Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA
Master Degree in Data Science

Deep Learning for Text Classification: an application of Generalized Language Models for Italian texts

Candidate:
Davide Parolo
Matricola 1178800

Supervisor:
Alessandro Sperduti

Co-Supervisor:
Michele Gabusi

Anno Accademico 2019-2020

Contents

Introduction

In the present work the main focus is solving the Text Classification (TC) problem using Deep Learning (DL) methods. The experiments will deal with the classification of Italian texts using different techiques to compute words embedding representations, including one coming from a Generalized Language Model (GLM).

Text Classification is one of the most studied problem in the Natural Language Processing (NLP) field. It deals with labeling texts with categories that can come from a set of multiple alternatives. Once some data preprocessing techniques are applied to the texts, deep learning models can be used to perform this task. Some data preparation operations are essential for NLP tasks in order to make the model work in a desirable way. Language data are discrete symbols and they need to be converted to numeric vectors in some useful way.

Deep Learning is part of a broader family of machine learning methods based on neural networks. Many deep learning models have been developed in the past to solve NLP tasks. In particular, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) can be used to solve the Text Classification task, since they capture local structures and long range dependencies in sequences of data respectively. Furthermore, various models based on neural networks have been developed to generate word embedding representations from large corpus of texts.

This work focuses on the word embedding technique, which consists in mapping words to feature vectors of fixed dimensions. The standard procedure in NLP tasks is to rely on pre-trained algorithms to produce word vectors. These are the vectors that are fed to the model which is trained to solve the NLP task. Two different word embedding algorithms will be explained: Word2Vec and ELMo. The alternative to the standard approach is to tune automatically the word vectors during the neural network training for the NLP task. In this work, the two approaches are presented and compared in details.

The experiments that will be presented have been carried out on a real-

world dataset. Different deep learning models have been applied to solve a text classification task on an Italian dataset. The Word2Vec and the ELMo algorithms are both used to compute pre-trained word representations, and this allowed to compare their performances on the considered task. The alternative of automatically tuning the word vectors has also been applied. Both Recurrent Neural Networks and Convolutional Neural Networks are considered as approaches to capture significant text features.

The structure of this work is presented as follows. Chapter 1 will focuse on the text classification problem and on some basic operations to process and transform the texts in order to let deep learning methods extract useful information from them. Chapter 2 will give an overview on the deep learning approach to supervised text classification and presents the main neural network models that can be applied to this specific task. Chapter 3 will deal with the word embedding problem and the detailed explanation of the Word2Vec and the ELMo algorithms. Chapter 4 will focus on how the presented models and algorithms have been applied in real-word scenario. The results of the trained models will be compared both in terms of efficiency and classification metrics.

Chapter 1

The Text Classification problem

Text Classification (TC a.k.a. text categorization) is the activity of labeling natural language texts with thematic categories from a predefined set. TC is a widely studied problem in the database, data mining and information retrieval communities and nowadays it is being applied in many contexts.

Text classification is one of the main tasks of Natural Language Processing (NLP) , the subfield of computer science concerned with using computational techniques to learn, understand and produce human language content. NLP is often called computational linguistics, and this fact implies that it not an independent scientific field, but rather an **interdisciplinary** field (Jurafsky and Martin, 2009). Over the past 20 years, the developments in the field are enabled not only by the increase in computing power and the development of highly successful algorithms but also by the availability of large amount of linguistic data and a much richer understanding of the structure of human language and its deployment in social contexts (Hirschberg and Manning, 2015). Section 1.1 will examine in details the characteristics and the challenges of processing natural language data.

Depending on the applications, the TC task may fall under the binary classification task or under the multi-label classification one. In particular, on the latter case, where categories are not mutually exclusive (like in the case of topic classification) it is not easy to choose which categories are the most appropriate for the text, even for a human expert. Nowadays TC is used for various applications, like document organization, text filtering, opinion mining and many others. Section 1.2 will go deeper into the problem of text classification and then present these three applications in detail.

Text pre-processing is the first step in every Natural Language Processing system pipeline and it has potential impact in its final performance. The aim of pre-processing is to transform text data to sequences of relevant tokens. The main steps of text pre-processing are: tokenization, lowercasing, and

stop words removal. Section 1.3 will present these pre-processing techniques and analyze their importance in a TC task.

After pre-processing, in order to use text data as inputs for a classifier, significant features have to be extracted from the tokens for each entry of the dataset. For text classification, the basic units are words or group of words (bi-grams or tri-grams). The most basic set of features for text classification is the bag of words in the document. It can be useful to weight each word with proportion to its informativeness, for example using TF-IDF weighting. This strategy does not take into account word meaning, but only the word statistics in the dataset. Embedding based methods can be used to derive abstract word features, and these features can be combined together to form the feature set of each document. Section 1.4 will go deeper in the analysis of these two feature representation for text classification. Further analysis on word embedding methods will be presented in Chapter 3.

Since this work focuses on deep learning for text classification, after the feature extraction phase the vector space representation for each document will be fed into a neural network model that will learn an intermediate representation of the document and then learn how to classify it correctly. Feed Forward neural networks, Convolutional neural networks and Recurrent neural networks can be used to solve the task, and they can be also combined together. Neural Networks can also serve as feature extractor at a word level, using an embedding layer or fine-tuning some pre-trained word embedding representations. A general introduction to these models will be given in Section 1.5 and then they will be deepened in Chapter 2 and 3.

The reference literature for this chapter is Sebastiani (2002) for the text classification definitions and applications, Kadhim (2018) for the text preprocessing techniques, and Goldberg (2017) for the remaining parts. This book will be also used as reference in following chapters, when talking about basic NLP concepts and basic deep learning architectures.

1.1 Natural Language Processing challenges

What distinguishes language processing applications from other data processing systems is their use of *knowledge of language*. Consider for example the Unix *wc* program, used to count the total number of bytes, words, and lines in a text file. When used to count bytes and lines, *wc* is an ordinary data processing application. However, when it is used to count the words in a file it requires knowledge about what it means to be a word, and thus becomes a language processing system. Of course, *wc* is an extremely simple system with an extremely limited and impoverished knowledge of language. So-

phisticated conversational agents, or machine translation systems, or robust question-answering systems require a much broader and deeper knowledge of language. Jurafsky and Martin (2009) claim that often language processing models and algorithms are designed to solve **ambiguity** in the input text. An example can be the sentence "Sarah gave a bath to her dog wearing a pink t-shirt" in which it is not immediate that it's Sarah that wears the t-shirt and not her dog.

According to Goldberg (2017), natural language exhibits an additional set of properties that make it even more challenging for computational approaches, including machine learning: it is *discrete, compositional, and sparse*. Language is discrete in the sense that its basic elements, characters and words are discrete symbols, whose meaning is external to them and left to be interpreted. There is no inherent relation between words such as "hamburger" or "pizza" that can be inferred from the symbols themselves, or from the individual letters they are made of. Instead in many signal analysis applications this can be possible: using a simple mathematical operation it's possible for example to move from a colorful image to a gray scale one. Language is compositional: letters form words, words form sentences, sentences form documents. The meaning of a sentence can be larger than the meaning of the words that comprise it, so in order to interpret a document we need to work beyond letter and word levels. These two properties lead to *data sparseness*. The way in which words (discrete symbols) can be combined to form meanings is practically infinite. There is no clear way of generalizing from one sentence to another, or defining the similarity between sentences that does not depend on their meaning, which is unobserved to us. This is very challenging when we come to learn from examples: even with a huge example set we are very likely to observe events that never occurred in the example set, and that are very different than all the examples that did occur in it.

1.2 Text Classification: definition and application

After dealing with NLP challenges, we will focus on the text classification problem. The reference literature for this Section is Sebastiani (2002), whenever not specified otherwise.

A definition of text classification

Text classification is the task of assigning a Boolean value to each pair $(d_j, c_i) \in D \times C$, where D is a domain of documents and $C = \{c_1, ..., c_{|C|}\}$ is a set of predefined *categories*. A value of T assigned to (d_j, c_i) indicates a decision to file d_j under c_i, while a value of F indicates a decision not to file d_j under c_i. More formally, the task is to approximate the unknown *target function* $\phi : D \times C \rightarrow \{T, F\}$ (that describes how documents ought to be classified) by means of a function $\hat{\phi} : D \times C \rightarrow \{T, F\}$ called the classifier (aka rule, or hypothesis, or model) such that $\hat{\phi}$ and ϕ "coincide as much as possible".

Two reasonable assumptions can be made when performing text classifications. Categories are just symbolic labels, and no additional knowledge about their meaning is available. No *exogenous* knowledge, that is data provided for classification purposes by an external source, is available; hence the task must be accomplished only on the basis of *endogenous* knowledge, information extracted from the documents. *Metadata* such as the publication date, the document type, the publication source, etc., is not assumed to be available.

Relying only on endogenous knowledge means classifying a document based solely on its semantics, and given that the semantics of a document is a subjective notion, it follows that the membership of a document in a category cannot be decided in a deterministic way. This is exemplified by the phenomenon of inter-indexer inconsistency (Cleverdon, 1984): when two human experts decide whether to classify document d_j under category c_i, they may disagree, and this in fact happens with relatively high frequency. A news article on *Clinton* attending *Dizzy Gillespie*'s funeral could be filed under *Politics*, or under *Jazz*, or under both, or even under neither, depending on the subjective judgment of the expert.

Single-label versus Multi-label text classification

Some constraints may be enforced on the TC task, depending on the application. For instance we might need that, for a given integer k, exactly k (or $\leq k$, or $\geq k$) elements of C must be assigned to each $d_j \in D$. The case in which exactly one category must be assigned to each $d_j \in D$ is often called the single-label (a.k.a. non-overlapping categories) case, while the case in which any number of categories from 0 to $|C|$ may be assigned to the same $d_j \in D$ is dubbed the multi-label (aka overlapping categories) case.

From a theoretical point of view, the binary case is more general than the multi-label, since an algorithm for binary classification can also be used

for multi-label classification: one needs only transform the problem of multi-label classification under $\{c_1, ..., c_{|C|}\}$ into $|C|$ independent problems of binary classification under $c_i, \bar{c}_i$, for $i = 1, ..., |C|$. However, this requires that categories are stochastically independent of each other, that is, for any $c\prime, c\prime\prime$, the value of $\hat{\phi}(d_j, c\prime)$ does not depend on the value of $\hat{\phi}(d_j, c\prime\prime)$ and vice versa; this is usually assumed to be the case.

In the multi-label classification problem, given $d_j \in D$ a system might simply *rank* the categories in $C = \{c_1, ..., c_{|C|}\}$ according to their estimated appropriateness to d_j, without taking any "hard" decision on any of them. Such a ranked list would be of great help to a human expert in charge of taking the final categorization decision, since it could thus restrict the choice to the category (or categories) at the top of the list, rather than having to examine the entire set.

Most binary classification problems are special case of multi-label problem. For example we may want to build a model which detects which documents talk aboul *Sports*: in the collection there may be documents which talk about *Economy*, *Politics* or other topics and some categories may overlap with the *Sports*. Binary classification problems are important and challenging, since they feature unevenly populated categories (e.g., few documents are about *Jazz*) and unevenly characterized categories (e.g., what is about *Jazz* can be characterized much better than what it is not). In this work, whenever it is not specified, models and algorithms for text classification can be used both for binary and also for multi-label classification problems.

Applications of text classification

Document organization

Many issues pertaining document organization and filing, be it for purposes of personal organization or structuring of a corporate document base may be addressed by TC techniques. Indexing with a controlled vocabulary is an instance of document organization. It consists in assigning to each document one or more key words or key phrases describing its content, where these key words or phrases belong to a finite set called *controlled dictionary*. Usually this assignment is done by trained human indexers, thus it is a costly activity.

