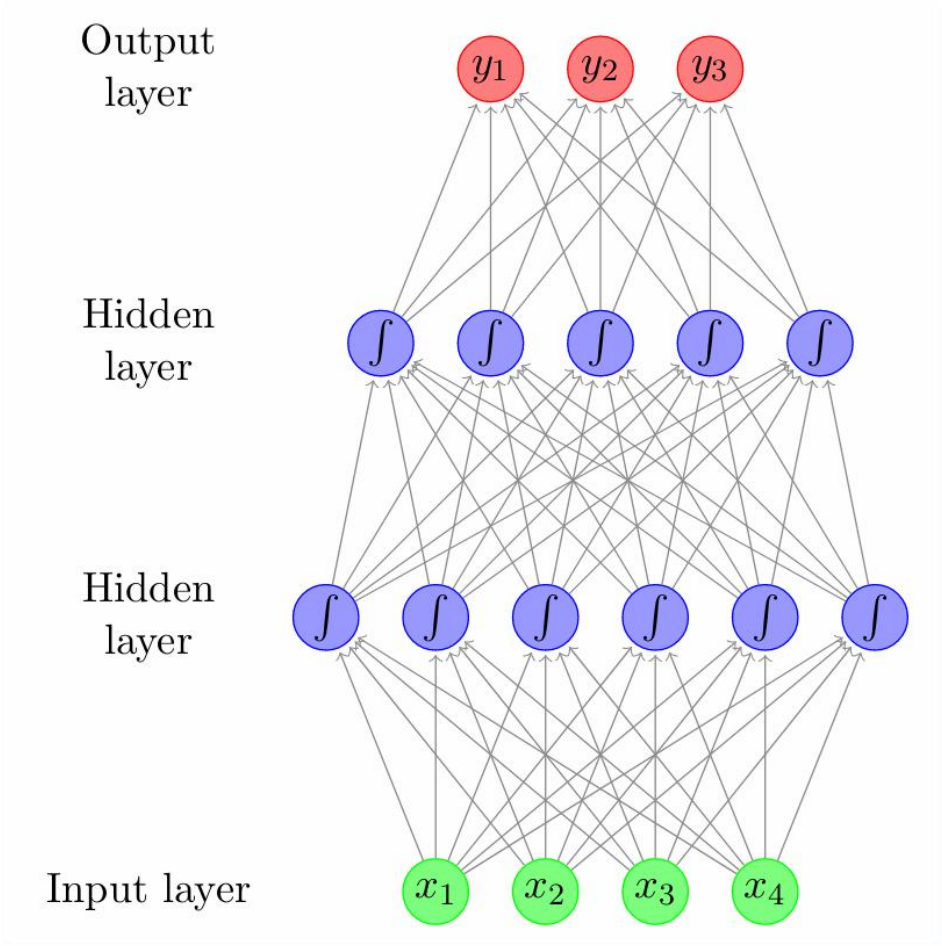

Figure 2.7: An illustration of a three-layer neural network comprised of two fully connected and one output layer. The input layer neurons are colored in green, the output layer neurons in red and the hidden layers, onto which a sigmoid non-linearity is applied, are colored in purple. [Gol15]

is usually the case for clarity reasons, as the bias can be easily implemented as part of weight matrices: instead of a bias term a layer will be augmented by a new neuron that does not have any incoming connections and whose value is always 1.

Output Transformation

Specifically in the context of classification, the vector resulting from the output layer operation is also transformed. Given its probabilistic interpretation, one of the most commonly used transformations is the softmax transformation:

$$\mathbf{x} = x_1, ..., x_k$$

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

The result of this transformation (which is once again applied element-wise on the output vector) is a vector of real valued numbers that sum to one, which makes it a probability distribution over k outcomes and thus a very important building block for probabilistic modeling using Neural Networks. When this layer is applied to a network without hidden layers, the resulting model is the same as a maximum-entropy classifier, or so called multinomial logistic regression model. This further shows that the generality of the basic building blocks of Neural Networks allow us to build models commonly used in other areas of Statistical Inference or Machine Learning.

2.1.5 A Note on Representation Power

The NN_{MLP} network described in previous sections has been found to be an universal approximator in [HSW89]. One may therefore wonder why would it be useful or necessary to use more than one hidden layer. While the theoretical proof in the aforementioned work states and proves that such a network exist, it does not specify its parameters, nor does it say anything about the way in which these parameters may be found. The reason why Neural Networks with multiple layers are used in practice (and why did the term Deep Learning become commonplace) is that as the authors of [LKJ15] say: "they compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice, and are also easy to learn using our optimization algorithms".

2.2 Feature Representation

While we considered the input $\mathbf{x}$ in the previous sections for clarity reasons to be just a vector of real values, especially in the context of NLP applications the representation of the input is of specific interest. This is because when dealing with natural language, the input $\mathbf{x}$ can represent various types of information, such as words, graphemes, part of speech tags, entity labels or a vast array of other linguistic information. One of the big changes the Deep Learning approaches for NLP brought to bare was to move from representing the input data as a set of unique features, where each unique feature would be represented by a specific dimension, but rather represent them using their "embedding" into a d-dimensional vector space instead.

Building upon this idea, the general structure of a NLP model for a classification task in this framework can be defined as follows:

1. Obtain a set of features $f_1, ..., f_n$ that are thought to be relevant to the task at hand.

2. For each feature obtain its representation in a d-dimensional vector space.

3. Combine the vectors into an input vector representation **x** by concatenation, summation, taking their element-wise average, max, mean or any combination of these.

4. Provide the input vector **x** to a Neural Network.

Let us briefly discuss the differences between the two approaches to representing the input features for Neural Networks. For a visualization of their differences, please refer to Figure 2.8.

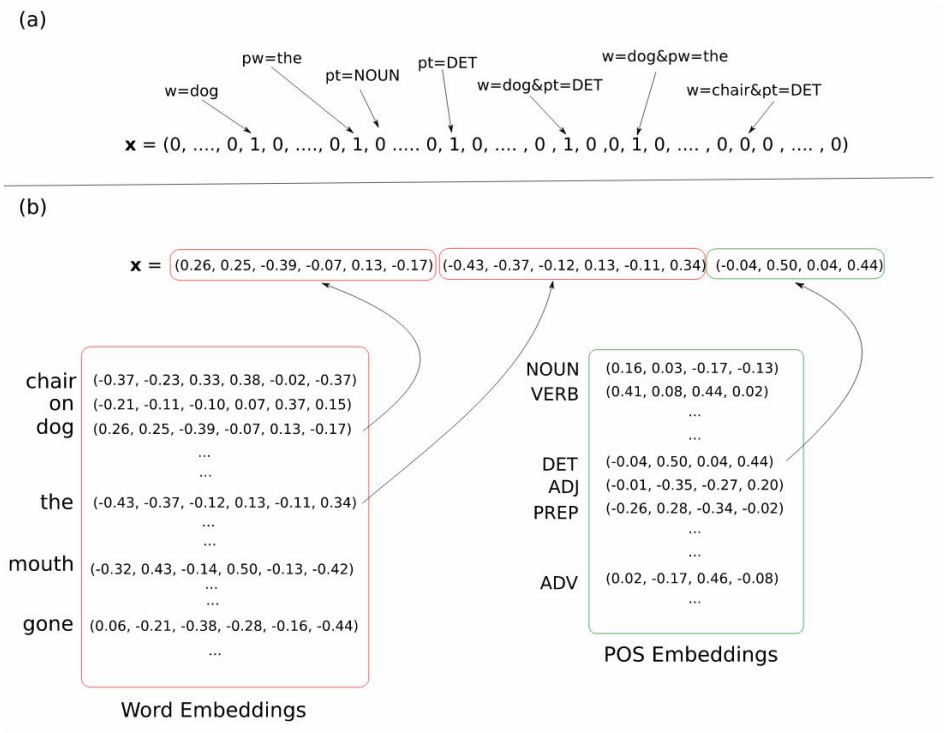

Figure 2.8: An illustration of the comparison between different feature representations. In both cases the following information is encoded: *current word is "dog"; previous word is "the"; previous part-of-speech tag is 'DET'*. In case of sparse feature vector (a), each feature, as well as combination of features, is represented by a separate dimension. Feature values are binary and the dimensionality of such a vector is very high. In case of dense feature vector (b), the dimensionality is lower, as the resulting feature vector consists of concatenated embeddings of both words and part-of-speech tags and combinations of features are not explicitly encoded (except for specific positions of respective embedding vectors in the resulting feature vector). [Gol15]

2.2.1 One Hot Representation

The one-hot representation is very similar to the well known "bag-of-words" representation that is often times one of the simplest ways of representing linguistic structure

in NLP applications.

In this representation, each feature is assigned a specific dimension. The dimensionality of the one-hot vector is the same as the number of distinct features. There is no notion of similarity between features represented this way – the feature *previous word is "neural"* would be as dissimilar to the feature *previous word is "king"* as the feature *previous word is "queen"*.

2.2.2 Dense Vector Representation

In this representation, each feature is assigned a d-dimensional vector. Similar features will tend to have similar vectors – information should be shared between similar features.

One of the direct advantages of dense vector representation is the input dimensionality: even the biggest d-dimensional embedding vectors are usually much smaller than the shallow one-hot encoded sparse vectors, which are usually difficult to work with. The main advantage of this type of representation is its improved generalization power. If we assume that similar clues may provide similar results, it may also make sense to make this similarity explicit in the input itself. More specifically, if we were building a classifier which should respond in a specific manner to mentions of entities of noble origin, it may make sense for it to share statistical strength between the features *previous word is "queen"* and *previous word is "king"*, which may in the end be represented by similar dense input vectors.

Although there may be cases in which it makes sense to use one-hot encoding, the majority of the recently published work makes use of dense representations. In many of these works the procedure of creating the representation of input in a dense fashion is generalized into so called "Embedding Layers", which actually make use of both of these representations.

2.2.3 Embedding Layers

The motivation for Embedding Layers is that while the input vector $\mathbf{x}$ can be comprised of multiple embedding vectors which can come from various sources, these embeddings of input can also be explicitly made part of the network itself.

Let us introduce $c(\cdot)$, a function which maps feature inputs into input structure suitable for the further parts of the Neural Network. Furthermore, let us introduce function $v(\cdot)$, which provides an embedding vector for a given feature input. A very common choice for the $c(\cdot)$ function is concatenation of embeddings of respective input features:

$$\mathbf{x} = c(f_1, \cdots, f_n) = [v(f_1); \cdots; v(f_n)]$$

Assuming that the embedding vector representations of input features share the same dimensions, another common option for $c(\cdot)$ is summation:

$$\mathbf{x} = c(f_1, \cdots, f_n) = v(f_1) + \cdots + v(f_n)$$

Many recent works in this area refer to $c(\cdot)$ as part of the network and assume the resulting vectors of the $v(\cdot)$ function to come from an "embedding layer" (sometimes also referred to as a "lookup layer"), which is usually the first layer of the network.