If the entries in the controlled vocabulary are viewed as categories, text indexing is an instance of TC, in particular an instance of multi-label TC since every document can have multiple overlapping key words or phrases. Other possible applications of TC for document organization are the organization of patents into categories for making their search easier, the automatic filing of newspaper articles under the appropriate sections (e.g., *Politics*, *Home*

News, *Lifestyles*, etc.) or grouping conference papers into sessions.

Text filtering

Text filtering is the activity of classifying a stream of incoming documents dispatched in an asynchronous way by an information producer to an information consumer. A typical case is a newsfeed, where the producer is a news agency and the consumer is a newspaper. In this case the filtering system should block the delivery of the documents the consumer is likely not interested in (e.g., the news not concerning sports, in the case of a sport newspaper).

Filtering can be seen as a case of binary classification, that is the classification of incoming documents into two disjoint categories, the relevant and the irrelevant ones. Additionally, a filtering system may also further classify documents deemed relevant to the consumer into thematic categories; in the example above, all articles about sports should be further classified according to which sport they deal with. A filtering system can be installed at the producer end, in which case it must route the documents to the interested consumers only, or at the consumer end, in which case it must block the delivery of documents deemed uninteresting to the consumer.

Opinion mining

As stated by Aggarwal and Zhai (2012), opinion mining, or sentiment analysis, is the computational study of people's opinions, appraisals, attitudes and emotions toward entities, individuals, issues, events, topics and their attributes. Businesses always want to find public or consumer opinions about their products and services. Potential customers also want to know the opinions of existing users before they use a service or purchase a product. With the growth of social media on the Web, individuals and organizations are increasingly using public opinions for their decision making. However, finding and processing useful information is not easy for the human reader due to the huge volume of texts present in the sites that is not easy to be deciphered. Moreover, human analysis of text information is subject to biases, e.g., people often pay greater attention to opinions that are consistent to their preferences.

Automated opinion mining can overcome subjective biases and mental limitations with an objective sentiment analysis system. Opinion mining systems can be seen as a case of multi-class classification, in which categories are non overlapping, since opinions (Useful, Not useful, Very Good, Very Bad) are mutually exclusive.

1.3 Text pre-processing techniques

As claimed by Kadhim (2018), text pre-processing is a vital stage in text classification particularly and in text mining generally. The major objective of text pre-processing is to obtain the key features or the key terms from text datasets and to improve the relevancy between words and documents and between words and categories. It has been proven that the time spent on pre-processing can take from 50% up to 80% of the entire classification process (Srividhya and Anitha, 2010), which clearly proves the importance of pre-processing in text classification processes. An effective preprocessor represents the document efficiently in term of both space (for storing the document) and time.

Tokenization

The first step of text pre-processing is tokenization, that is the process of separating strings into basic processing textual units and grouping isolated tokens to create higher level tokens. This operation is the most delicate one and it can be made up of several phases. First, raw texts are converted to a list of symbols including words, numbers and punctuation signs. Secondly, depending on the task, numbers can be removed, kept, converted or grouped together with other numbers or symbols if they have a particular meaning (dates for example). Finally punctuation signs and empty sequences are removed. After these operations each document is segmented into a list of tokens which can be further processed if necessary.

Lowercasing

This is the simplest pre-processing technique which consists of lowercasing each single token of the input text. It has the desiderable property of reducing sparsity and vocabulary size, but it may negatively impact system's performance by increasing ambiguity. For instance, the *Apple* company and the *apple* fruit would be considered as identical tokens.

Stop words removal

A stop word list is a list of commonly repeated words which emerge in every text document. After tokenization, and if necessary lowercasing, every word belonging to the stop word list is removed. Common words such as conjunctions and pronouns need to be removed because their presence have very little or no value on the classification process. In order to find the stop

words, one might use a precomputed list of most common terms of the reference language or compute most frequent words in the dataset and add the first k of such words ($k = 50, 80, ..$) to the list.

Multiword grouping

This last pre-processing technique consists of grouping consecutive tokens together into a single token if it can lead to more meaningful tokens. One example is the expression *United States* which is clearly considered as a unit when reading a piece of text that contains the two words close to one another. The meaning of these multiword expressions are often hardly traceable from their individual words, so treating multiwords as single units may lead to better training of a given model. But this is one of the most time expensive procedure when programming a text preprocessor, because one needs to read (most of) the documents to spot these word groups. Neural network models, as we will see in Chapter 2, can spot groups of words that have particular meanings when they are close to one another.

1.4 Feature design for text classification

The process of feature extraction (also called feature engineering) is one of the most crucial and, at the same time, delicate in any machine learning task. In TC a feature vector needs to be derived from each text document in order to reflect various linguist properties of the text. While deep neural networks alleviate a lot of the need in feature engineering, a good set of core features still needs to be defined. This is especially true for language data, which come in the form of sequences of discrete symbols. These sequences need to be converted somehow to numerical vectors, in a non-obvious way.

Bag of words

When we consider a sentence, a paragraph or a document, the observable features are the counts and the order of the words within the text. A very common feature extraction procedure for sentences and documents is the bag-of-words approach (BOW). In this approach the histogram of the words within the text is computed, i.e., each word count is considered as a feature. As a result, word order is not taken into account when computing document level feature representations. When using the bag-of-words approach, it is common to use TF-IDF weighting (Manning et al., 2008). Consider a document d which is part of a larger corpus D. Rather than representing

each word w in d by its normalized count in the document $\frac{\#_d(w)}{\sum_{w' \in d} \#_d(w')}$ (the **T**erm **F**requency), TF-IDF weighting represents it instead by the quantity $\frac{\#_d(w)}{\sum_{w' \in d} \#_d(w')} \times log \frac{|D|}{|d \in D : w \in d|}$. The second term of the quantity is the **I**nverse **D**ocument **F**requency: the inverse of the number of distinct documents in the corpus in which this word occurs. This highlights words that are distinctive of the current text.

Distributional features

In the bag of words approach words are discrete and unrelated symbols: the words *pizza*, *burger* and *chair* are all equally similar (and equally dis-similar) as far as the algorithm is concerned. Instead, the distributional hypothesis of language states that the meaning of a word can be inferred from the contexts in which it is used. By observing co-occurrence patterns of words across a large body of text, one can discover that the contexts in which *burger* occurs are similar to those in which *pizza* occurs and very different from those in which *chair* occurs. Embedding-based algorithms make use of this property and learn generalizations of words based on the context in which they occur. They represent each word as a vector such that similar words (words having a similar distribution) have similar vectors. These algorithms uncover many facets of similarity between words and can be used to derive good word features: word vectors can be used as representation of the word. Since in this work word embedding vectors are used as features for text classification, they will be discussed in depth in Chapter 3.

From textual features to inputs

When representing a document as feature vector $\boldsymbol{x}$ in a way which is amenable for use by a statistical text classifier, two options are available, the bag-of words representation and the dense semantic embedding vectors. There are some trade-offs and also some relations between the two approaches.

In the bag-of-words approach, documents are usually represented as sparse vectors where the non-zero elements are the ones corresponding to the columns indicating the words in the document. Then, some weighting (i.e. TF-IDF) or some dimensionality reduction technique can be used to transform these document representations.

In the word embedding approach, each word is *embedded* into a d dimensional space, and represented as a vector in that space. The dimension d is much smaller than the number of words in the vocabulary. The word embedding representations can be pre-trained or they can be treated as parameters

of the network, and trained like the other parameters of the classification network. After getting the word vectors, they must be combined in some way (by summation, concatenation or some other operations) to get the document vector $\boldsymbol{x}$.

The main benefit of the semantic representations is their generalization power: if some words provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities. For example, the word *dog* may be observed many times during training, and the word *cat* few times. If each word has its own dimension, occurrences of *dog* will not tell anything about the occurrences of *cat*. However, in the semantic vectors representation the learned vector for *dog* may be similar to the learned vector for *cat*, allowing the model to share statistical strength between the two events.

This work will focus on the use of word embedding vectors as inputs for text classification. Neural network models will be used to obtain word vectors. In Chapter 3 the difference between pre-trained word embedding vectors and automatically learned embedding vectors will be discussed and then two of the most popular word embedding algorithms (Word2Vec and ELMo) will be presented.

1.5 Deep Learning models

As anticipated in the previous sections, this work will focus on Deep Learning models for text classification. Deep Learning is a re-branded name for neural networks, a family of learning techniques which can be characterized as learning of parametrized differentiable mathematical functions. The name deep learning stems from the fact that many layers of these differentiable functions are often chained together.

While all of machine learning can be characterized as learning to make predictions based on past observations, deep learning approaches work by learning not only to predict but also to *correctly represent* the data, such that they are suitable for predictions. We have seen in Section 1.4 what embedding vectors are and what they are used for in the text classification procedure. A major component in neural networks for language processing is the use of the *embedding layer*, a mapping of discrete symbols to continuous vectors in a relatively low dimensional space. The representation of words as vectors can be the objective of the neural network training (in case of Language Modeling) or it can be learned by the network during the training process of NLP applications (for example in the case of Text Classification). In the latter case, going up the hierarchy, the network also learns how to

combine word vectors in a way that is useful for prediction.

There are three major kinds of neural network architectures that can be combined in various ways: feed-forward networks, convolutional networks and recurrent networks. Feed-forward networks, in particular Multi-Layer Perceptrons (MLPs), allow to work with fixed sized inputs, or with variable length inputs in which we can disregard the order of the elements. Convolutional Neural Networks (CNNs) are specialized architectures that excel at extracting local patterns in the data: they are capable of extracting meaningful local patterns that are sensitive to word order, regardless of where they appear in the input. These models work very well for identifying indicative phrases or idioms up to a fixed length in long sentences or documents. Recurrent Neural Networks (RNNs) are specialized models for sequential data. In particular, gated architectures, like Long Short Term Memory Networks, are a family of effective neural network models when working with sequences. These are network components that take as input a sequence of items, and produce a fixed size vector that summarizes that sequence. Convolutional and Recurrent networks are rarely used as standalone components, and their power is in being trainable components that can be fed into other network components, and trained to work in tandem with them. For example, the output of a recurrent network can be fed into a feed-forward network that will try to predict some value. In this case, the recurrent network is used as an input-transformer that is trained to produce informative representations for the feed-forward network that will operate on top of it.

In Chapter 2 the three main neural network architectures (MLPs, CNNs and RNNs) will be explained as well as their possible application for text classification. In Chapter 3 the problem of word embedding vectors generation will be discussed and in particular it will introduce the concept of *language modeling*, a task that can be used to compute pre-trained word representations.

Chapter 2

Deep Learning models for NLP

Neural Networks, the main focus of this work, belong to the class of supervised machine learning algorithms. The main definitions and mathematical concepts of supervised classification will be presented in Section 2.1. In order to construct a proper designed classifier, a dataset[1] should first be split into 3 different sets which are called training set, validation set and test set. Then, an optimization algorithm using the training set observations will find the proper values of the classifier parameters that minimize a loss function. Finally, using proper classification metrics, the classifier should be evaluated using the validation and the test sets.

After presenting these concepts, Section 2.2 will present the Feed Forward Neural Network models, which are the most simple yet very powerful neural network architectures. The introduction of nonlinear activation functions and multiple layers of learned representation is crucial for the good performance of these models for most tasks.

Convolutional Neural Networks (CNNs) are popular models for computer vision applications and in recent years they have been applied to NLP tasks with good success. CNN networks, like RNN ones (as we will see later) are not used as standalone components but as components of larger networks that serve as feature extractors. Using the convolution and the pooling operations, convolutional layers look at groups of words (ngrams) in a text and learn which word groups can be informative for the task, regardless of their position in the text. Two different types of Convolutional Neural Network architectures will be discussed in Section 2.3.