Suppose we have a Neural Network architecture which is tasked with predicting the part of speech tag of a given word. We would therefore like its input to consist of the previous word, current word and the following word. In order to facilitate this, we consider a vocabulary V of $|V|$ words, where each of the words is represented by a d-dimensional dense vector. The resulting collection of vectors can be represented by an embedding matrix $\mathbf{E}$ of dimensionality $|V| \times d$, where each row of this matrix corresponds to an embedded word. Furthermore, let us us define $\mathbf{f}_i$ as one-hot encoded vector that represents the i-th word. We can then easily see that evaluating $\mathbf{f}_i\mathbf{E}$ produces the embedded representation of the i-th vector: the aforementioned multiplication "selects" the corresponding row of the embedding matrix $\mathbf{E}$.

This leads us to a natural choice for the $v(\cdot)$ function:

$$v(f_i) = \mathbf{f}_i\mathbf{E}$$

which then allows the input of the network to consist of just one-hot encoded feature vectors. Note also that this choice also allows us to easily model a continuous alternative to the so called bag-of-words representation, in which only the presence of a given feature in the input matters. This continuous alternative, so called "Continuous Bag-Of-Words" (CBOW), in which the representations of respective words that can be found in the "bag" are summed together can be defined as:

$$CBOW(f_1, \cdots, f_n) = \sum_{i=1}^{n}(\mathbf{f}_i\mathbf{E}) = (\sum_{i=1}^{n}\mathbf{f}_i)\mathbf{E}$$

The $(\sum_{i=1}^{n}\mathbf{f}_i)$ part of the aforementioned definition then directly corresponds to the bag-of-words representation of the input data.

2.3 Recurrent Neural Networks

Up until now, our discussion on Neural Networks and their architectures assumed that they work in a feed-forward fashion. When looking at natural language, however, we can see that it is generally comprised of sequences: words are sequences of letters, sentences are sequences of words, documents can be though of sequences of sentences. As we pointed out, there are ways and techniques such as CBOW representation that let us encode features of variable length, which may be sufficient for the considered applications. However, this view ignores the sequentiality of language, namely that the order of letters in a word does matter, as well as the order of words in a sentence or the order of sentences in a document. The Recurrent Neural Networks (RNN), first introduced in [Elm90], provide a framework for modeling sequences while focusing on long term dependencies, which are of great importance when dealing with natural text.

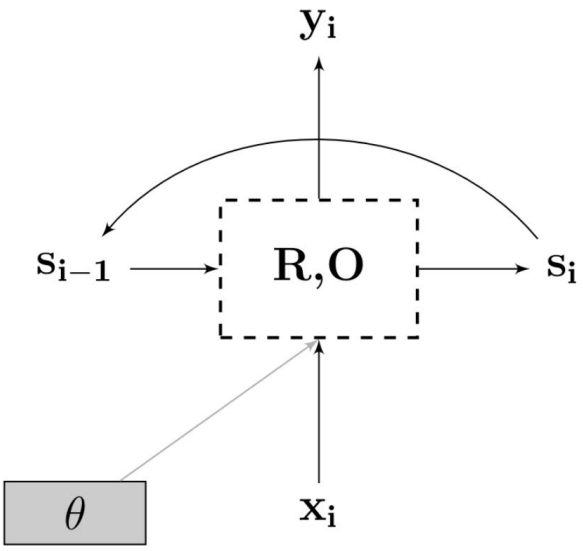

Figure 2.9: An illustration of the RNN architecture. [Gol15]

2.3.1 Defining Recurrent Neural Networks

Any RNN architecture can be expressed using mathematical apparatus by defining its two parts: the recursively defined function R that maps between the previous state of the model and current input into the next state of the model, and the output function O that takes as input the current state and outputs a output of the model. More concretely we have

$$
\begin{aligned}
RNN(\mathbf{s}_0, \mathbf{x}_{1:n}) &= \mathbf{s}_{1:n}, \mathbf{y}_{1:n} \\
\mathbf{s}_i &= R(\mathbf{s}_{i-1}, \mathbf{x}_i) \\
\mathbf{y}_i &= O(\mathbf{s}_i)
\end{aligned}
$$

where $\mathbf{x}_i \in \mathbb{R}^{d_{in}}$, $\mathbf{y}_i \in \mathbb{R}^{d_{out}}$ and $\mathbf{s}_i \in \mathbb{R}^{d_{hid}}$. Note that the functions R and O stay the same and thus the model is "forced" to keep track of the state of computation via the state vector. A visualization of this general architecture can be found in Figure 2.9.

Given the recursive nature of the R function and given an input of finite length (which is always the case with regards to NLP problems), this architecture can also be "unrolled" across different time steps. Visually this would result in the illustration that can be found in Figure 2.10.

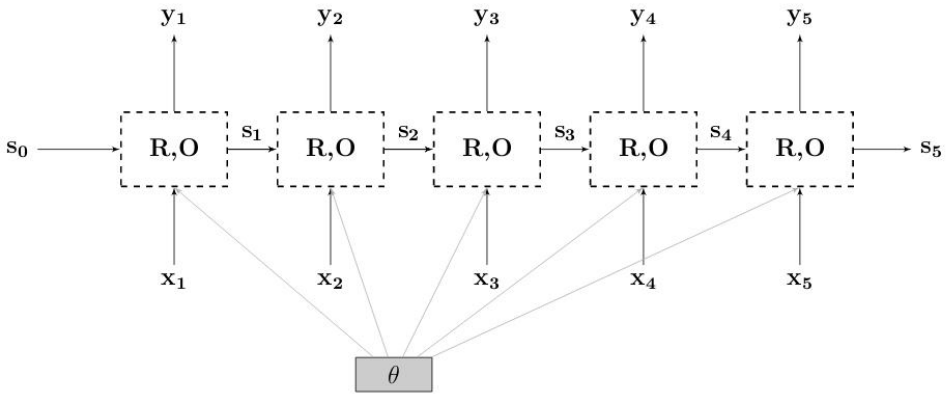

Figure 2.10: An illustration of the "unrolled" RNN architecture. Note especially the parameters θ which are shared across different time steps. [Gol15]

Multi-layer RNN

In much the same way as multiple feed-forward layers can be stacked one after another, so can any RNN layer produce output that can be consumed by another RNN layer. This layered architecture is also known as *deep RNN*. A visualization of this architecture can be seen in Figure 2.11

While to the best of the knowledge of the author of this document there did not exist at the time of its writing a plausible theory that would explain whether or how much representational power do the additional layers add. It has been empirically observed, however, that for some tasks adding further RNN layers improves the performance of the resulting model.

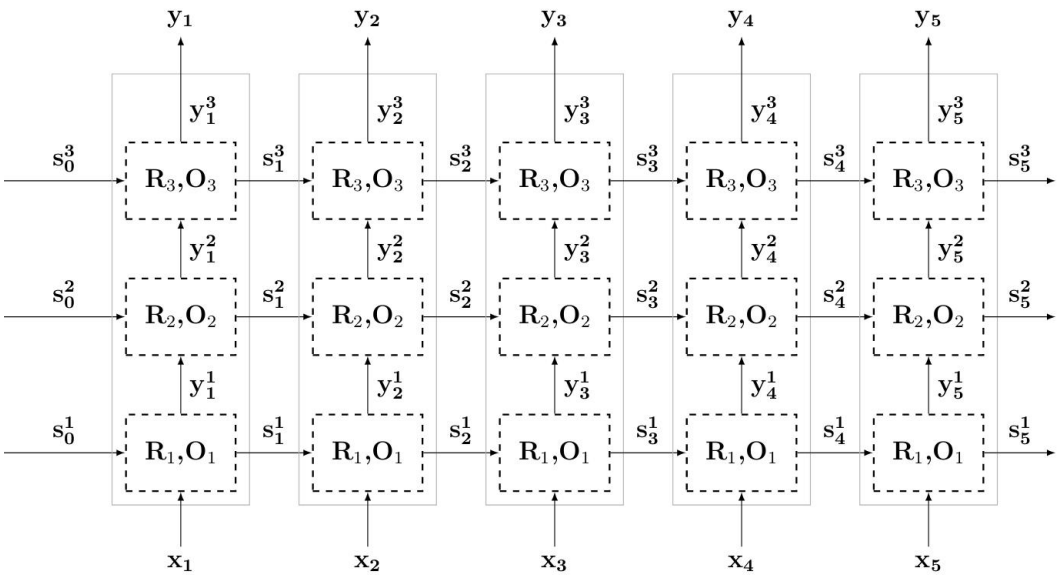

Figure 2.11: An illustration of the deep RNN architecture. [Gol15]

Bidirectional RNN

Another common variation of RNN models is to make them process the input in both directions, making them *bidirectional*. It can be easily motivated when considering the NLP task of predicting part-of-speech tags while giving the network plain words as the input.Using a single RNN gives the model at time step t access to all previous words from time 1 to time step t. However, given its task it may be beneficial for the model to be able to refer to also consider words after time step t. A simple way of providing this information would be letting the model process the input in reverse, from the end to the beginning.

In order to facilitate this, a RNN layer is often times doubled: one RNN is used for the forward (beginning to end) and one for the backward (end to beginning) pass through the data. The final output is then computed as a concatenation of outputs of both the forward and backward RNNs.

RNNs augmented in this way are usually denoted by the "Bi-" prefix – it is therefore not uncommon to find names such as BiRNN in the literature.

2.3.2 Simple RNN

The simplest RNN formulation (aptly named Simple RNN or Elman Network in honor of its author) can be described as follows:

$$\mathbf{s}_i = R_{SimpleRNN}(\mathbf{s}_{i-1}, \mathbf{x}_i) = g(\mathbf{x}_i\mathbf{W}^x + \mathbf{s}_{i-1}\mathbf{W}^s + \mathbf{b})$$
$$\mathbf{y}_i = O_{SimpleRNN}(\mathbf{s}_i) = \mathbf{s}_i$$

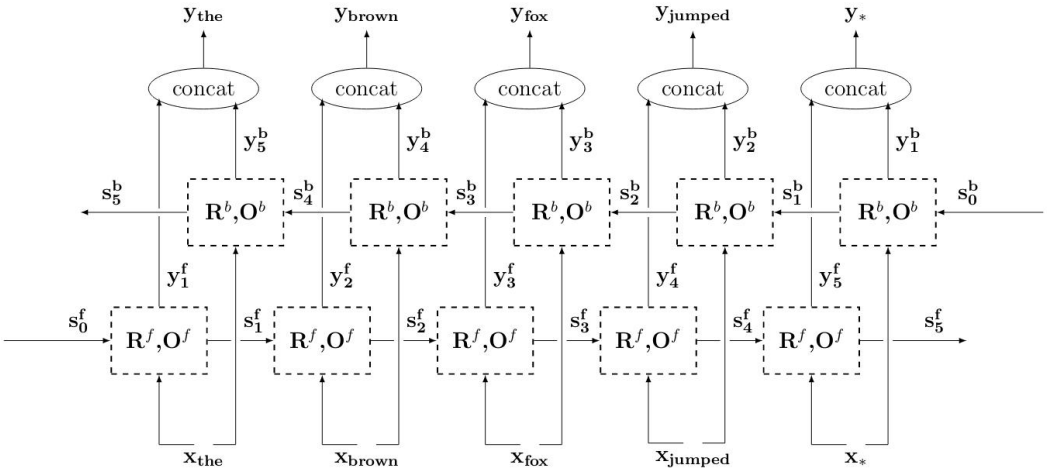

Figure 2.12: An illustration of the bidirectional RNN architecture applied on the sentence "the fox jumped .". Note that the illustration shows a single BiRNN layer with a separate RNN for both forward and backward directions of processing the input. [Gol15]

where $\mathbf{x}_i \in \mathbb{R}^{d_{in}}$, $\mathbf{s}_i, \mathbf{y}_i \in \mathbb{R}^{d_{out}}$, $\mathbf{W}^x \in \mathbb{R}^{d_{in} \times d_{out}}$, $\mathbf{W}^s \in \mathbb{R}^{d_{out} \times d_{out}}$, $\mathbf{b} \in \mathbb{R}^{d_{out}}$ and g is a non-linear activation function – most commonly tanh or ReLU. One interesting peculiarity to note is that while the state at time step i is computed as linear combination of previous state and current input further trasnformbed by a non-linear activation function, the output of this model is directly its state at time step i. This output function is sometimes defined to be more complex (it may for instance utilize further linear transformation and pass the result through another non-linearity), but for clarity of representation we will not consider these further transformations to be part of the RNN.

Despite its simplicity, SimpleRNN has been shown to provide very good performance in Natural Language Processing tasks, specifically language modeling and sequence tagging.

2.3.3 LSTM

One problem the SimpleRNN defined above faces is the so called "vanishing gradient" problem. When training the network, the error signals (most commonly differences between the predicted and true outputs, also referred to as gradients) decrease in magnitude with the length of sequence through which the error signal needs to propagate back. Given a sequence of sufficient length, the error signal may vanish completely, hence the name "vanishing gradient".

The Long Short-Term Memory (LSTM) architecture has been introduced in [HS97] to remedy this problem. Its basic idea is to augment the state of the network with a

sort of a "memory cell"[3] which is capable of preserving gradients over time. How much gradients flow into this reservoir is controlled by a so called "gates", a term borrowed from and very similar to gates in logical circuits.

$$
\begin{aligned}
\mathbf{s}_t = R_{LSTM}(\mathbf{s}_{t-1}, \mathbf{x}_t) = [\mathbf{c}_t; \mathbf{h}_t] \\
\mathbf{c}_t &= \mathbf{c}_{t-1} \odot \mathbf{f} + \hat{\mathbf{c}} \odot \mathbf{i} \\
\mathbf{h}_t &= tanh(\mathbf{c}_t) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{x}_t \mathbf{W}^{xi} + \mathbf{h}_{t-1} \mathbf{W}^{hi}) \\
\mathbf{f} &= \sigma(\mathbf{x}_t \mathbf{W}^{xf} + \mathbf{h}_{t-1} \mathbf{W}^{hf}) \\
\mathbf{o} &= \sigma(\mathbf{x}_t \mathbf{W}^{xo} + \mathbf{h}_{t-1} \mathbf{W}^{ho}) \\
\hat{\mathbf{c}} &= tanh(\mathbf{x}_t \mathbf{W}^{x\hat{c}} + \mathbf{h}_{t-1} \mathbf{W}^{h\hat{c}}) \\
\mathbf{y}_t = O_{LSTM}(\mathbf{s}_t) = \mathbf{h}_t
\end{aligned}
$$

where $\mathbf{x}_t \in \mathbb{R}^{d_{in}}$, $\mathbf{s}_t, \mathbf{y}_t \in [\mathbb{R}^{d_{out}}; \mathbb{R}^{d_{out}}]$, $\mathbf{W}^{x\square} \in \mathbb{R}^{d_{in} \times d_{out}}$, $\mathbf{W}^{s\square} \in \mathbb{R}^{d_{out} \times d_{out}}$ and furthermore $\mathbf{c}_t, \mathbf{h}_t, \mathbf{i}, \mathbf{f}, \mathbf{o}, \hat{\mathbf{c}} \in \mathbb{R}^{d_{out}}$. Note that the bias terms normally present in the definition of the respective gates has been omitted for clarity. A visualization of this model can be found in Figure 2.13.

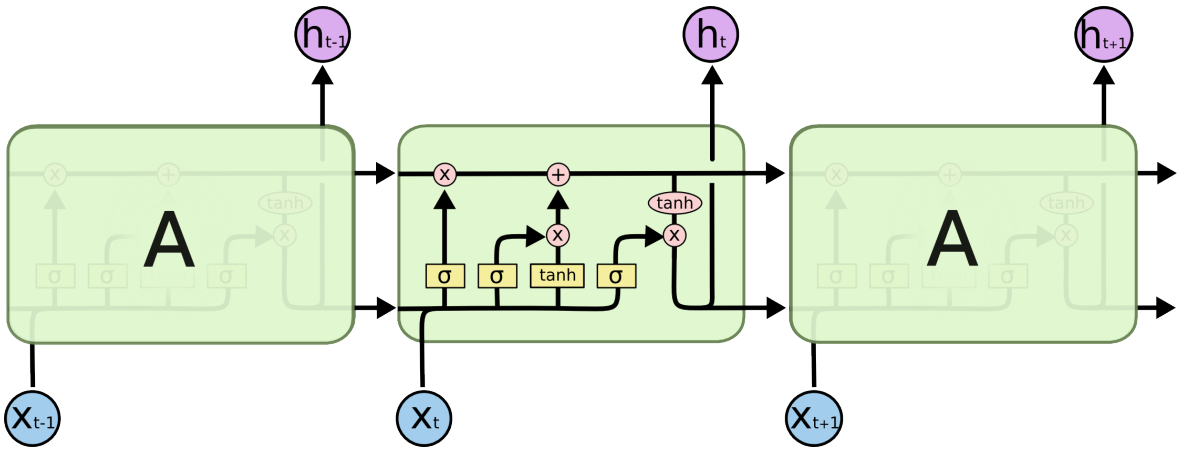

Figure 2.13: An illustration of the LSTM architecture in forward direction. Note that the red circles contain point-wise mathematical operations, yellow boxes contain non-linearities associated with Neural Network layers and black pathways signify the flow of information. [Ola15]

Let us attempt to describe the functionality of this model by first describing the respective gates. Each of this gates make use of the sigmoid non-linearity, which allows the model to learn parameters that affect how much should the current input affect the current state and how much of the current state should actually be provided as

[3] Many authors in the literature use the term "cell state"

the output at the current time step. The sigmoid non-linearity gives these gates a nice representation: for each dimension of the considered state it provides a number from 0 to 1 which signifies how much of this dimension's information should be discarded (0) or preserved (1).

Candidate memory state $\hat{\mathbf{c}}$ Although the so called "candidate memory state" $\hat{\mathbf{c}}$ is not strictly a gate, we include it here as it is necessary for further discussions on the function of the other gates.

As we can see in the definition above, the candidate state is computed from the current input and previous state, after which the tanh non-linearity is applied.

Forget gate f The function of the forget gate is to control how much of the previous state should be "forgotten" when computing the new memory state. A reverse interpretation is also possible: the value of $(\mathbf{1} - \mathbf{f})$ would determine how much of the previous memory state should be retained. A visualization of this gate can be seen in Figure 2.14.

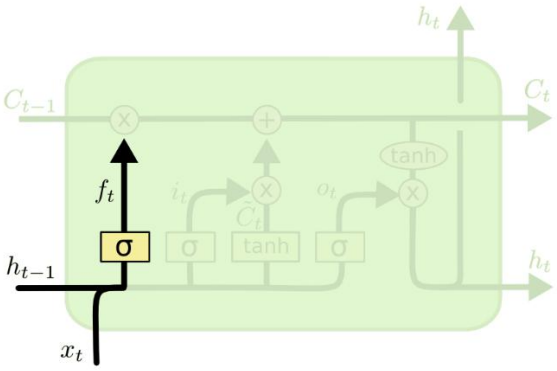

Figure 2.14: A visualization of the forget gate. [Ola15]

Input gate i The input gate is responsible for the amount of information that the current state receives from the previously computed candidate memory state, which is in turn computed as a function of current input to the model. A visualization of this gate can be seen in Figure 2.15.

Output gate o The output gate takes care of controlling what will the model's output at a given time step be, once again using current input and previous state as its input.

Using these gates and other auxiliary computations, the two parts of an RNN, the recursive function R and the output function O can be defined. Following the formalism

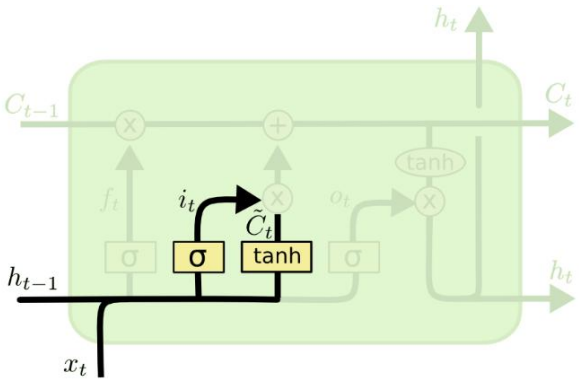

Figure 2.15: A visualization of the input gate. Note that the uppercase $\hat{C}$ in the picture is functionally the same as $\hat{\mathbf{c}}$ we used in our descriptions. [Ola15]

introduced above, we will do so by discussing the function of memory state $\mathbf{c}_t$ at time t and hidden state $\mathbf{h}_t$ at time t.

Memory state $\mathbf{c}_t$ When computing a new memory state $\mathbf{c}_t$, the memory state of previous time stamp $\mathbf{c}_{t-1}$ is multiplied by $\mathbf{f}$, which has the effect of "forgetting" parts of the previous memory state that the model decided not to pay attention to at the current time step. The result of $\hat{\mathbf{c}} \odot \mathbf{i}$ is then added, which has the effect of only including those parts of the candidate memory state in the new memory state which were decided to be worth of including during the computation of the value of the input gate $\mathbf{i}$. A visualization of this process can be seen in Figure 2.16.