Recurrent Neural Networks (RNNs) are popular neural network models when working with sequences. They extract features from arbitrarily sized sequential inputs. RNNs are defined recursively and they can extract features

[1] In this work we assume that a proper dataset in given to the algorithm designer; we do not deal with data collection methods.

by looking the sequence forward, backward or in both directions. Long-Short Term Memory (LSTM) networks use the gate operation which make them very effective for solving machine learning problems. RNNs are presented in depth in Section 2.4.

The reference book for figures, terminology and mathematical notation in this Chapter is Goldberg (2017), whenever not specified otherwise.

2.1 Supervised classification

The essence of supervised machine learning is the creation of mechanisms that can look at examples and produce generalization. More concretely, it means that the ultimate scope is to design an algorithm whose input is a set of labeled examples (e.g. a set of documents) and its output is a function that can automatically label a given instance (e.g. assign it the proper classes). When we choose the machine learning algorithm (Random Forest, Support Vector Machines or Neural Networks) we restrict ourselves to search over specific families of functions, rather than all possible functions. Such families of functions are called *hypothesis classes* while the form of the solution is called *inductive bias*.

The most common and simple hypothesis class is that of high-dimensional linear functions, i.e., functions of the form:

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b} \tag{2.1}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}}.$$

The vector $\boldsymbol{x}$ is the *input* to the function, while the matrix $\boldsymbol{W}$ and the vector $\boldsymbol{b}$ are the *parameters*. The goal of the learner to perform an optimization search over the space of functions is reduced to a search over the space of parameters. It is common to refer to parameters of the function as Θ and to include them in the function definition: $f(\boldsymbol{x}\,;\Theta) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$.

In this Chapter the deep learning models presented will be applied to solve a classification task. As we said in Chapter 1, a text classification problem can be seen as a binary classification or as a multi-class classification problem. In the latter case texts can have a single label or multiple labels (multi-label classification). We are going to restrict to the binary case, since it is the most simple one. If necessary, we are going to explain the different methodology when dealing with multi-class or multi-label classification.

Train, validation and test split

Before introducing the math behind supervised classification, we want to focus on how to use the dataset available in order to produce a function $f(\boldsymbol{x})$ which is able to map inputs to outputs correctly and also to generalize well to unseen data. A possible solution is to split the training set into two subsets, say in a 80%/20% split, train a model on the larger subset (the *training set*) and test its performances on the smaller set (the *test set*). Some care must be taken when performing the split, to ensure that the two sets maintain the original dataset distribution (especially in terms of the labels). This split works well if we want to train a single model and assess its quality. However, if we want to train several models, compare their quality and select the best one, the two-way split approach is insufficient. When using a single test set we cannot be sure if the chosen setting of the final classifier are good in general, or just good for the test set examples. For this reason the standard methodology in machine learning is to use a three-way split of the data into *training*, *validation* and *test set*. All the experiments, tweaks, error analysis and model selection should be based on the validation set. Then, a single run of the final model over the test set will give a good estimate on its expected quality on unseen examples.

The Classification problem

In binary classification problems we have a single output for each instance in the dataset. When using a linear classifier, we can use a restricted version of Equation 2.1 in which the weight matrix is the one-dimensional vector $\boldsymbol{w}$, $d_{out} = 1$ and the bias vector is a scalar b:

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w} + b \tag{2.2}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{w} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{b} \in \mathbb{R}.$$

The output $f(\boldsymbol{x})$ is in the range $[-\infty, +\infty]$. In order to use it for the binary classification we need to map it to the range $[0, 1]$. The most used map is the sigmoid function, which is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

Figure 2.2 shows a plot of the sigmoid function. It is monotonically increasing and maps values to the range $[0, 1]$, with 0 being mapped to $\frac{1}{2}$. When using a suitable *loss function*, the value of $\sigma(x)$ can be interpreted as an estimate of the probability that the document with the feature vector $\boldsymbol{x}$ belongs to the considered class.

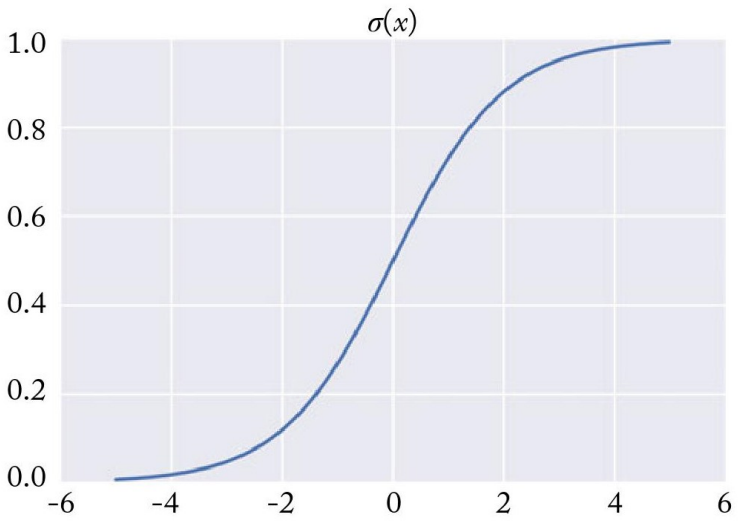

Figure 2.1: The sigmoid function $\sigma(x)$

When we have multiple classes, in the case of multi-label classification we can train a model that solves multiple binary classification problems at once. This means that we have an output $f(\boldsymbol{x}) \in \mathbb{R}^{d_{out}}$ which is a vector of the same dimension as the number of labels. We can apply to the output $f(\boldsymbol{x})$ the element-wise sigmoid function in order to estimate the probabilities that a document belongs to each of the classes. Instead, when we have a multi-class classification problem we want the sum of the estimated probabilities to be exactly 1, since instances can only have one correct label. In this case we compute the element-wise softmax function which is a slightly modified version of the sigmoid function that fits this criteria:

$$softmax(\boldsymbol{x}_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}. \tag{2.4}$$

The details on how the training procedure is carried out in machine learning applications will be explained in the Appendix Section A.1.

Classification metrics

After training the model and predicting the conditional probabilities of the validation set instances, to measure the model performances we need to convert prediction values $\hat{p}_i$ from the interval $(0, 1)$ to the set $\{0, 1\}$, i.e. assign the document to the class i or not, for each i. In case of multi-class classification problems the conversion method is pretty straightforward; we assign to

the document the class that has the greatest value of predicted probability:

$$\hat{\boldsymbol{y}} = \underset{i}{\operatorname{argmax}}\, \hat{p}_i. \tag{2.5}$$

For the binary classification and the multi-label classification problems the most used choice is using a threshold strategy, with a threshold value typically set to 0.5 . The prediction is set to 1 for each document in which the predicted probability for the i-th class is above the threshold, otherwise it is set to 0:

$$\hat{y}_i = \begin{cases} 0, & \hat{p}_i < 0.5 \\ 1, & \hat{p}_i \geq 0.5. \end{cases} \tag{2.6}$$

However this solution may not be very effective, in particular when we have a very unbalanced distribution between positive and negative true labels for a specific category. This is particularly valid for multi-label classification problems with lots of classes. A different approach will be discussed in Chapter 4.

For binary classification problems, the discrimination evaluation of the best (optimal) solution during the model training can be defined based on the confusion matrix shown in Table 2.1. The rows of the table represent the predicted class, while the columns represent the actual class. From this confusion matrix, tp and tn denote the number of positive and negative instances that are correctly classified. Meanwhile, fp and fn denote the number of misclassified negative instances and the number of misclassified positive instances respectively. From Table 2.1 several commonly used metrics can be generated.

Table 2.1: Confusion Matrix for Binary Classification

	Actual Positive Class	**Actual Negative Class**
Predicted Positive Class	True Positive (tp)	False Positive (fp)
Predicted Negative Class	False Negative (fn)	True Negative (tn)

The most popular classification metrics are *Accuracy*, *Precision*, *Recall* and $F1 - Score$ ($F1$). The definitions of these metrics are reported below:

$$Accuracy = \frac{tp + tn}{tp + fp + fn + tn} \tag{2.7}$$

$$Precision = \frac{tp}{tp + fp} \tag{2.8}$$

$$Recall = \frac{tp}{tp + fn} \tag{2.9}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{2.10}$$

For multi-class and multi-label problems, the metrics presented above can be extended by taking the average of the metric values over the classes.

Despite the fact that *accuracy* is the most used metric in classification problems, it is not the most appropriate one. The reason is that when we have a very unbalanced dataset and we are not able to charaterize very well all the classes, the value of *accuracy* does not tell this information. Instead this job is done by *precision* and *recall* focus on the positive observations and thus they are able to tell us if the model is able to character well the positive class.

2.2 Feed Forward architecture

Neural networks were inspired by the brain's computational mechanism, which consists of computational units called neurons. In the metaphor between the brain and artificial neural networks, a neuron is a computational unit that has scalar inputs, some output and an associate weight. Figure 2.2 shows an example of a neuron.

Neurons are connected to each other, forming a network: the output of a neuron may feed into the inputs of one or more neurons. If the weights are set correctly, a neural network with enough neurons and a nonlinear activation function can approximate a very wide range of mathematical functions.

A typical feed-forward neural network is shown in Figure 2.3. Each circle is a neuron, with incoming arrows being the neuron's inputs and outgoing arrows being the neuron's outputs. Neurons are arranged in layers reflecting the flow of information. The bottom layer has no incoming arrow and it is called the input of the network. The top-most layer has no outgoing arrows and it is the output of the network. The other layers are considered "hidden". Inside the middle layers circle the sigmoid shape represents the nonlinear activation function. In a network with different types of layers, the one in which each neuron is connected to all of the neurons in the next layers is called a *fully connected* layer.

From a mathematical point of view, a feed-forward network is simply a stack of linear models separated by nonlinear functions. The simplest neural network is called *Perceptron*. It is simply a linear model:

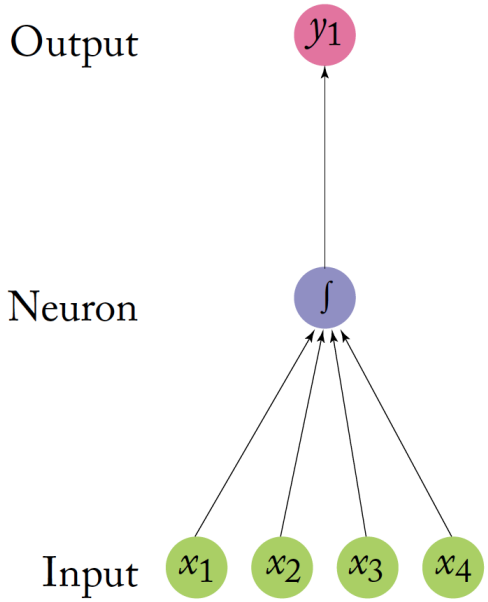

Figure 2.2: A single neuron with four inputs

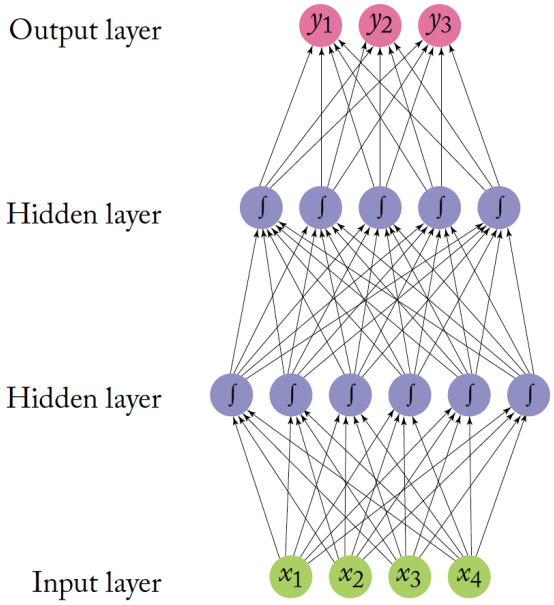

Figure 2.3: A Feed-forward neural network with two hidden layers

$$NN_{Perceptron}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b} \tag{2.11}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}} \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}}.$$

In order to go beyond linear functions, we introduce a nonlinear hidden layer resulting in the Multi Layer Perceptron with one hidden layer($MLP1$):

$$NN_{MLP1}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1)\,\boldsymbol{W}^2 + \boldsymbol{b}^2 \tag{2.12}$$
$$\boldsymbol{x} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{W}^1 \in \mathbb{R}^{d_{in} \times d_1} \quad \boldsymbol{b}^1 \in \mathbb{R}^{d_1} \quad \boldsymbol{W}^2 \in \mathbb{R}^{d_1 \times d_2} \quad \boldsymbol{b}^2 \in \mathbb{R}^{d_2}$$

Here $\boldsymbol{W}^1$ and $\boldsymbol{b}^1$ are a matrix and a bias term for the first linear transformation of the input, g is an element-wise applied nonlinear function and $\boldsymbol{W}^2$ and $\boldsymbol{b}^2$ are the matrix and the bias term for a second linear transformation.