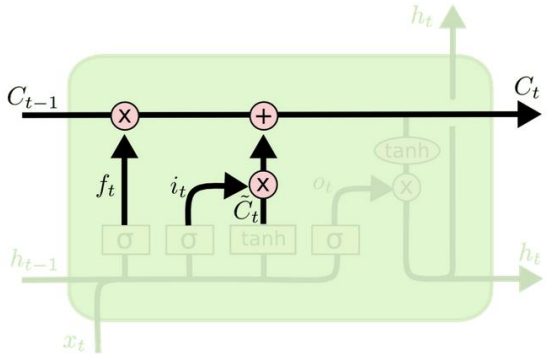

Figure 2.16: A visualization of the computation of the new memory state. Note that the uppercase $\hat{C}$ in the picture is functionally the same as $\hat{\mathbf{c}}$ we used in our descriptions. [Ola15]

Hidden state / output $\mathbf{h}_t$ At this point the model makes a decision on what to output (or what to put into its hidden state). This output is based on the new memory state for the given time step t defined in the previous paragraph. The tanh non-linearity

is first applied, which pushes the values of the new memory state into the range $[-1, 1]$. The result is then multiplied by the value of the output gate $\mathbf{o}$, which ensures that the output only contains parts of the new memory state filtered by the tanh activation function, which were decided to be of interest during computation of the output gate $\mathbf{o}$. A visualization of this process, along with the computation of the output gate $\mathbf{o}$ can be seen in Figure 2.17.

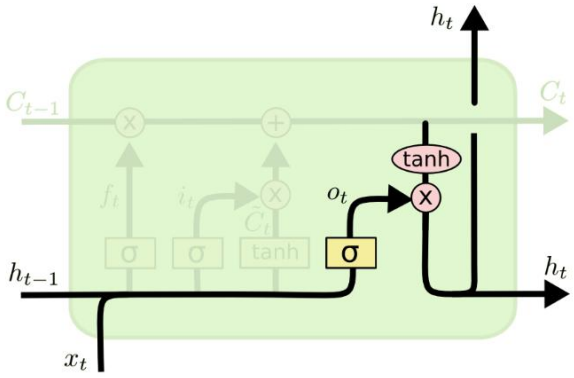

Figure 2.17: A visualization of the computation of the new memory state. Note that the uppercase $\hat{C}$ in the picture is functionally the same as $\hat{\mathbf{c}}$ we used in our descriptions. [Ola15]

The LSTM model is currently generally considered to be the best performing RNN architecture when it comes to applications of sequence modeling. This is especially in the NLP context, where it is responsible for many of the state-of-the-art results. Given its complexity, however, a set of alternative architectures have been proposed, the most promising of which we describe in the following sections.

2.3.4 GRU

As we can see from the extensive discussion above, the LSTM model is quite complex. This is not only problematic when it comes to analyzing its function but also practical applications, since the LSTM architecture is quite computationally expensive.

To remedy these issues a new architecture called Gated Recurrent Unit (GRU) has been proposed in [CVMG+14]. Compared to LSTM, this architecture also makes use of gating mechanisms, while reducing the number of gates and the need for a separate memory component. The full definition of the architecture can be found below:

$$\begin{aligned}
\mathbf{s}_t = R_{GRU}(\mathbf{s}_{t-1}, \mathbf{x}_t) &= (\mathbf{1} - \mathbf{z}) \odot \mathbf{s}_{t-1} + \mathbf{z} \odot \hat{\mathbf{h}} \\
\hat{\mathbf{h}} &= tanh(\mathbf{x}_t \mathbf{W}^{xh} + (\mathbf{h}_{t-1} \odot \mathbf{r}) \mathbf{W}^{hh}) \\
\mathbf{z} &= \sigma(\mathbf{x}_t \mathbf{W}^{xz} + \mathbf{h}_{t-1} \mathbf{W}^{hz}) \\
\mathbf{r} &= \sigma(\mathbf{x}_t \mathbf{W}^{xr} + \mathbf{h}_{t-1} \mathbf{W}^{hr}) \\
\mathbf{y}_t = O_{GRU}(\mathbf{s}_t) &= \mathbf{s}_t
\end{aligned}$$

where $\mathbf{x}_t \in \mathbb{R}^{d_{in}}$, $\mathbf{W}^{x\square} \in \mathbb{R}^{d_{in} \times d_{out}}$, $\mathbf{W}^{h\square} \in \mathbb{R}^{d_{out} \times d_{out}}$ and furthermore $\mathbf{z}, \mathbf{r}, \mathbf{s}_t, \hat{\mathbf{h}} \in \mathbb{R}^{d_{out}}$. The crucial part of this architecture is the $\mathbf{r}$ gate which controls how much of the previous state's information gets incorporated into the proposed state update vector $\tilde{\mathbf{h}}$. The value of the current hidden state is then a combination of the previous state and the proposed state update vector where the proportion of either of these is determined by the value of $(\mathbf{1} - \mathbf{z})$ and $\mathbf{z}$, respectively. We can note that while LSTM used two gates to control the computation of a new state, the GRU architecture can make do with just one gate $\mathbf{z}$, replacing the other with $\mathbf{z}$'s inverse. A visualization of this architecture can be seen in Figure 2.18.

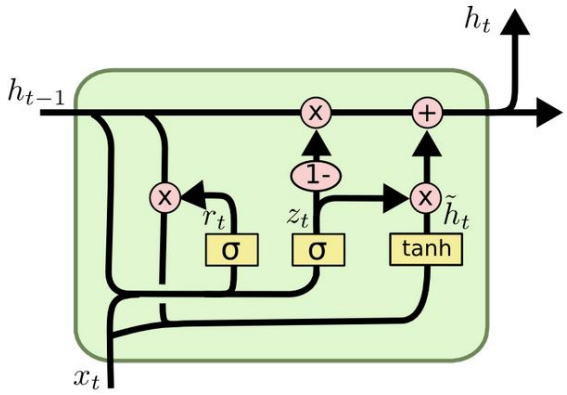

Figure 2.18: A visualization of the GRU architecture. . [Ola15]

Given the simpler nature of the GRU model, an obvious question to ask is whether there is any price to be payed in terms of performance. While there are tasks when one of these models is better than the other, it seems to be fair to say that their performance is quite similar, with GRU performing a bit worse in general (see [GSK+17] for more in-depth discussion on the topic). It is therefore of special interest when the size of the model plays a bigger role than its final performance.

In [JZS15] the authors tested more than ten thousand RNN architectures with the goal of finding one that would consistently outperform the LSTM architecture. Although they were able to find architectures that worked better than the LSTM architecture on some models, they failed to find one that would consistently beat the LSTM and the GRU.

Keeping this in mind, in the next section we describe Independent RNN, a new architecture specifically designed in order to model long term dependencies, which are often times found in natural language.

2.3.5 Independent RNN

The Independent RNN (IndRNN) architecture [LLC+18] has been recently introduced as an alternative to other RNN alternatives as LSTM and GRU, which have issues with long-term dependencies, and their complexity makes it difficult to interpret their behaviour. The IndRNN can be described in a formalism we utilized so far as:

$$\mathbf{s}_i = R_{IndRNN}(\mathbf{s}_{i-1}, \mathbf{x}_i) = g(\mathbf{x}_i \mathbf{W}^x + \mathbf{s}_{i-1} \odot \mathbf{u} + \mathbf{b})$$
$$\mathbf{y}_i = O_{IndRNN}(\mathbf{s}_i) = \mathbf{s}_i$$

where $\mathbf{x}_t \in \mathbb{R}^{d_{in}}$, $\mathbf{W}^x \in \mathbb{R}^{d_{in} \times d_{out}}$, $\mathbf{s}_i, \mathbf{u}, \mathbf{b} \in \mathbb{R}^{d_{out}}$ and $\odot$ represents piece-wise multiplication. Although the definition of this architecture is very simple and surprisingly similar to the SimpleRNN described above, the authors provide theoretical background as to why does this formulation lead to robust training, while also pointing out how can this model be effectively regulated, provided a suitable non-linearity g is used (the authors suggest ReLU or tanh). Furthermore, the authors evaluate this architecture on Natural Language Processing related tasks, such as Language Modeling, where this model obtained the best results out of all of the considered RNN models.

2.4 Training of Neural Networks

Up until now we treated the described Neural Network architectures as if their parameters were already set. In this section we discuss how can these parameters be found as well as other matters related to effective training of Neural Networks.

In essence, all of the training algorithms described below work by trying to minimize a so called loss function over a specific dataset. They do so using a gradient-based method in which they estimate the error over the dataset, compute the gradient with respect to the error and move the parameters in the direction of the gradient. The specifics of the respective algorithms are discussed in the following sections.

2.4.1 Loss Functions

The loss functions are an important part of Neural Network training regimes, as they provide the numerical values which are then back-propagated truth the Neural Network structure. In essence a loss function $L(\hat{\mathbf{y}}, \mathbf{y})$ returns a number which says how much

was the model wrong when it predicted $\hat{\mathbf{y}}$ when the ground truth output was $\mathbf{y}$. In our experiments we mostly use a specific example of the loss called the categorical cross-entropy loss.

Categorical cross-entropy loss

The aim of the categorical cross-entropy loss is to measure the difference in similarity between the distribution of ground truth $\mathbf{y}$ and predictions $\hat{\mathbf{y}}$. It is particularly useful when a probabilistic interpretation of the output of a network is desired. It is defined as the cross entropy between the two distributions:

$$L_{cross-entropy}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log(\hat{y}_i)$$

Note that this loss expects the outputs of the network to be transformed using the softmax transformation described in Section 2.1.4 in order for it to have probabilistic interpretation. This loss function can be used not only to train the network to produce the one best output, but also to return the k best candidate outputs.

2.4.2 Variants of Gradient Descent

The Gradient Descent algorithm [LBOM98] can generally be found in three variants. They differ in how much data is used for estimating the error – effectively trading between accuracy of parameter updates and training speed.

Batch Gradient Descent

The Batch Gradient Descent, also known as Vanilla Gradient Descent, uses the full dataset to compute the gradient of the cost function J with regards to the parameters θ:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

where η denotes the learning rate, the size of the step we take in the direction of the gradient.

The principal problem of this update method is the fact that going through the whole dataset is required for the computation of the gradient. It can therefore be quite slow, especially for large datasets. Furthermore, it may also not be feasible to use this method when the size of the dataset is larger than the size of the available memory. On the plus side, however, as the authors of [Rud16] note, it "is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surface".