The vector resulting from each linear transformation is referred to as a *layer*. The outer-most linear transformation results in the *output layer* and the other linear transformations result in the *hidden layers*. The matrices and the bias terms that define the linear transformations are the *parameters* Θ of the network. The nonlinearity of the network is given by the functions g and this is crucial to represent complex functions.

We refer to the Appendix Section A.2 for some details on the parameter optimization procedure in deep neural networks and on some techniques that can be used to prevent overfitting.

2.3 Convolutional Neural Networks

Sometimes we are interested in making predictions based on ordered sets of items. This section introduces the Convolutional Neural Network (CNN) architecture, which is tailored for this problem. A convolutional neural network is designed to identify indicative local predictors in a large structure, and to combine them to produce a fixed size vector representation of the structure, capturing the local aspects that are most informative for the prediction task at hand. The convolutional architecture also allows to share predictive behaviors between ngrams that share similar components. The CNN is essentially a feature-extracting architecture. It is meant to be integrated into a larger network and to work in tandem with other network components in order to produce an end result.

Basic convolutional architecture

The main idea behind the convolutional architecture for language tasks is to apply a nonlinear function over each instantiation of a k-word slicing window over the sentence. This function (also called *filter*) transforms a window of k word vectors into a scalar value. Several such filters can be applied,

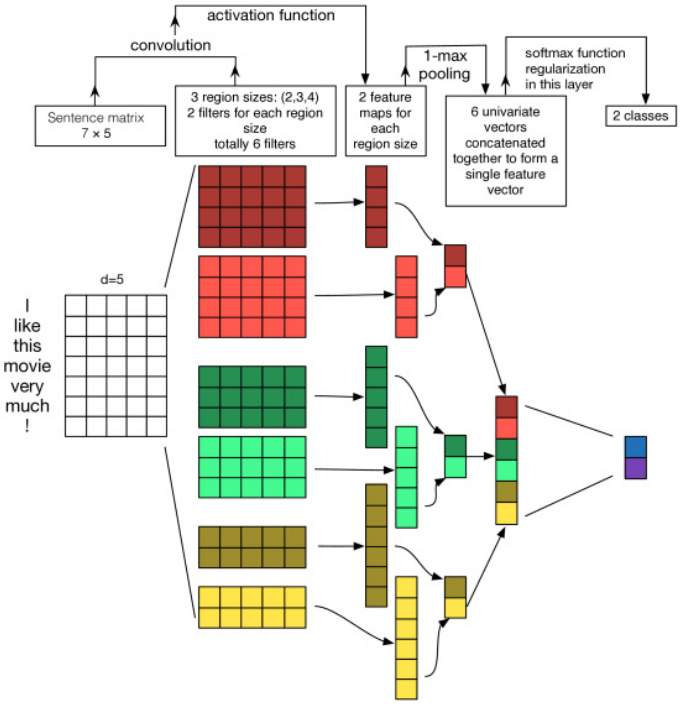

Figure 2.4: Illustration of a CNN architecture with three filter region sizes: 2, 3 and 4, each of which has 2 filters. The final vector is fed to a fully-connected output layer of size 2.

resulting in a l-dimensional vector that captures important properties of the words in the window. Then, a *pooling* operation is used to combine the vectors resulting from the windows into a single l-dimensional vector. The intention is to focus on the most important features in the sentence, regarding on their location, while the pooling operation zooms in on the important indicators. The resulting vector is fed into a network that is used for predictions. During the training process the parameters of the filter function are optimized using gradient-based methods to highlight the aspects of the data that are important for the task the network is trained for. When the sliding window of size k is run over a sequence, the filter function learns to identify informative k-grams. More details on the convolution and pooling layers are given in the Appendix Section A.3.

Multiple filter size architecture

For most computer vision tasks, multiple layers of Convolution and Pooling can be used, in order to capture different kinds of features in the different

layers. In NLP applications, the features we are interest in are the meanings of the k-grams in the sentences, independently of their position. This information can be captured using a single convolutional layer, by setting the proper value of k. One can set k to be very little and use multiple layers. In the case of $k = 2$ the first layer learns features about bigrams, the second about trigrams and so on and so forth. This kind of architecture requires good knowledge by the neural network designer, and it may capture useless information since trigrams or four grams could be useless for the task at hand.

Zhang and Wallace (2015) proposed a CNN architecture which is more efficient than the standard one and it allows to specify the multiple filter sizes. In the same layer m filters of size k are computed. If r different filter sizes are specified, m filters are computed for each value of k. Then, the max pooling operation is applied r times to get r vectors of size l. Together, the outputs generated from each filter map can be concatenated into a fixed length, top-level feature vector that can be fed to a MLP for further layers of learned representations. Figure 2.4 shows an example of such architecture.

2.4 Recurrent Neural Networks

When dealing with language data, it is very common to work with sequences, such as words (sequences of letters) and sentences (sequences of words). CNN representations are useful as they offer some sensitivity to word order, but their order sensitivity is restricted to mostly local patterns, and disregard the order of patterns that are far apart in the sequences.

Recurrent Neural Networks allow representing arbitrarily sized sequential inputs in fixed-size vectors, while paying attention to the structured properties of the inputs. RNNs, particularly the ones with gated architectures such as the LSTMs, are very powerful at capturing statistical regularities in sequential inputs.

RNN abstraction

Call $\boldsymbol{x}_{i:j}$ the sequence of vectors $\boldsymbol{x}_i, ..., \boldsymbol{x}_j$. The RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in}-dimensional vectors $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$ and returns as output a single d_{out}-dimensional vector $\boldsymbol{y}_n$:

$$\boldsymbol{y}_n = RNN(\boldsymbol{x}_{1:n}) \tag{2.13}$$

This implicitly defines an output vector $\boldsymbol{y}_i$ for each prefix $\boldsymbol{x}_{1:i}$ of the sequence $\boldsymbol{x}_{1:n}$. The output vector is fed to other neural network layers for

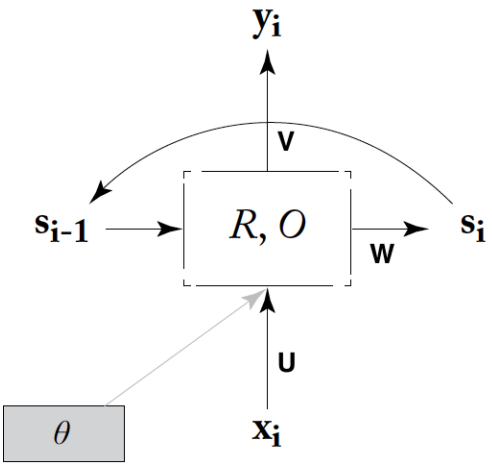

Figure 2.5: Graphical representation of a RNN(recursive)

further predictions. A RNN is defined recursively, by means of a function R taking as input a state vector $\boldsymbol{s}_{i-1}$ and an input vector $\boldsymbol{x}_i$ and returning a new state vector $\boldsymbol{s}_i$. The state vector $\boldsymbol{s}_i$ is then mapped to an output vector $\boldsymbol{y}_i$ using a deterministic function $O(\cdot)$. The base of the recursion is an initial state vector $\boldsymbol{s}_0$ which we can assume to be the the zero vector. The RNN definition becomes:

$$\begin{aligned} \boldsymbol{y}_n &= RNN(\boldsymbol{x}_{1:n}; \boldsymbol{s}_0) \\ \boldsymbol{y}_i &= O(\boldsymbol{s}_i) \\ \boldsymbol{s}_i &= R(\boldsymbol{s}_{i-1}, \boldsymbol{x}_i) \end{aligned} \tag{2.14}$$

The functions R and O are the same across the sequence positions, but the RNN keeps track of the states of computation through the state vector $\boldsymbol{s}_i$. Graphically the RNN has been traditionally presented as in Figure 2.5. The representation follows the recursive definition and it is correct for arbitrarily long sequences. However, for a finite sized input sequence (all input sequences we deal with are finite) one can *unroll* the recursion, resulting in the structure shown in Figure 2.6.

It is easy to see that an unrolled RNN is just a very deep neural network in which the same parameters are shared across many parts of the computation, with an additional input added at various layers. In literature the backpropagation applied to a recursive network like RNN is referred to as *backpropagation through time* (BPTT) (Werbos, 1990). The RNN is a fea-

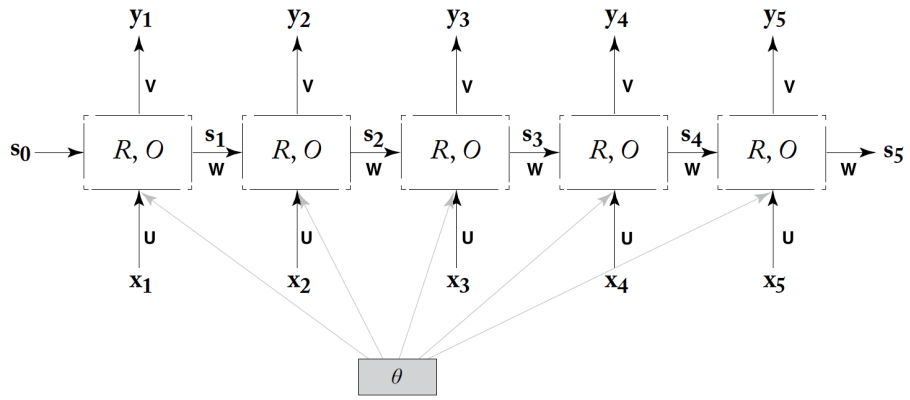

Figure 2.6: Graphical representation of a RNN(unrolled)

ture detector much like the CNN. It encodes properties of the input sequence that are useful for predictions.

RNN models

Bidirectional RNN

An useful elaboration of an RNN is a *bidirectional-RNN*, commonly referred as biRNN (Schuster and Paliwal, 1997). When using a word embedding representation for a document set, a RNN allows to compute a function of the i-th word based on the words $x_{1:i}$ up to including it. However the following words $\boldsymbol{x}_{i+1:n}$ may be also useful to prediction.

Consider an input sequence $\boldsymbol{x}_{i:n}$. The biRNN works by maintaining two separate states $\boldsymbol{s}_i^f$ and $\boldsymbol{s}_i^b$ for each input position i. The *forward state* $\boldsymbol{s}_i^f$ is based on $\boldsymbol{x}_{1:i}$, while the *backward state* $\boldsymbol{s}_i^b$ is based on $\boldsymbol{x}_{n:i}$. The forward and backward states are generated by two different RNNs. The first RNN (R^f, O^f) reads the input sequence $\boldsymbol{x}_{1:n}$ as it is, while the second RNN (R^b, O^b) reads the input sequence in reverse. The state representation $\boldsymbol{s}_i$ is then composed of both states. The output of the function $biRNN$ corresponding to the i-th word is the concatenation of two RNNs, one reading the sequence from the beginning and the other one reading it from the end:

$$biRNN(\boldsymbol{x}_{i:n}, i) = \boldsymbol{y}_i = [RNN^f(\boldsymbol{x}_{1:i}); RNN^b(\boldsymbol{x}_{n:i})]. \tag{2.15}$$

The bidirectional RNN is useful as a general-purpose trainable feature-extracting component that can be used whenever a window around a given word is required.

Multi-layer RNN

RNNs can be stacked in layers, forming a grid (El Hihi and Bengio, 1995). While it is not theoretically clear what is the additional power gained by the deeper architecture, it was observed empirically that deep RNNs work better than shallower ones on some tasks.

Concrete RNN architectures

In Equation 2.14 the functions R and O are left generic. Every RNN architecture has its specific definition of R and O.

Simple RNN

The simplest RNN model that is sensitive to the ordering of the elements in the sequence is known as an Elman Network or Simple-RNN (Elman, 1990). It takes the form:

$$\begin{aligned} \boldsymbol{s}_i &= R_{SRNN}(\boldsymbol{x}_i, \boldsymbol{s}_{i-1}) = g(\boldsymbol{s}_{i-1}\,\boldsymbol{W}^s + \boldsymbol{x}_i\,\boldsymbol{W}^x + \boldsymbol{b}) \\ \boldsymbol{y}_i &= O_{SRNN}(\boldsymbol{s}_i) = \boldsymbol{s}_i. \end{aligned} \tag{2.16}$$

The state $\boldsymbol{s}_{i-1}$ and the input $\boldsymbol{x}_i$ are linearly transformed, the results are added together with a bias term and then passed through a nonlinear activation function g. The output at position i is the same as the hidden state in that position. The notation in Equation 2.16 can be simplified by concatenating the vectors $\boldsymbol{s}_{i-1}$ and $\boldsymbol{x}_i$ using only one matrix $\boldsymbol{W}$.

The SRNN is hard to train effectively because of the vanishing gradient problem (Pascanu et al., 2013). Gradients in later steps diminish quickly in the back-propagation process, and do not reach earlier input signals, making it hard for the S-RNN to capture long range dependencies. Gating-based architectures such as LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Cho et al., 2014) are designed to solve this deficiency.

LSTM

The building block of LSTM and GRU architectures are *gate vectors*. These vectors g are parameters of the RNN which are passed to a sigmoid activation function and thus they can have only elements in the interval $(0, 1)$. After that, an element-wise multiplication between $\sigma(\boldsymbol{g})$ and a memory vector $\boldsymbol{x}$ is performed. This allows the RNN to select which information in the vector $\boldsymbol{x}$ to keep, by setting a value in the vector $\sigma(\boldsymbol{g})$ near 1, and which information to

disregard, by setting a value in the vector $\sigma(\boldsymbol{g})$ near 0. This gating mechanism allows to solve the vanishing problem.

The Long-Short-Term Memory (LSTM) architecture is the first to introduce gating mechanism. Mathematically, the LSTM architecture is defined as:

$$\begin{aligned} \boldsymbol{s}_j = R_{LSTM}(\boldsymbol{s}_{j-1}; \boldsymbol{h}_j) &= [\boldsymbol{c}_j; \boldsymbol{h}_j] \\ \boldsymbol{c}_j &= \boldsymbol{f} \odot \boldsymbol{c}_{j-1} + \boldsymbol{i} \odot \boldsymbol{z} \\ \boldsymbol{h}_j &= \boldsymbol{o} \odot tanh(\boldsymbol{c}_j) \\ \boldsymbol{y}_j = O_{LSTM}(\boldsymbol{s}_j) &= \boldsymbol{h}_j \end{aligned} \tag{2.17}$$

The state at time j is composed of two vectors, $\boldsymbol{c}_j$ and $\boldsymbol{h}_j$, where $\boldsymbol{c}_j$ is the memory component and $\boldsymbol{h}_j$ is the hidden state component. There are three gates, called the **i**nput, **f**orget and **o**utput gates, computed by linear combinations of the current input and the previous state. An update candidate $\boldsymbol{z}$ is also computed as a linear combinations of the current input. The memory $\boldsymbol{c}_j$ is updated combining the forget gate, which controls how much of the previous memory to keep, and the input gate, which controls how much of the proposed update to keep. The value of $\boldsymbol{h}_j$ is determined based on the content of the memory $\boldsymbol{c}_j$, passed through a tanh nonlinearity and controlled by the output gate. The gating mechanisms allow for gradients related to the memory part $\boldsymbol{c}_j$ to stay high across very long time ranges.

LSTMs are currently the most successful type of RNN architecture, and they are responsible for many state-of-the-art sequence modeling results. Their main competitor is the GRU. In the NLP applications we will see, LSTM will be used as RNN models, but in practice they are two valid alternatives.

Chapter 3

Word embedding strategies

A main component of the neural network approach in Natural Language Processing is the use of word embedding vectors. Embedding means representing each word as a vector in a low dimensional space. These vector representations are used to perform various NLP tasks (text classification for example).

The word embedding representations can be included in the neural network parameters of a given task and tuned by the network itself. The alternative is using pre-trained word vectors to map each word in the training set to its corresponding representation. After introducing the word embedding problem, in Section 3.1 we will discuss and compare these two approaches.

Word representations can be learned as a byproduct of the language modeling task. Different models have been developed in recent years to solve this task. We will focus on neural language models, i.e. models using nonlinear neural networks. In Section 3.2 we will go deeply into the language modeling problem and then we will focus on one of the most famous approach, the Word2Vec algorithm.

The most recent trend in language modeling is the development of contextual word embedding algorithms, which are commonly called *Generalized Language Models*. ELMo is one of the first generalized language model to be developed. Convolutional and LSTM layers are combined in the ELMo architecture in such way that the learned embedding vectors exploit both sub-word and sentence-level information. In Section 3.3 we will go deeply on the architecture of ELMo and then analyze some of its characteristics, like the treatment of out-of-vocabulary tokens, its use for supervised NLP tasks and the information that is captured by its layer representations.

The reference book for this Chapter is Goldberg (2017), whenever not specified otherwise.

3.1 The word embedding problem

The goal of the word embedding problem is to produce a mathematical function that is capable of mapping in a meaningful and useful way symbolic categorical features (such as words) to a d-dimensional feature vector for some d. The number of dimensions allocated for each word is not a fixed parameter. When training word vectors usually the data scientist experiments a few different sizes and chooses the one that ensures a good trade-off between speed and task performance. The set of symbols for which we can associate an embedding vector is called *vocabulary*. This vocabulary is usually based on the training set, or, if we use pre-trained embedding vectors, on the training corpus of the pre-trained model. The vocabulary usually includes a special symbol representing out-of-vocabulary (OOV) words in order to assign them a special vector.

As we briefly mentioned in Section 1.4 we want to produce word vectors that have some desirable properties. The most important property is their generalization power: if a sentence has never been seen by the task model during training, its meaning should be inferred by the combination of its word embedding vectors. As we will see later, pre-trained embedding vectors should have properties that make them suitable for the use in several tasks.

Embedding layers

When enough supervised training data are available, one can just treat the embedding feature vectors the same as the other parameters: initialize the vectors to random values and let the network training procedure tune them into "good" vectors. The mapping from a symbolic feature value (the i-th word) to a d-dimensional vector is performed by an *embedding layer* (also called *lookup layer*). The parameters in an embedding layer are simply the entries of the matrix $\boldsymbol{E} \in \mathbb{R}^{|vocab| \times d}$, where each row corresponds to a different word in the vocabulary. The lookup operation is simply indexing: the vector v_i corresponding to the i-th entry in the vocabulary is mapped to the i-th row of the matrix $\boldsymbol{E}$.

Pre-trained embedding vectors

The common case is that we need to perform a supervised task without large enough amounts of annotated data. In such case, we resort to unsupervised auxiliary tasks, which can be trained on huge amounts of text.

The key idea behind the unsupervised approaches is that one would like the embedding vectors of similar words to have similar vectors. The current

approaches derive from the distributional hypothesis (Harris, 1954), stating that words are similar if they appear in similar contexts. The different methods all create supervised training instances in which the goal is to either predict the word from its context or predict the context from the word.

An important benefit of training word embedding vectors on large amounts of unannotated data is that it provides semantic representations for words that do not appear in the supervised training set. Ideally, the representations for these words will be similar to those of related words that do appear in the training set, allowing the model to generalize better on unseen events. Thus it is desired that the similarity between word vectors learned by the unsupervised algorithm captures the same aspect of similarity that are useful for the supervised task.

In the neural network approach, where the community has a tradition of thinking in terms of *distributed representations* (Hinton et al., 1986), each entity is represented as a vector and the meaning of the entity and its relations with other entities are captured by the similarities between vectors. In the context of language processing it means that words and sentences are mapped to a shared low dimensional space. The word vectors are built so that each dimension is not interpretable, and specific dimensions do not necessarily correspond to specific concepts. The meaning of the word will be captured by its relation to other words and the values in their vector.

In the following sections of this chapter we will see two examples of pre-trained word embedding algorithms: the Word2Vec family of algorithms (Mikolov et al., 2013b,a) and the ELMo (Peters et al., 2018) one. As we will see pre-trained word embedding algorithms create word vectors as a byproduct of the language model task training.

Word embedding fine-tuning

When using pre-trained word embedding vectors, we have to choose if we want to exploit the fine-tuning operation for the word vectors. This consists in initializing the embedding matrix $\boldsymbol{E}$ with the pre-trained vectors, and then change it with the rest of the network. While this works well, it has the potential undesirable effecto of changing the representations for words that appear in the training data, but not for other words that used to be close to them in the original pre-trained matrix $\boldsymbol{E}$. An alternative is to leave the pre-trained vectors $\boldsymbol{E}$ fixed. This keeps the generalization, but prevents the model from adapting the representations for the given task.

3.2 Language Modeling

Language modeling (LM) is the use of various statistical and probabilistic techniques to determine the probability of a given sequence of words occurring in a sentence. Besides assigning a probability to each sequence of words, the language model also assigns a probability for the likelihood of a given word (or sequence of words) to belong in a given context.

Even without achieving human-level performance, language modeling is a crucial component in real-word applications such as machine-translation and automatic speech recognition, where the system produces several translation or transcription hypotheses, which are then scored by a language model.

Formally, the task of language modeling is to assign a probability to any sequence of words $w_1, ..., w_n$. Using the chain-rule of probability, it can be written as:

$$P(w_1, ..., w_n) = P(w_1)P(w_2|w_1)P(w_3|w_2, w_1)...P(w_n|w_1, ...w_{n-1}). \quad (3.1)$$

for the unsupervised word embedding computation task.This is a sequence of word-prediction tasks where each word is predicted conditioning on the preceding words. A language model based on neural networks was popularized by Bengio et al. (2003). More details on this model can be found at the Appendix Section B.1. A neural language model can be trained in order to obtain semantic word representations as a byproduct of training. Basing on this property different algorithms have been built for the unsupervised word embedding computation task.

Word2Vec

Several algorithms have been developed in recent years for the unsupervised word embedding computation task inspired by language modeling. In particular the algorithm of Collobert and Weston (Collobert and Weston, 2008) and the Word2Vec family of algorithms (Mikolov et al., 2013b,a) are designed to perform the same side effects of language modeling, using a more efficient and more flexible framework. The GloVe algorithm by Pennington et al. (2014) follows a similar objective.