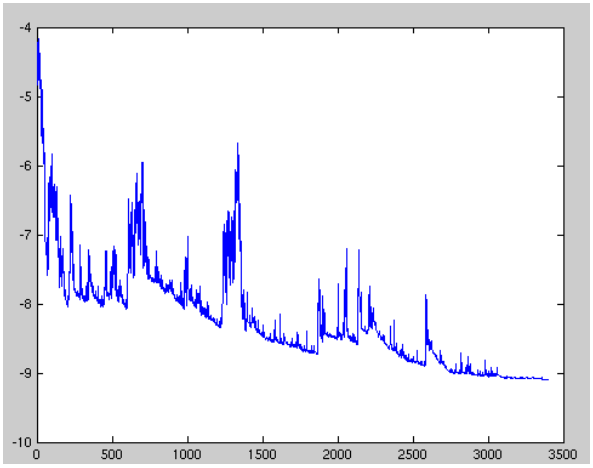

Figure 2.19: A visualization of fluctuation during training using the SGD algorithm. [Com17]

Stochastic Gradient Descent

Compared to Batch Gradient Descent the Stochastic Gradient Descent (SGD) does the other extreme: it computes the gradient using just one training pair $(x^{(i)}; y^{(i)})$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Such a formulation no longer suffers from memory issues when it comes to large datasets and is usually much faster. Since the steps in the direction of the gradient are taken very frequently (and sometimes redundantly), the training process can fluctuate a lot in terms of the loss function at a given point in training, as shown in Figure 2.19.

Minibatch Gradient Descent

The Minibatch Gradient Descent combines both of the previously described algorithms by computing the gradient using a mini-batch of n training pairs:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

This has the benefit of reducing the variance of SGD and also allows for an efficient implementation using highly optimized matrix optimizations that are available in many Deep Learning frameworks.

When it comes to choosing the mini-batch size, common choices range from 50 to 256, but larger sizes can also be used.

Momentum

One of the biggest problems of the SGD algorithm is that in parts of the error surface where it curves disproportionally more in mode dimension than any other the algorithm

tends to oscillate along these surfaces and only very slowly move towards the local optimum.

One way of remedying that behaviour is the use of momentum [Qia99] – essentially accumulating a small fraction of previous gradient along with the one computed for the current time step:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

where the default value for the momentum term η is usually 0.9.

The function of this model can be compared to pushing a ball down a hill. Apart from being pushed, it also gathers acceleration over time. However, such a ball would have a hard time stopping in front of another hill on the other side of the valley.

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient [Nes83] provides a way for the algorithm to be able to anticipate any slopes it may run into along the way and thus be able to avoid them.

Using the momentum formulation above, we can see that at once we have the value of v_t, the parameters θ will be decreased by γv_{t-1}. The term $\theta - \gamma v_{t-1}$ then gives us an approximation of what the derivative will look like after the current momentum update. We can use this notion to take the gradient with regards to this "approximation of the future" which provides us with a sort of "anticipatory capability":

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

where the momentum term η is usually also set to the standard value of 0.9.

2.4.3 Adaptive Learning Rate Algorithms

A common problem with all of the training algorithms we mentioned up until now is that they require the learning rate γ, a parameter which affects the training to a great extend, to be set manually. Furthermore, this parameter is shared for all of the considered parameters, regardless of their magnitude. These issues are addressed by a class of algorithms that make use of an adaptive learning rate.

Adagrad

Adagrad [DHS11] tries to use a separate learning rate for each parameter. It does so by keeping track of squares of gradients with regards to each specific parameter

during training. Let us denote $G_t \in \mathbb{R}^{d \times d}$, where the diagonal item i, i is the sum of the aforementioned squares up until the t-th training step and $g_{t,i} = \nabla_\theta J(\theta_{t,i})$. The parameter update equation then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

where ϵ is a very small number (for instance 10^{-8}) that ensures that division by zero issues are generally avoided. As the learning rate η no longer needs to be manually optimized, it is usually set to a default value (such as 0.01) and not optimized further.

Adadelta

Despite being able to set specific weights per each considered parameter, the sum of squares in the denominator of Adagrad turns out to be problematic, as this number just increases with further training steps. With enough training steps this results in a learning rate so small that no learning takes place.

Instead of keeping the sum of squares for the whole training sequence, Adadelta [Zei12] only keeps track of a pre-defined number of training steps. Furthermore, it does so by keeping the exponentially decaying average rather than a sum:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

where γ would be once again set similarly as the momentum term to a number close to 0.9. For clarity let us further rewrite the SGD update rule as:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The Adagrad update would the be:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

Using the exponentially decaying average described above we get

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Note that in the denominator the Root Mean Squared (RMS) error of the gradient is actually computed. We can therefore instead write:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

The authors in [Zei12] note that the imaginary units of parameters and the result of this parameter update rule differ[4]. To fix this, they define another exponentially decaying average:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

Its RMS can then be defined as

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

At time step t, however, $RMS[\Delta\theta]_t$ is not available yet, and thus it needs to be approximated with $RMS[\Delta\theta]_{t-1}$. Using this term instead of the learning rate γ gives us the Adadelta update equation:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adam

The Adaptive Moment Estimator (Adam) [KB14] also adaptively sets the learning rate for each parameters. It also keeps the exponentially decaying average of past squared gradients (denoted v_t in this case). It also keeps the exponentially decaying average of past gradients, which is very similar to what momentum does (this exponentially decaying average will be fittingly denoted m_t):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

As these decaying averages are initialized as zero-filled vectors, the authors note that they are biased towards zero. In further computations they therefore use the bias-corrected versions of these estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Using these values in the update rule similarly to the Adadelta's one gives us the Adam update rule:

[4]Note that the situation is the same in case of SGD, Momentum or Adagrad

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

The β_1 and β_2 parameters are generally set to the values proposed by the authors in [KB14]: 0.9 and 0.999, respectively.

Despite the introduction of various training algorithms and the benefits of those that make use of various regimes for adapting the learning rate, it is not always clear which one to choose. As a default choice, the Adam algorithm is usually used, but it may be worthwhile to try out other options, as many published works arrive at their results using just the vanilla SGD.

2.4.4 Training RNNs

While the methods mentioned in this section are general and should work for any Neural Network architecture, there are a few peculiarities worth discussing when dealing with training of Recurrent Neural Networks. In this section we discuss two of those: the so called Backpropagation Through Time and the concept of Teacher Forcing.

Backpropagation Through Time

Given their recursive definition, it may not be straightforward to see how can the RNNs be trained using the procedures based on gradient descent which were described above. By taking a closer look at Section 2.3.1 and especially at Figure 2.10, we can see that once unrolled, they are just a very deep instance of a Neural Network, onto which the gradient back-propagation algorithms described above can be applied.

Hence in order to train an RNN the following steps need to be followed:

1. Unroll the RNN in question along the provided input sequence.
2. Compute the loss from the point when error signal is available.
3. Back-propagate the error signal through the unrolled network.

In the literature this procedure is called back-propagation through time [Wer90].

Teacher Forcing

One of the core features of RNNs is the fact that they can process sequences, without placing constraints on their origin or structure, provided the network can consume them in an appropriate way. Let us consider Language Modeling for example, where the task is to predict the next symbol (word) provided a sequence of previous symbols (usually the sequence of past words of a given length). Suppose further that we would

like to train this model to finish a sentence, provided it receives a few words from its beginning.

A simple architecture of a model dealing with this problem may be one in which a specific number of words (context) is passed to the network and it is trained to predict the next word. During train time, this process is obvious, as the ground truth contexts are available from the training data. When we consider the test time, however, the situation is a bit more problematic, as the true context is not directly available. A common way of approximating the true context is to use the previous output of the network. Put differently, at train time the network learns to make decisions from input that is guaranteed to be "correct" (since it comes from the training set), while at test time it makes decisions based on its own output in the past. This concept is called *teacher forcing* (inspired by a real-world metaphor) and is visually illustrated in Figure 2.20.

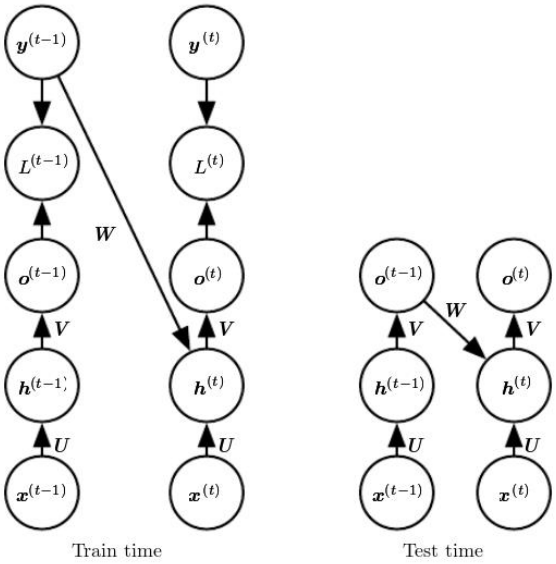

Figure 2.20: A visualization of the concept of teacher forcing. During train time (left) the ground truth $\mathbf{y}^{(t-1)}$ is provided to the network as an input, whereas at test time (right), it receives its own past output $\mathbf{o}^{(t-1)}$ as input. [GBCB16]

This approach is still quite problematic in practice, as during test time an error at a single time step may affect all future predictions. As a results the authors of [BVJS15] propose a scheduled sampling approach, in which the teacher forcing style of training alternates with training that is similar to what happens during test time (the network is tasked to learn to make good predictions even when its own previous outputs are provided as input) according to a decay schedule.

2.4.5 Other considerations

When it comes to Neural Network training, there are a few concepts that are necessary for it to be successful or actually viable. The following sections provide a short descriptions of some of these concepts.

Initialization

As the authors of [Gol15] note, "the magnitude of the random values has an important effect on the success of training." A popular scheme for initialization was introduced in [GB10]. It is known as *Xavier initialization* after the first name of the first author of that work. It proposes to initialize the weight matrix $\mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}$ in the following way:

$$\mathbf{W} \sim U[-\frac{\sqrt{6}}{\sqrt{d_{in} + d_{out}}}, \frac{\sqrt{6}}{\sqrt{d_{in} + d_{out}}}]$$

where $U[a, b]$ denotes a randomly sampled value from the uniform distribution in the range $[a, b]$.

Shuffling

Given the nature of the discussed training algorithms, the order in which the network receive its training examples has a potential to hamper its overall performance (by for instance only focusing on important features from the first parts of the training set and discarding all the others). It is therefore advised to shuffle the data before providing it to the trained model.

Vanishing and Exploding Gradient

When dealing with deep networks during training, it is not uncommon for their gradient to either vanish (become very small) or explode (become very large). The issue of vanishing gradient has motivated the introduction of new models specifically designed to counter this problem by assisting the gradient flow, such as the LSTM or GRU models described in Section 2.3.3 and Section 2.3.4.