The Word2Vec family of algorithms differ from a neural language model in the network training objective. The design of the neural language model architecture is driven by the language modeling task, which poses two important requirements: the need to produce a *probability distribution* over words, and the need to condition on contexts that can be combined to produce sentence-level probability estimates. If we only care about the resulting

representations, both constraints can be released, as was done in the model by Collobert and Weston. These changes are also exploited in the Word2Vec family of algorithms. The first change introduced was changing the context of a word from the preceding $k$gram to a word window surrounding it. This implies computing $P(w_3|w_1, w_2, w_4, w_5)$ instead of $P(w_5|w_1, w_2, w_3, w_4)$. Secondly, instead of having as output a probability distribution of words given some context, the model just attempts to assign a score to each word, such that the correct word scores above incorrect ones.

The widely popular Word2Vec algorithm was developed by Tomáš Mikolov and colleagues over a series of papers (Mikolov et al., 2013b,a). It starts with a neural language model and modifies it to produce faster results.

Let w be a target word and $c_1, ..c_k$ an ordered list of context items. The word embedding vectors are computed as outputs of a fully connected neural network with one hidden layer. After that, the model computes a score $s(w; c_1, ..c_k)$ of a word-context pair. The training objective is to maximize the likelihood of the correct word-context pairs. The two context definitions (CBOW and Skip-Gram) lead to different definitions of the word-context scoring function. The network can be optimized using two optimization objectives (Negative-Sampling and Hierarchical Softmax). More details on the two context definitions and on the two optimization objectives can be found in the Appendix Section B.2.

The Word2Vec algorithms are very effective in practice and are highly scalable, allowing to train word representations with very large vocabularies over bilions of words of text in matter of hours, with very modest memory requirements.

3.3 ELMo

Contextual word representations

The quality of word representations is generally gauged by their ability to encode syntactical information and handle polysemic behavior (or word senses). Recent approaches in this area encode such information into its embedding by leveraging the context. These methods, often called Generalized Language Models (GLMs), provide deeper networks that calculate word representations as a function of its context.

Consider the two sentences: "The *bank* will not accept cash on Sunday" and "The river overflowed the *bank*". The word senses of *bank* are different in these two sentences because the two contexts are different. Hence, one might want two different vectors representing the word *bank* in the two sen-

tences. The new class of algorithms diverges from the concept of global word representations and proposes contextual word embedding representations.

Embedding from Language Model (ELMo), developed by Peters et al. (2018) is one such method that provides deep contextual embedding vectors. It provides an unsupervised pre-trained model to compute word representations as a function of the entire input sentence. These representations can be fine-tuned for a given supervised NLP task. Other models with different architectures have been developed with the same objective, such as the OpenAI-GPT (Radford, 2018) and BERT (Devlin et al., 2019) which adapt and utilize the Transformer model (Vaswani et al., 2017).

ELMo is a deep neural network model that computes multiple layer of representations for each word. We will now see how it computes a context-independent representation for each word.

Character level convolution

The first layer of the ELMo model is the output of a character-level convolutional neural network (CharCNN). In Section 2.3 we saw the CNN architecture and we saw that using temporal convolution in NLP tasks we can learn to capture the local aspects that are most informative for the prediction task at hand. The CharCNN model used in ELMo to compute the context-independent representations is the same presented by Kim et al. (2016).

Let C be the vocabulary of characters, d the dimension of the character embedding vectors and suppose that the token t_k is made up of a sequence of characters $c_1, ..., c_l$, where l is the length of the token t_k. Then the character level representation of k is given by the matrix $\boldsymbol{C}_k \in \mathbb{R}^{d \times C}$, where the j-th column corresponds to the character embedding for c_j. Using the same logic as the architecture proposed by Zhang and Wallace (2015), already described in Section 2.3, the convolution and the pooling operation is used to compute multiple feature maps, one for each width of the convolutional kernel. An example of a CharCNN can be seen in Figure 3.1. The output of the CharCNN is fed to a highway network, recently proposed by Srivastava et al. (2015). Similar to the memory cells in LSTM networks, highway layers are fully connected layers modified in order to carry some dimensions of the input directly to the output.

Deriving representations of words from the representation of their characters is motivated by the *unknown word* problem, which is the problem of assigning an embedding to out-of-vocabulary (OOV) words. Working on the level of characters alleviates this problem to a large extent, as the vocabulary of possible characters is much smaller than the vocabulary of possible words. As pointed by Kim et al. (2016), the word representations learned after the

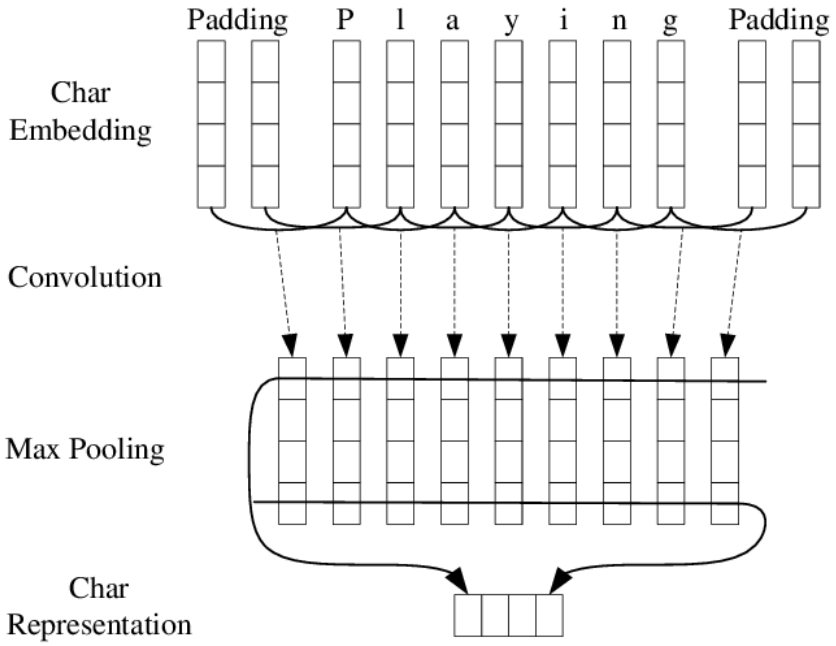

Figure 3.1: The convolution neural network for extracting character-level representations of words

CharCNN and the highway layer seem to enable the encoding of semantic features that are not discernable from orthography alone. The learned representations of OOV words are similar to words with similar meaning and the model is also able to correct misspelled words.

Bidirectional language model

The mechanism of ELMo is based on the representations obtained from a *bidirectional Language Model*. A bidirectional Language Model (biLM) consists of two language models: a forward LM and a backward LM. Both LM take as input a context independent representation $\boldsymbol{x}_k^{LM}$ and passes it through L layers of a LSTM network. Each of these representations, being hidden representations of recurrent neural networks, is context-dependent.

In the j-th layer, the forward LM outputs at each position k a representation $\overrightarrow{\boldsymbol{h}}_{k,j}^{LM}$, which is a representation of the token t_k given the $k-1$ previous tokens. Similarly, at the same position the backward LM outputs a representation $\overleftarrow{\boldsymbol{h}}_{k,j}^{LM}$ that represents the token t_k given the following $N-k$ tokens. The output of the last biLSTM layer is used to predict the token t_k, with a Softmax activation function computed on both directions. The forward

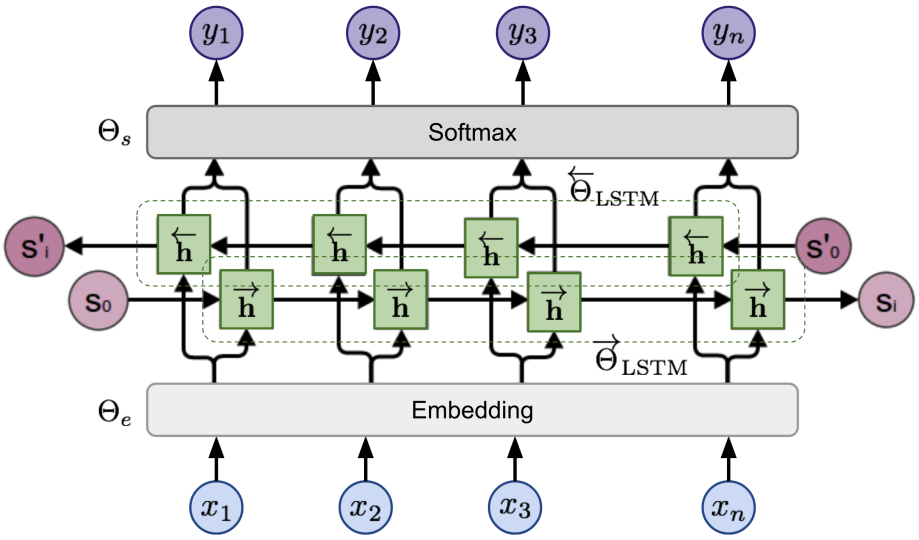

Figure 3.2: The biLSTM base model of ELMo.

and backward LMs share the CharCNN embedding parameters, Θ_e, and the Softmax parameters, Θ_s. The training objective is to maximize jointly the log-likelihood of the forward and the backward directions:

$$\begin{aligned}\mathcal{L} = \sum_{k=1}^{N} &\log\, p(t_k|t_1, ..., t_{k-1}; \Theta_e, \overrightarrow{\Theta}_{\text{LSTM}}, \Theta_s) + \\ &\log\, p(t_k|t_{k+1}, ..., t_N; \Theta_e, \overleftarrow{\Theta}_{\text{LSTM}}, \Theta_s).\end{aligned} \tag{3.2}$$

The biLM architecture is basically a bidirectional LSTM, where the forward and the backward representations are treated as two separate language models and trained jointly. In Figure 3.2 we can see a graphical representation of the biLSTM architecture. For each token t_k the ELMo biLM computes a set of $2L + 1$ representations R_k:

$$\begin{aligned}R_k &= \{\boldsymbol{x}_k^{LM}, \overrightarrow{\boldsymbol{h}}_{k,j}^{LM}, \overleftarrow{\boldsymbol{h}}_{k,j}^{LM} | j = 1, ..., L\} \\ &= \{\boldsymbol{h}_{k,j}^{LM} | j = 0, .., L\},\end{aligned} \tag{3.3}$$

where $\boldsymbol{h}_{k,0}^{LM}$ is the context independent representation and $\boldsymbol{h}_{k,j}^{LM} = [\overrightarrow{\boldsymbol{h}}_{k,j}^{LM}; \overleftarrow{\boldsymbol{h}}_{k,j}^{LM}]$ for each biLSTM layer. As we will see later, each $\boldsymbol{h}_{k,j}^{LM}$ representation captures different information and, depending on the task, we can find to be more useful either the low-level or the high-level representations.

Use ELMo representations in NLP tasks

ELMo is a task specific combination of the intermediate layer representations in the biLM. For inclusion in a supervised task model, ELMo collapses all layers in R_k into a single vector $\mathbf{ELMo}_k^{task}$, which is a task specific weighting of the biLM layers:

$$\mathbf{ELMo}_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \boldsymbol{h}_{k,j}^{LM}. \tag{3.4}$$

If we want to use a pre-trained biLM in a supervised architecture built to solve a target NLP task, we need to proceed in two steps. First, we simply run the biLM and record all of the layer representations for each word. Then, we let the supervised model learn the linear combination of these representations which optimizes the performances on the given task. Alternatively, as the authors did, the ELMo vector can be add to an existing NLP model[1].

The model architecture published by Peters et al. (2018) uses $L = 2$ biLSTM layers. As a result, the biLM provides three layers of representations for each input token, including those outside the training set due to the purely character input.

The ELMo authors showed that adding the biLM linear combination to a state-of-the-art supervised model improves the performances for six different NLP tasks. This means that the biLM's contextual representations must encode information generally useful for NLP tasks that is not captured in other types of word vectors. Moreover, different layers in the biLM represent different types of information: for some task the performance values were higher when including only the first biLM layer, while for some other task the best ones were obtained including only the top biLM layer. Therefore, as the authors suggest, we should use all biLM layers to get the highest performance in supervised tasks.

The presented approach for contextualized word embedding representations, as well as other approaches (i.e. BERT, OpenAI-GPT), promise high quality representations for words. The pre-trained deep language models also provide an advantage for supervised tasks in the form of transfer learning. Whether this will become a standard approach in the NLP community remains to be seen in the future.

[1]The procedure of including the biLM to a supervised model depends both on the task and on the model. In this work we do not deal with how this inclusion is done.

Chapter 4

Experiments

After describing the theory behind text classification and the neural network approach applied to the text classification and language modeling problems, in this Chapter a real-world application of the above-mentioned methods will be presented. The Python programming language has been used to carry out the whole project.

In Section 4.1 the dataset description will be presented, along with the task we want to solve. In Section 4.2 the pre-processing procedure, namely the sequence of algorithms needed to prepare and transform the data (also named pre-processing pipeline) will be presented. Afterwards the training protocol will be described, as explained in Chapter 2, along with the models implemented to solve the task.

In Section 4.3 we will see the results of the data processing and training procedures and the performances of the models on the validation set. In Section 4.4 these results will be discussed in detail.

4.1 Dataset properties

Dataset description

The dataset used in this work is a subset of the CCNL italian contracts. The texts can be found online at the link `https://www.cnel.it/Archivio-Contratti/Contrattazione-Nazionale/Analisi-Avanzate`. In this database, contracts are indexed basing on different variables:

- the category of workers they deal with (farmers, mechanics, public administrators, ecc...);
- their current state, i.e. if they are currently valid or if they are expired;

- various dates: date of stipulation, dates of effectiveness, deadline date;
- type of deal (renewal deal, definitive text, ecc..);
- topic or topics they deal with (health protection, working hours, ecc...).

The variable we are interested is the topics which the contracts deal with. Texts are divided in paragraphs, each of which deals with a different topic. Topics are:

- AL, which stands for *Ambiente Lavoro e tutela salute*;
- CC, which stands for *Contrattazione Collettiva*;
- CR, which stands for *Cessazione del Rapporto*;
- CS, which stands for *Costituzione del rapporto*;
- GE, which stands for *Gestione delle Eccedenze*;
- LL, which stands for *Luogo del Lavoro*;
- MQ, which stands for *Mansioni e Qualifiche*;
- ND, which stands for *Norme Disciplinari*;
- OD, which stands for *Organizzazione del lavoro*;
- OR, which stands for *Orario*;
- RS, which stands for *Rapporti e diritti Sindacali*;
- SR, which stands for *Sospensione del Rapporto*;
- TE, which stands for *Trattamento Economico*;
- AA, which stands for *Altre materie*.

Task definition

At the end we will have a dataset of examples $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ being $\boldsymbol{x}_i$ the preprocessed text and $\boldsymbol{y}_i$ a 14-dimensional vector such that each element $\boldsymbol{y}_{ij}$ is equal to 1 if the i-th document belongs to the class j and 0 otherwise. The goal is to train a classifier that estimates as well as possible for any text $\boldsymbol{x}_i$ the conditional probability that the text belongs to each of the 14 classes. The same paragraph can have multiple labels, as we will see later, so the task at hand is multi-label classification.

4.2 Machine Learning pipeline

Pre-processing

As described in Section 1.3, different text pre-processing operations can be performed. For this specific problem, the most suitable pre-processing pipeline consists in the following steps:

- tokenization;
- lowercasing;
- stop words removal.

In the tokenization phase, strings of text are split into tokens. In this phase, the pre-processing system, based on regular expressions, recognizes different types of symbols: words, numbers, punctuation signs or special symbols. Words are kept as they are while punctuation signs and special symbols are removed, as they are useless. Numbers and sequence of numbers are transformed into special placeholder tokens: every date is turned to the token *xxxdatexxx*, every abbreviations of legislative decrees is turned to the token *xxxdlxxx* and every other number is turned into the token *xxxnumxxx*. The motivation behind this choice is to keep numbers, because they have a meaning within the text. However the meaning of the text would not change if the number changes.

The lowercasing phase follows the tokenization one. As stated previously, it reduces sparsity but it can increase ambiguity. However, the training corpus of a pre-trained model that needs a vocabulary (Word2vec in this case) includes only lowercased tokens. This ambiguity can be solved in Generalized Language Models like ELMo which offer the possibility of computing contextual word embedding vectors.

Lastly, stop words are removed from the texts. The italian stop word list used in this work has been featured by Bird et al. (2009) and can be downloaded using the Python commands that can be found at the following link: `https://www.nltk.org/book/ch02.html`.

Train, validation and test protocol

To split the dataset into the training, validation and test sets we need to take into account the unbalanced label distribution and the possibility that the texts can have multiple labels. Given the task at hand and the considerations above, the implemented protocol can be described as follows:

1. For each class 80% of its observations have been put in the training set and the remaining 20% in the test set;

2. After assigning the observations to the training and the test sets, the texts which are present both in the training and in the test set have been removed from the test;

3. Then, the same splitting procedure is repeated in order to split the training set in two separate sets: the final training set and the validation set.

The second point in the procedure is necessary because observations may have multiple labels and the same observation might be included in the training set when considering one of his labels and in the test set when considering another one. The fact that the training and the test (or the validation) sets share some observations should be avoided because it would introduce some bias and thus the model performances would be influenced by this bias. However, removing the observations with multiple labels from the test (or validation) set implies that the test (and the validation) sets have less observations with multiple labels with respect to the training set. The purpose has been to train classification models which learn (as well as possible) to classify observations that could have multiple labels.

Classification models

The text classifier has been implemented according to different model architectures. The general structure in which every model falls into is:

$$\begin{aligned} M(\boldsymbol{x}_i) &= \boldsymbol{o}_i \\ E(\boldsymbol{x}_i) &= \boldsymbol{E}_i \\ F(\boldsymbol{E}_i) &= \boldsymbol{f}_i \\ MLP1(\boldsymbol{f}_i) &= \boldsymbol{o}_i \\ \hat{\boldsymbol{p}}_i &= sigmoid(\boldsymbol{o}_i) \end{aligned} \tag{4.1}$$

Every model M takes as input a sequence of tokens $\boldsymbol{x}$ belonging to the i-th text and gives as outputs a vector $\boldsymbol{o}$. First, tokens are embedded using the embedding function E resulting in an embedding matrix $\boldsymbol{E}$ that contains an embedding vector for each word in the text. Embedding vectors pass through a feature extraction layer F which outputs a vector $\boldsymbol{f}$. The last part of the model M is a MLP layer which projects the vector $\boldsymbol{f}$ into the output vector $\boldsymbol{o}$ which has the same shape as the label vector $\boldsymbol{y}$. After that, each

element of the output vector is transformed into the range $(0, 1)$ using the sigmoid function. Depending on the embedding and on the feature extraction strategy, different models can be defined.

Embedding strategy

As for the embedding strategy, different approaches have been explored. As a first approach, the automatic word embedding procedure described in Chapter 3 has been implemented. The network learns appropriate word representations starting from randomly initialized embedding vectors. The advantage of this approach consists in specializing the embedding vectors exclusively for a specific task (text classification). This should avoid the creation of more general purpose Word Embedding Models. On the other hand, automatic embedding vectors might not be suitable for inference on text with wider vocabularies.

The second approach consists in using a pre-trained $Word2Vec$ representation for each word. In this case, the embedding layer simply maps every word that is present in the model vocabulary to its embedding representation, and these representations remain fixed during the whole training process. The advantange of this approach consists in relying on a well-established representation of the Italian language based on the Wikipedia corpus. Moreover, Word2Vec provides word vectors which are task-independent, re-usable and simpler than other Word Embedding approaches. This may save training time and makes the hyperparameters optimization strategy easier. The model used for word embedding was provided by the Wikipedia2Vec Python package and it is available at `https://wikipedia2vec.github.io/wikipedia2vec/`. Such model provides both word and entity embedding representations. Only word embedding vectors have been used in this work.

The third approach is the application of the $ELMo$ embedding vectors. As noted in Section 3.3, ELMo is a Generalized Language Model that provides pre-trained contextual representations in which the layer weighs are task dependent, making them suitable for using the transfer learning technique. This approach leads in theory to more quality word embedding representations comparing to the other considered alternatives. The pre-trained Elmo model used is the one implemented by Schuster et al. (2019) for the Italian language. Model weights and parameters, along with the instruction to use ELMo pre-trained representation can be found at `https://github.com/TalSchuster/CrossLingualContextualEmb`. Both Word2Vec and ELMo language models are trained using the Wikipedia Italian corpus.

Feature extraction

For the feature extraction layer, the goal of this work is to compare the performances of Convolutional Neural Networks and Recurrent Neural Networks. In particular, for the CNN the multiple filter size architecture has been used in order to learn meaningful local patterns that give appropriate feature representations for the paragraphs.

On the other hand, the Long-Short-Term Memory RNN has been used. As we saw in Chapter 2, it is an appropriate model for representing sequences and using this feature extractor layer may lead to better results in this specific task.

The models

The combinations of an embedding layer and a feature extractor resulted in 5 different models. The names and the characteristics of the models are listed below:

- The first model uses a trainable embedding layer with a CNN feature extractor. This model is called $TrainableCNN$;
- The second model uses a trainable embedding layer with a LSTM feature extractor. This model is called $TrainableLSTM$;
- The third model uses the Word2Vec representations with a CNN feature extractor. This model is called $Word2VecCNN$;
- The fourth model uses the Word2Vec representations with a LSTM feature extractor. This model is called $Word2VecLSTM$
- The fifth model uses the pre-trained ELMo representations with a LSTM feature extractor. ELMo weights are fine-tuned during training. This model is called $ELMoLSTM$.

The hyperparameters of the five models are reported in the Tables 4.1 - 4.5.

Note that 4 out of 5 models (TrainableCNN, TrainableLSTM, Word2VecCNN and Word2VecLSTM) have the same output dimensions for each layer corresponding to the same purpose. This means that the Embedding output dimension is equal for the 4 models, as well as the feature extractor output dimension. The embedding output dimension of the TrainableCNN and TrainableLSTM models has been set to 300, corresponding to the dimension of the Word2Vec embedding vectors. For the feature extraction layer, since

Table 4.1: TrainableCNN model hyperparameters

TrainableCNN	
Embedding output dimension	300
Embedding dropout	0
Convolution filter sizes	(2,3,4,5)
Number of filters	256
Projection dimension	512
Feature extraction layer dropout	0
Output dimension	14

Table 4.2: TrainableLSTM model hyperparameters

TrainableLSTM	
Embedding output dimension	300
Embedding dropout	0
Bidirectional RNN	True
Feature dimension	256
Projection dimension	None
Feature extraction layer dropout	0
Output dimension	14

the multiple filter CNN has a bigger output than the LSTM recurrent neural network, a $MLP1$ layer has been implemented after the CNN feature extractor in order to project the CNN output to a feature dimension of 512, the same output size of the Bidirectional LSTM layer. This adds more parameters to the CNN feature extractor; however for the CNN layer the total number of parameters remains much lower than the LSTM layer.

Table 4.3: Word2VecCNN model hyperparameters

Word2VecCNN	
Embedding output dimension	300
Embedding dropout	0
Convolution filter sizes	(2,3,4,5)
Number of filters	256
Projection dimension	512
Feature extraction layer dropout	0
Output dimension	14

There are some differences between the ELMoLSTM and the other im-

Table 4.4: Word2VecLSTM model hyperparameters

Word2VecLSTM	
Embedding output dimension	300
Embedding dropout	0
Bidirectional RNN	True
Feature dimension	256
Projection dimension	None
Feature extraction layer dropout	0
Output dimension	14

plemented models. The embedding part of the ELMoLSTM network, as discussed in Chapter 3, is made up of several layers which output an embedding vector with dimension of 1024. Since the ELMo embedding network is very deep, dropout has been used for generalization purposes in the fine-tuning procedure, as suggested by the ELMo authors[1]. Then, the feature extraction strategy is the bidirectional LSTM used in the TrainableLSTM and in the Word2Vec models, with the same hyperparameters. It will be interesting to see if using the ELMo algorithm to produce word representations will lead this model to better performances in the considered task.