For dealing with exploding gradient a simple solution has been proposed in [PMB12]: if the L_2 norm of the gradients of all parameters in the network $\hat{\mathbf{g}}$ becomes higher than a threshold, the following update is to be performed:

$$\hat{\mathbf{g}} = \frac{threshold}{||\hat{\mathbf{g}}||}$$

Regularization

Since deep neural networks have a lot of parameters, they are prone to overfitting the training data. A simple but very effective and popular method called *dropout* proposed in [SHK+14] provides a way of regularizing the network by randomly setting a proportion of neurons in the network (or a specific layer) to 0. This has the effect of forcing the network not to rely on specific weights and can also be interpreted as modeling an "ensemble of experts" inside a single network. This technique is often described as one of the key factors in Deep Neural Network improvements on massive image datasets [KSH12], especially when combined with the ReLU non-linearity [DSH13]. It has, however, been used with great success in Natural Language Processing settings as well, such as for instance in [IC14].

Chapter 3

Deep Neural Networks for Diacritics Restoration

3.1 Existing Deep Neural Network Architectures for Diacritics Restoration

The historically first work that utilized a Deep Neural Network for Diacritics Restoration [RASRR14], which made use of Deep Belief Networks [Hin09], a predecessor of deep architectures that are popular today. In this work the author's approach included processing a context of words as an input, while having separate modules for extracting features form this input (such as for instance part-of-speech tags) which were then merged together for the final prediction.

A different approach can be found in [BG15], where the author considers character-level features and uses a Deep Neural Network architecture composed of embedding layer, a set of bidirectional LSTM layers, followed by a softmax layer to produce the final classification output. A visualization of this architecture can be seen in Figure 3.1.

A similar approach can also be found in [Náp17], where the author used both character-based and word-based, as well as translation models for the task of diacritics restoration. The character-based models presented in this work differ from the previously mentioned work mostly by modeling the task as a sequence labeling problem, in which a model maps a sequence of input symbols to the same number of output symbols.

In our work we decided to focus on character-based models for this task, which are attractive mostly thanks to their simplicity and size. Furthermore, we consider the task to be a sequence classification problem, in which the model produces a single output (in our case a character) for a provided input sequence.

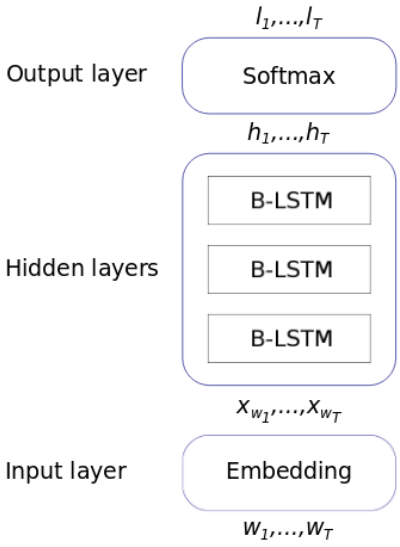

Figure 3.1: An illustration of the topology of the network used in [BG15]

3.2 Our Proposed Model

Our proposed model can be viewed as an encoder-decoder architecture, which is jointly trained to predict the correct character, whose latinized version is included in the center of the input sequence, with the other symbols in the sequence serving as its context. The Encoder and Decoder parts of this model are described in greater detail in the following sections.

3.2.1 Encoder architecture

The encoder part of the model first receives the input sequence as a sequence of numbers (IDs) which represent the respective input characters. These IDs are then passed through an embedding layer described in Section 2.2.3 that produces a specific embedding for each given ID. These embeddings are then processed by an RNN (as introduce in Section 2.3.1), which produce an output for every input of the input sequence. These outputs provide an encoded representation of the model's input, which is then decoded by the next part of the model.

3.2.2 Decoder architecture

The decoder architecture receives the encoding of created in the previous part of the model and is tasked with generating a prediction off it. The principal task of the decoder therefore is to transform the output in such way that the scores for each of the considered output class can be computed and then transformed using the softmax

output transformation described in Section 2.1.4. In general, this is done by combining the encoded input sequence into an intermediary representation, onto which the tanh non-linearity is then applied and the output is further transformed by a feed-forward layer, which outputs class scores, that are transformed by the softmax output transformation.

As the intermediary representation can be created in multiple ways, we describe each one in greater detail in the following paragraphs.

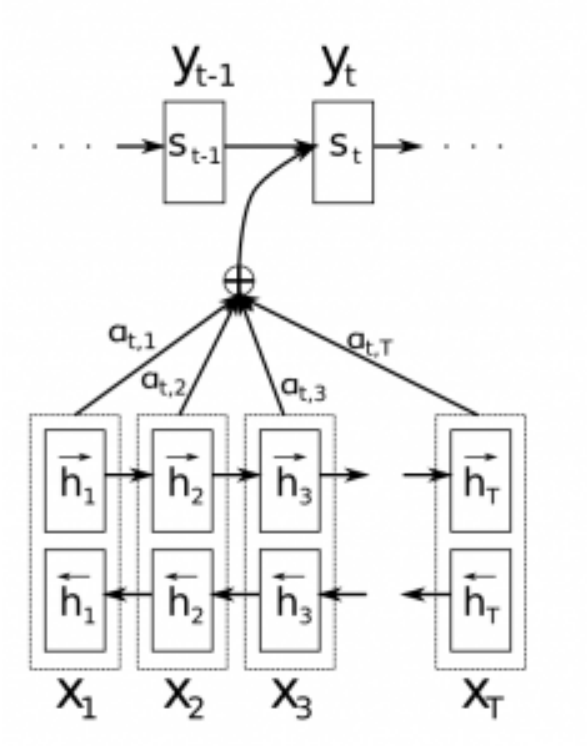

Figure 3.2: An illustration of the attention mechanism [BCB14].

"Center only" This representation takes the whole encoded input $\mathbf{E}_i \in \mathbb{R}^{n \times d_{enc}}$ where d_{enc} is the dimensionality of the output of the encoder's RNN and outputs just the encoding that represents the central symbol of the encoded input, that is the encoding that belongs to the symbol at the $\lfloor \frac{n}{2} \rfloor + 1$ place in the input.

"Flatten" This representation takes the whole encoded input $\mathbf{E}_i \in \mathbb{R}^{n \times d_{enc}}$ and outputs a single "flattened" vector $\mathbf{e} \in \mathbb{R}^{n \cdot d_{enc}}$.

"Attention" The attention mechanism also receives the whole encoded input $\mathbf{E}_i \in \mathbb{R}^{n \times d_{enc}}$. It then first extracts the central part of the encoded input as described in the paragraph above. Using this paragraph a so called attention is computed over the encoded input. This is done by using the central part of the encoded input as an

input to a small feed-forward neural network, which outputs a probability for each of the n dimensions of the encoded input. Given this "weight matrix", the encoded input $\mathbf{E}_i \in \mathbb{R}^{n \times d_{enc}}$ is summed into a single vector $\mathbf{c} \in \mathbb{R}^{d_{enc}}$. The small network that provides scores for each of the encoded input symbol representations is then jointly trained with other parts of the model.

This concept is called "attention" and was introduced in [BCB14] and was responsible for great advances in the filed of Neural Machine Translation. A visualization of this concept can be seen in Figure 3.2.

3.2.3 Implementation

The proposed model described in the previous section has been implemented using the PyTorch [PGCC17] framework in the Python programming language [S+99]. Most of the previously discussed and described components, such as the loss function, the Adam optimizer, the fully connected layer and the SimpleRNN, LSTM and GRU modules were implemented using the stock implementation provided by the PyTorch framework. A notable difference is the IndRNN architecture, which has been adopted from the implementation located at `https://github.com/StefOe/indrnn-pytorch` to which we added the bidirectional option, which this implementation did not provide.

In order to keep track of various experiments the `sacred` framework[1] has been used. Among other things, it provides an easy way for specifying hyper parameters of the model from the command line. For visualization during training we used the TensorBoard project, which curiously enough comes from a "competing" framework called TensorFlow [ABC+16], but can also be used without it thanks to a project called `tensorboardX`[2]. For evaluation of the considered models we used helper functions from the `scikit-learn` library [PVG+11] and for confusion matrix visualization we used the `pandas-ml` library[3]. The implementation is available online at `https://github.com/mrshu/diaqres`.

3.3 Evaluation

3.3.1 Datasets

To evaluate our proposed approach we used two datasets comprised of Slovak texts: a Wikipedia dataset which was primarily used in training, and the Digital Corpus of the European Parliament, which was mostly used for testing the generalization capability of the models we trained.

[1] `https://github.com/IDSIA/sacred`

[2] `https://github.com/lanpa/tensorboard-pytorch`

[3] `https://github.com/pandas-ml/pandas-ml`

Wikipedia dataset

The Wikipedia dataset was obtained by from the Wikipedia dump service on the 20th of January, 2018[4]. The Wikipedia dump, however, does not include only Natural Language, but also other information in Wikipedia Markup Language. In order for a dataset like this to be useful in a Natural Language Processing setting, this markup needs to be stripped.

This was done using a tool called `wiki2text`[5]. Furthermore, the text was lowercased, and all non-alphanumeric characters were removed. In the end this yields a dataset of 35 613 022 words, out of which 1 194 781 are unique.

DCEP: Digital Corpus of the European Parliament

The second dataset we used was the Slovak part of the Digital Corpus of the European Parliament [HKV+14]. This dataset consists of texts produced at the European Parliament which are mostly of administrative nature. The Slovak part of this dataset consists of 21 841 documents, 4 281 697 sentences, 42 536 235 words and 713 273 unique words. While this dataset is most often used for Machine Translation tasks, its Slovak part provides an interesting dataset for the task of diacritics restoration as well.

When using this dataset for testing, it has been transformed using a similar cleaning procedure as was described in the case of the Wikipedia dataset. For the purpose of model training these datasets were split in 70:30 ratio between the training and the test sets.

3.3.2 Preprocessing, Training and Testing Regimes

Preprocessing

While the input data has been cleaned to a great extend, the input text can still contain characters that are not part of the Slovak alphabet. The preprocessing step therefore replaces these characters with a special new character denoted `<unk>`.

Furthermore, the preprocessing step is responsible for reading in the text and creating training pairs for input windows of size n. Depending on whether teacher forcing is to be applied or not, the input part of the pair is stripped of diacritics mark, whereas the character (with diacritics) in the center of the window is assigned as the output part of the training pair. It therefore follows that n needs to be an odd integer.

[4] `https://dumps.wikimedia.org/skwiki/20180201/`

[5] Which can be obtained from `https://github.com/rspeer/wiki2text`

Training

During one training step a mini-batch of size m is passed through the network. A categorical cross-entropy loss (described in Section 2.4.1) is applied to predicted and true outputs, and this loss is then optimized using the Adam optimizer (introduced in Section 2.4.3). By default, the models were set up to be trained for three epochs (three passes through the input data), but their training was also stopped early, when their loss no longer seemed to improve. The learning rate was set to 0.002, while the weight decay (which can be interpreted as the L_2 penalty) has been set to 1.2×10^{-6}. At each training step the weights were clipped (as described in Section 2.4.5) at 5.0.

We used mini-batches of size $m \in \{100, 128, 200, 500, 800, 1000, 1200, 2000, 8000\}$, mostly to balance out the size of the network with the throughout speed. As the extremely large mini-batch size was only tested in case of a very small model and not in the other cases[6], we were unable to confirm the conclusions of [KMN+16], where the authors conclude that large mini-batches lead to poor convergence.

All of the models were trained on an NVIDIA GeForce GTX 1080 GPU. The training time stemmed from a few hours to more than a week.

Testing

During testing, the same preprocessing procedure has been applied to the test data. A mini-batch of m input windows of size n was then provided to the network as an input. For each predicted distribution over all possible output characters the most probable prediction was taken as the predicted output.

In case of a model for which the teacher forcing type of training has been considered, for the first $\lfloor \frac{n}{2} \rfloor$ training steps we proceed in the same way as described above. In the following steps the first $\lfloor \frac{n}{2} \rfloor$ inputs in the input window are replaced with already predicted values from the model.

3.3.3 Evaluation Metrics

For evaluation, we used the F1 metric described in Section 1.1.2. It is defined as:

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

where *precision* and *recall* are defined as

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

[6]The size of the model would prevent that, as it would not fit into the GPU memory.

Since we operate on character level, *tp* in for each character means that the predicted character was indeed the correct one, *fp* means that this character was predicted, even though a different character was the correct one and *fn* means that this character was not predicted, even though it was the correct one according to our gold truth. This metrics were computed for each considered output class (output character). To compute a score for a given model the metric scores for each class were averaged, while each class was weighted by its support (the number of true instances for a given class).

3.3.4 Results

In this section we present the results of our evaluation. In Table 3.1 we can see the respective hyper parameters of our model, along with their possible values. Since testing every single combination would require us to try 5760 combinations (which would be computationally intractable in our conditions), we decided to test the effect of each of the hyper parameters by changing just one of them while keeping the others fixed. This approach is to some extend warranted by [GSK+17], where the authors conclude that the hyper parameters of LSTM-based models seem to be independent of one another.

hyperparameter	type	values
RNN type	categorical	RNN, LSTM, GRU, IndRNN
Use bidirectional model	boolean	True, False
RNN hidden layer dimension	integer	50, 100, 200, 500
Number of RNN layers	integer	1, 2, 3, 4, 6
Input embedding dimension	integer	50, 100, 200
Input length	integer	11, 21, 51, 101
Decoder type	choice	”central only”, ”flatten”, ”attention”

Table 3.1: A listing of our model's hyper parameters and their possible values.

Encoder Type

encoder type	precision	recall	F1 score
RNN	0.859	0.917	0.885
LSTM	0.856	0.919	0.884
GRU	0.861	0.918	0.886
IndRNN	**0.944**	**0.947**	**0.941**

Table 3.2: A listing of results of evaluation of encoder types.

As we can see in Table 3.2, the newly proposed IndRNN architecture performed the best compared to all of the other architectures, whose results are interestingly close to one another. We hypothesize that this may be due to the other parameters being set to values that were not favorable to the other architectures (namely input length was set to 51 characters, the networks used two layers, 200 neurons in their hidden layer and input embedding of 100 neurons).

Decoder Type

decoder type	precision	recall	F1 score
”central only”	0.942	0.945	0.938
”flatten”	0.976	0.976	0.976
”attention”	**0.978**	**0.977**	**0.977**

Table 3.3: A listing of results of evaluation of decoder types.

As the Table 3.3 shows, the ”attention” and ”flatten” types of decoder produced quite comparable results, with ”attention” being slightly better.

Input Length

input length	precision	recall	F1 score
11	0.963	0.963	0.958
21	0.964	**0.964**	0.960
51	**0.965**	**0.964**	**0.961**
101	0.963	0.963	0.959

Table 3.4: A listing of results of evaluation of input length.

The evaluation of input length, the result of which you can see in Table 3.4, shows that the differences between the respective input lengths are generally minimal. Furthermore, it seems to show that increasing the input length does help, but impact of this factor plateaus once the input is longer than 51 characters.

Use of Bidirectional Model

The general attitude towards the use of bi-directional models seems to be that they generally help improve performance of models that make use of them. As we can see in Table 3.5, we can confirm that in our case as well, noting that in our test the use of bi-directional approach in our model substantially helped to improve its performance.

direction	precision	recall	F1 score
single-directional model	0.942	0.945	0.938
bi-directional model	**0.972**	**0.971**	**0.971**

Table 3.5: A listing of results of evaluation of single-directional versus bi-directional model.

hyperparameter	type	value
RNN type	categorical	IndRNN
Use bidirectional model	boolean	True
RNN hidden layer dimension	integer	500
Number of RNN layers	integer	2
Input embedding dimension	integer	200
Input length	integer	51
Decoder type	choice	”attention”

Table 3.6: A listing of hyper parameters of our final model.

Final Model

We also evaluated other hyper parameters which are not discussed in further detail here. In general, however, we can note that adding further layers helped, although in the case of three and more layers the improvements were either negligible or the model's performance decreased. Increasing the model's hidden layer and embedding dimensions seems to help: in our tests the maximum values of 500 and 200 produced the best results. In the end, we arrived at a set of hyper parameters that can be seen in Table 3.6.

model	dataset	precision	recall	F1 score
baseline	Wikipedia	0.853	0.917	0.881
baseline	DCEP	0.851	0.917	0.881
our best model	Wikipedia	**0.987**	**0.989**	**0.988**
our best model	DCEP	**0.985**	**0.986**	**0.986**

Table 3.7: A listing of results of evaluation of our best model as compared to the baseline.

As we can see in Table 3.7, our model not only managed to produce performance which was significantly better than the baseline (a model that just copies its input to the output), but also managed to generalize well across various datasets, as suggested by the comparison on the Wikipedia and DCEP datasets. Let us stress out once again that the model did not see the data from the DCEP dataset during training at all.

3.3.5 Analysis

In order to better understand how does the model work and why does it fail to produce the correct prediction, we analyzed the cases in which it produced the wrong prediction compared to the test set. We did so by taking a closer look at the input to the model and also its predicted distribution over the possible output characters.

Our first observation is that in all considered cases the model correctly learned which latinized characters map to which character with a diacritic mark. In other words, when considering the ground truth could always be found in the top 3^7 most probable predictions. While this is not an impressive feat in and of itself (as a simple lookup table or an n-gram model might be easily able to do so as well), it suggests that the model did try to solve the task of diacritics restoration.

input	`chych kosti lebky pozri [s]ev anatomia zidia v s`
predicted distribution	`s`: 0.89946985, `š`: 0.10027383, `j`: 0.00010318483
true output	`š`

Table 3.8: First example of an error made by the model. The character that is to be predicted is emphasized in a small black box in the first row. The top 3 most probable choices from the output distribution are described in the second row.

Let us discuss some of the mistakes the model did at test time. Table 3.8 describes a situation in which the model mistakenly predicted `s` in the word `sev` (without specific meaning) whereas the correct prediction should have been `š` and the word would become `šev` (suture). One may argue that this case is quite ambiguous, as even a native speaker may wonder whether letters `sev` represent some sort of an abbreviation or `šev` (suture).

input	`metropolitnej oblasti gu[s] dan v izraeli v relativ`
predicted distribution	`s`: 0.8891944, `š`: 0.110016435, `o`: 0.00023678609
true output	`š`

Table 3.9: Second example of an error made by the model. The character that is to be predicted is emphasized in a small black box in the first row. The top 3 most probable choices from the output distribution are described in the second row.

In the example in Table 3.9 the situation is even more complicated, as the model is tasked with predicting whether the correct name of a metropolitan area in Israel is `gus dan` or `guš dan`. It seems that an ambiguity of this type would be very difficult to

[7]Note that the choice of top 3 stems from the fact that in Slovak the characters `a` and `o` can be mapped to two diacritized characters (that is `á`, `ä` and `ó`, `ô`, respectively) as we can also note in Figure 1.1. Since the model needs to consider the two diacritized options and the possibility that the character does not need to be diacritized, the top 3 most probable predictions need to be considered.

handle, even for a native speaker, as it requires knowledge of the name of a geographical location.

input	iniciativ sucasnej etiky [c]nosti konkretne jej aris
predicted distribution	č: 0.6697172, c: 0.33018324, ŕ: 1.5026837e-05
true output	c

Table 3.10: Third example of an error made by the model. The character that is to be predicted is emphasized in a small black box in the first row. The top 3 most probable choices from the output distribution are described in the second row.

The situation described in Table 3.10 shows a very intricate situation in which the model is tasked with deciding whether to use cnosť or čnosť. Both of these words mean virtue in Slovak and can be used interchangeably. We believe that we can conclude that disambiguating this situation is difficult even for native speakers, as we were able to find an instance of a question for the Ľ. Štúr Institute of Linguistics of Slovak Academy of Sciences, in which a (presumably) native speaker asked which of these two possible choices would be more fitting in the context of a bachelor thesis[8].

In the interest of fairness, however, let us stress out that these examples were cherry-picked, in order to better illustrate the difficulty these models run into when trying to restore diacritics. They do not represent the majority of cases, in which the prediction of the model was simply wrong without being ambiguous to the extend described in the presented examples.

[8] https://jazykovaporadna.sme.sk/q/291/

Visualization of Learned Embeddings Given the interpretable nature of word embeddings [TFD16], we visualized the weights using the t-SNE visualization technique [MH08]. The result can be seen in Figure 3.3.

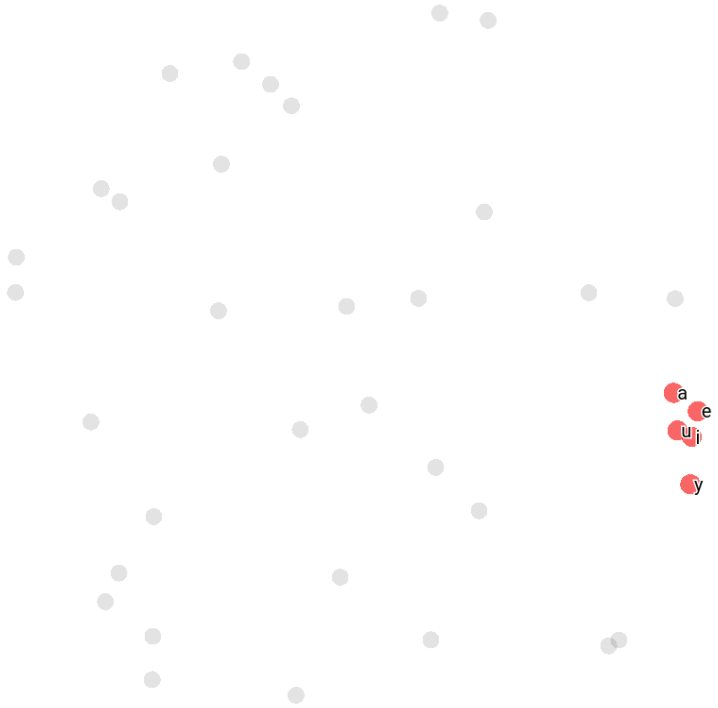

Figure 3.3: A visualization of the embedding layer learned as part of training the Diacritics Restoration model.

While the vowels a, e, i, y and u were found to be close to one another, the vowel o is notably missing. We also failed to find other examples of this phenomena (such as for instance numbers).

Conclusions

In this work we provided an introduction to the field of Diacritics Restoration through a discussion on published approaches that strive to deal with this problem. We further provided an extensive introduction to the generally used architectural parts of Neural Networks, with a special focus on Natural Language Processing. Finally, having analyzed previously used Neural Network architectures that were used for this task we introduced a new architecture and evaluated it on two datasets: the Slovak Wikipedia dataset and the Digital Corpus of the European Parliament.

Our best model obtained an F1 score of 0.988 on the Slovak Wikipedia dataset, an 88% improvement over the baseline. Furthermore, the same model has been tested on the Digital Corpus of the European Parliament, where it obtained a similar F1 score of 0.986, suggesting at it capability to generalize well across different text data domains. As part of this work we also performed a comparison of various hyper parameters and their effect at the performance of the model, as well as an analysis of the errors the model made, thus providing better insight with regards to its performance and viability.

One of the most principled critiques of this work can be made on the basis of the used encoder architecture. While various types of RNNs are certainly not a wrong choice in a case like ours, there are other architectures to choose from, most notably Convolutional Neural Networks (CNN). As the authors of a recent comparison study note in [BKK18]: "the preeminence enjoyed by recurrent networks in sequence modeling may be largely a vestige of history". It may therefore be worth further investigating these architectures, as well as the self-attention based Transformer architecture from an aptly named paper "Attention is all you need" [VSP+17], which recently achieved interesting results when applied in a diacritics restoration setting on world-level features for texts in Yoruba language [Ori18]. In general, especially given recent successes in Neural Machine Translation, it seems that it may be also worth investigating how much does rephrasing the task of diacritics restoration from a sequence labeling one to a machine translation one, in which the task would be to translate from distorted non-diacritized text into grammatically correct one. Preliminary results featured in [Náp17] and [PPLH17] seems to suggest that such a rephrasing may indeed be beneficial.

Furthermore, although we have investigated the effect of various hyper parameters on the performance of our proposed models, utilizing the assumption from [GSK+17],

whose authors found that the LSTM hyper parameters were for the most part independent from each other, this assumption may no longer hold in case of newly proposed models or more convoluted architectures we utilized. Although as we noted a grid search over the space of all hyper parameters may be computationally intractable in our setup, the notion of 'shooting blind' with the actual values of hyper parameters may be remedied to a great extend by randomly sampling from the space of hyper parameters, which would also give us a better notion of how their interaction affects the final performance. On a related note, given recent progress on the effect of batch size during training of Neural Networks (such as for instance [LLQL17] and [SKL17]), it may be worth treating the batch size as a hyper parameter and including it in the optimization process so that it is set in a principled way.

From the linguistics perspective, our approach at feeding the data into the model seems to be questionable at best. Since we just sequentially split the data into input windows, it can easily happen that the input sequence starts in the middle of the word or just at its final character. It would be prudent to see whether "better formed" input may have any effect on the final performance.

Finally, although the goal of this thesis was the evaluate the proposed model on Slovak texts, we only did that on Slovak texts that are of questionable quality (Wikipedia dataset for instance contains a fair amount of text in other languages) or on Slovak texts that come from a very specific domain (Digital Corpus of the European Parliament contains mostly texts of administrative nature, which in turn contains only a subset of natural language Slovak native speakers use in ordinary communication). It would therefore be of interest to test this model on Slovak texts that would be of better (grammatical) quality and would also better represent natural language in general use. Moreover, given the fact that the model was proposed with generality in mind and during its design no hard choices in terms of biasing it towards Slovak language were made, it would be interesting to evaluate it on texts written in other languages: all that would need to change would be definition of the considered alphabet. We suggest that these ideas be explored as part of future work.

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[AFNR12] Jordi Atserias, María Fuentes Fort, Rogelio Nazar, and Irene Renau. Spell checking in spanish: The case of diacritic accents. In *LREC*, pages 737–742, 2012.

[AGAS+15] Gheith A. Abandah, Alex Graves, Balkees Al-Shagoor, Alaa Arabiyat, Fuad T. Jamour, and Majid A. Al-Taee. Automatic diacritization of arabic text using recurrent neural networks. *International Journal on Document Analysis and Recognition (IJDAR)*, 18:183–197, 2015.

[AH16] Judit Acs and József Halmi. Hunaccent: Small footprint diacritic restoration for social media. In *Normalisation and Analysis of Social Media Texts (NormSoMe) Workshop Programme*, page 1, 2016.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[Ben09] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[BG15] Yonatan Belinkov and James Glass. Arabic diacritization with recurrent neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2281–2285, 2015.

[BHM+16] Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances In Neural Information Processing Systems*, pages 4331–4339, 2016.

[BHR14] Nicolas Brunel, Vincent Hakim, and Magnus JE Richardson. Single neuron dynamics and computation. *Current opinion in neurobiology*, 25:149–155, 2014.

[BKK18] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[BM12] Herve A Bourlard and Nelson Morgan. *Connectionist speech recognition: a hybrid approach*, volume 247. Springer Science & Business Media, 2012.

[BVJS15] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[CK11] John Cocks and Te Taka Keegan. A word-based approach for diacritic restoration in māori. In *Australasian Language Technology Association Workshop 2011*, page 126, 2011.

[CM14] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.

[Com17] Wikimedia Commons. File:stogra.png — wikimedia commons, the free media repository, 2017. [Online; accessed 2-May-2018].

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[DPWDS07] Guy De Pauw, Peter W Wagacha, and Gilles-Maurice De Schryver. Automatic diacritic restoration for resource-scarce languages. In *International Conference on Text, Speech and Dialogue*, pages 170–179. Springer, 2007.

[DSH13] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In

Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pages 8609–8613. IEEE, 2013.

[Elm90] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GBCB16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[Gol15] Yoav Goldberg. A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*, 2015.

[GSK+17] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.

[HDY+12] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[Hif12] Yasser Hifny. Smoothing techniques for arabic diacritics restoration. In *12th Conference on Language Engineering*, pages 6–12, 2012.

[Hin09] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[HKV+14] Najeh Hajlaoui, David Kolovratnik, Jaakko Väyrynen, Ralf Steinberger, and Daniel Varga. Dcep-digital corpus of the european parliament. In *LREC*, pages 3164–3171, 2014.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[IC14] Ozan Irsoy and Claire Cardie. Deep recursive neural networks for compositionality in language. In *Advances in neural information processing systems*, pages 2096–2104, 2014.

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-05-16].

[JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350, 2015.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KDDV17] Jurgita Kapočiūtė-Dzikienė, Andrius Davidsonas, and Aušra Vidugirienė. Character-based machine learning vs. language modeling for diacritics restoration. *Information Technology And Control*, 46(4):508–520, 2017.

[KMN+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LBOM98] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 1998.

[LEF16] Nikola Ljubešic, Tomaz Erjavec, and Darja Fišer. Corpus-based diacritic restoration for south slavic languages. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016). European Language Resources Association (ELRA)(may 2016)*, 2016.

[LH05] Michael London and Michael Häusser. Dendritic computation. *Annu. Rev. Neurosci.*, 28:503–532, 2005.

[LKJ15] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Cs231n: Convolutional neural networks for visual recognition. *University Lecture*, 2015.

[LLC+18] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. *arXiv preprint arXiv:1803.04831*, 2018.

[LLQL17] Chris Junchi Li, Lei Li, Junyang Qian, and Jian-Guo Liu. Batch size matters: A diffusion approximation framework on nonconvex stochastic gradient descent. *arXiv preprint arXiv:1705.07562*, 2017.

[MH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[MKB+10] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[MN02] Rada Mihalcea and Vivi Nastase. Letter level learning for language independent diacritics restoration. In *proceedings of the 6th conference on Natural language learning-Volume 20*, pages 1–7. Association for Computational Linguistics, 2002.

[Náp17] Jakub Náplava. Natural language correction. 2017.

[Nes83] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[Ola15] Christopher Olah. Understanding lstm networks, 2015. *URL https://colah. github. io/posts/2015-08-Understanding-LSTMs*, 2015.

[Ori18] Iroro Orife. Attentive sequence-to-sequence learning for diacritic restoration of yor\ub\'a language text. *arXiv preprint arXiv:1804.00832*, 2018.

[PGC15] Wenzhe Pei, Tao Ge, and Baobao Chang. An effective neural network model for graph-based dependency parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 313–322, 2015.

[PGCC17] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2017.

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

[PPLH17] Thai-Hoang Pham, Xuan-Khoai Pham, and Phuong Le-Hong. On the use of machine translation-based approaches for vietnamese diacritic restoration. *arXiv preprint arXiv:1709.07104*, 2017.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[RASRR14] Mohsen AA Rashwan, Ahmad A Al Sallab, Hazem M Raafat, and Ahmed Rafea. Automatic arabic diacritics restoration based on deep nets. *ANLP 2014*, page 65, 2014.

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[S+99] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[SD01] Michel Simard and Alexandre Deslauriers. Real-time automatic insertion of accents in french text. *Natural Language Engineering*, 7(02):143–165, 2001.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SKL17] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[TFD16] Yulia Tsvetkov, Manaal Faruqui, and Chris Dyer. Correlation-based intrinsic evaluation of word vector representations. *arXiv preprint arXiv:1606.06710*, 2016.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[TP89] David S Touretzky and Dean A Pomerleau. What's hidden in the hidden layers. *Byte*, 14(8):227–233, 1989.

[üAE14] K übra Adalı and Gülsen Eryigit. Vowel and diacritic restoration for social media texts. In *Proceedings of the 5th Workshop on Language Analysis for Social Media (LASM)@ EACL*, pages 53–61, 2014.

[UBP+08] Cătălin Ungurean, Dragoş Burileanu, Vladimir Popescu, Cristian Negrescu, and Aurelian Dervis. Automatic diacritic restoration for a tts-based e-mail reader application. *UPB Scientific Bulletin, Series C*, 70(4):3–12, 2008.

[VDWCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

[Wer90] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[Yar99] David Yarowsky. A comparison of corpus-based techniques for restoring accents in spanish and french text. In *Natural language processing using very large corpora*, pages 99–120. Springer, 1999.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.