Table 4.5: ELMoLSTM model hyperparameters

ELMoLSTM	
Embedding output dimension	1024
Embedding dropout	0.5
Bidirectional RNN	True
Feature dimension	256
Projection dimension	None
Feature extraction layer dropout	0
Output dimension	14

The models are implemented using the AllenNLP framework. It is a research Python library, built on PyTorch, for developing state-of-the-art deep learning models on a wide variety of linguistic tasks. Further details can be found at `https://allennlp.org/`.

[1]As we can see at https://docs.allennlp.org/v1.0.0rc3/tutorials/how˙to/elmo/

Model training

All the five models are trained using the same settings. The most important decisions when training a neural network are:

- the choice of the loss function;
- the choice of the optimization algorithm;
- the use of early stopping.

As for the loss function the binary cross entropy loss has been used. In the multi-label classification case, the binary cross entropy loss value in each observation is the average of the values computed for each class. When computing the average cross entropy value for each observation, the label values can be multiplied by some weights. Actually the label distribution can be unbalanced, so class weights are used to overcome this issue.

As for the optimization algorithm, the Adam optimizer has been used, since it is the most popular and the most effective optimizer when training deep neural networks. The batch size value has been set to 32.

The use of early stopping in neural network training is crucial since usually as the number of epochs increases, the loss function value in the training set decreases but in the validation set it reaches a minimum and then it slightly decreases. So the maximum number of epochs has been set to 20 with a patience value of 4. This means that the best model in terms of validation loss is saved in memory and if the validation loss value does not decrease after 4 epochs, the training procedure is stopped.

The training of the models have been performed on a Virtual Machine Instance on Azure Cloud, powered by the NVIDIA Tesla K80 card. The choice of a GPU reduces significantly the training of deep learning models. In this setting it became necessary when the ELMoLSTM training on a CPU was initiated and its estimated training time was much long.

Prediction strategy

At the end of each model training the vector of predicted probabilities $\hat{\boldsymbol{y}}_i$ has been computed by calculating the output values of the model for every observation in the dataset. To evaluate the model with the appropriate metrics, we need to turn the predicted probabilities, that can take values in the interval $(0, 1)$, to predicted labels.

Considering the multi-label classification problem as multiple (one for every label) binary classification problems, for each label j we consider as

positive examples the texts which belong to that class and negative examples the ones that don't belong to that class. So we can use the strategy described in Equation 2.6 which consists in assigning the i-th observation to the positive class if the estimated probability value is bigger that a fixed threshold value $t_j = 0.5$, which is equal for every label. However this may not be the best solution in a multi-label classification task, since we may have an unbalanced label distribution.

An optimal threshold t_j for every label j has been calculated. The strategy used consists in taking the thresholds t_j that minimize the weighted error rate in the training set:

$$\hat{\boldsymbol{t}} = \underset{t}{\operatorname{argmin}} \frac{1}{\sum_j w_j} \sum_{j=1}^{14} w_j \frac{1}{N} \sum_{i=1}^{N} (y_{ij} - \hat{y}_{ij})^2, \tag{4.2}$$

where $\boldsymbol{w}$ is a vector of weights that serves to overcome the class imbalance and N is the number of observations $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ in the dataset. The prediction $\hat{y}_{ij}$ of the j-th label for the i-th observation is defined as:

$$\hat{y}_{ij} = \begin{cases} 0, & \hat{p}_{ij} < t_j \\ 1, & \hat{p}_{ij} \geq t_j. \end{cases} \tag{4.3}$$

To perform the minimization defined in Equation 4.2 has been used the truncated Newton algorithm, which is a constrained optimization algorithm that exploits gradient information (Nash, 1984).

Classification metrics

The metrics used to evaluate and compare the performances of the models are Accuracy, Precision, Recall and F1-Score.

When evaluating binary classification models, the accuracy metric is defined as the ratio between the number of observation classified correctly and the total number of observations, as in Equation 2.7. When we deal with multi-label classifiers, it may happen that the model predicts correctly only one label for some observations that should be associated with two correct labels, or it may predict an extra label for some other observations. If we apply the same definition of accuracy, we consider as observations classified correctly the observations in which all their labels are predicted correctly and no extra label is predicted. We can call *HardAccuracy* the metric defined as the ratio between the number of correctly classified observations and the number of total observations:

$$HardAccuracy = \frac{|\{i \,:\, \hat{y}_{ij} = y_{ij},\, \forall j\}|}{N} \tag{4.4}$$

Precision and Recall are computed for each label. As in Equation 2.8, Precision is defined as the ratio between the number of true predicted positive observations and the number of predicted positive observations. On the other hand, as in Equation 2.9, Recall is defined as the ratio between the number of true predicted positive observations and the number of positive observations in the dataset. The F1-Score can also be computed for each label taking the harmonic mean between Precision and Recall. For each of these three metrics we can also calculate the weighted average of the metric values for each label.

4.3 Results

Dataset analysis

After text pre-processing, some descriptive analysis have been performed on the dataset: the analysis of the distribution of the number of words and the analysis of the distribution of labels. These two analysis suggest some crucial decisions to make in the machine learning pipeline.

Figure 4.1 shows the distribution of the number of words in the dataset. As we can see from the figure, the majority of paragraphs have less than 200 words, while few of them have more than a few hundred of words. As a matter of fact, only 25% of the texts have more than 200 words. This could mean that some contracts are split wrongly, resulting in texts matching contracts instead of paragraphs. Therefore some texts cannot be considered moving forward. The maximum number of words has been set to a threshold $t = 300$ for each preprocessed text. Each text that, after pre-processing, has more than 300 words is excluded from the dataset. Moreover, after performing some preliminary tests, if the threshold t was set to a bigger value than 300, or not set at all, it resulted in models that require an unfeasible RAM space, even if the batch size was set to a pretty low value.

Figure 4.2 shows the distribution of labels in the dataset. As we can see the distribution is highly unbalanced: the classes TE and CS are highly represented, while the classes OD and GE have very few examples. Therefore some adjustments need to be made in the training procedure, like applying some weights in the loss function, to overcome this unbalance. Figure 4.3 shows the distribution of labels after the train, validation, test split. As we can see, the proportion of labels in the dataset is roughly the same as the one in the three sets.

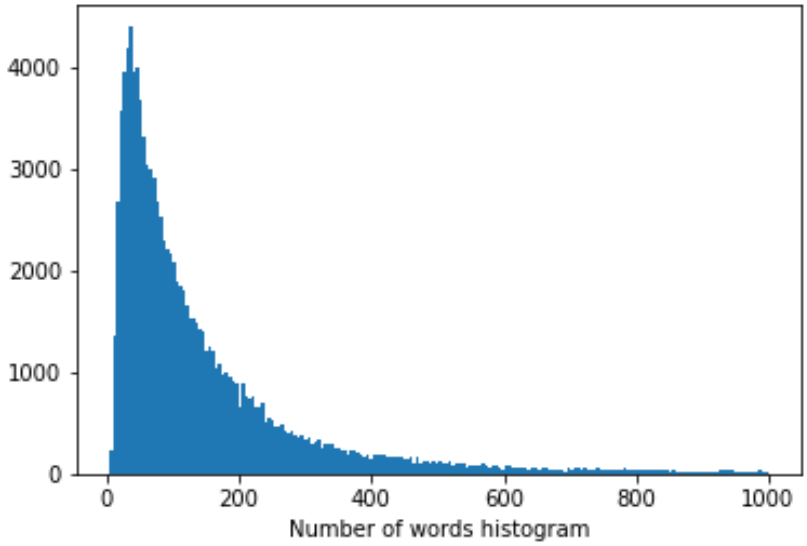

Figure 4.1: Distribution of the number of words for each text in the dataset

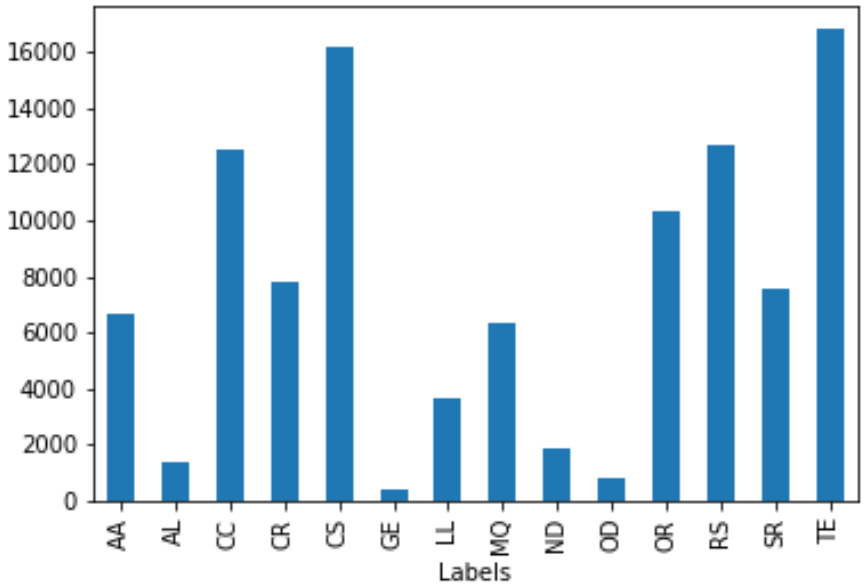

Figure 4.2: Distribution of the labels in the dataset

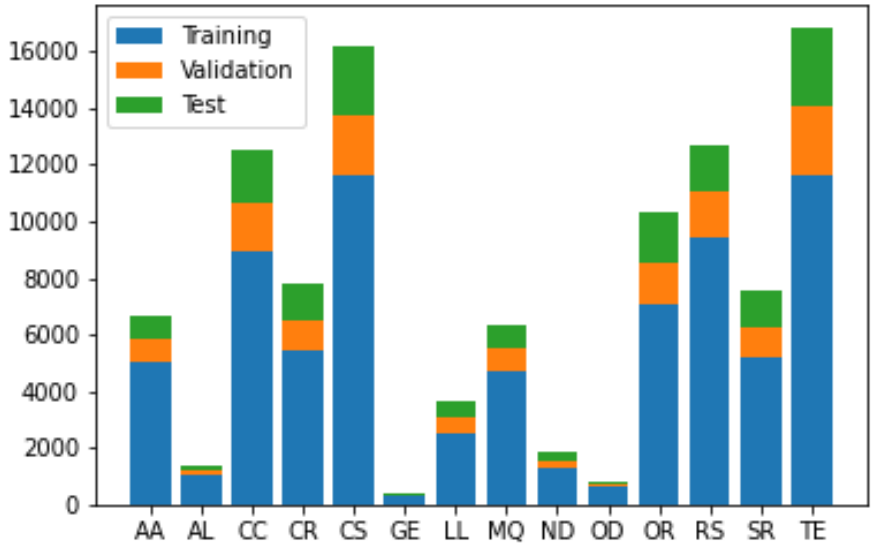

Figure 4.3: Distribution of the labels in the dataset after the train,validation,test split

Table 4.6: Distribution of the number of labels for each text in the dataset

	Training set	**Validation set**	**Test set**	**Whole dataset**
One label	46416	11611	14489	72516
Two labels	12409	1262	619	14290
Three or more labels	1186	43	6	1235
Total examples	60011	12916	15114	88041

It is interesting also to look at the distribution of the number of labels per paragraph. Table 4.6 shows that the majority of examples have only one label, as in the case of multi-class classification. Moreover this table shows that in the validation and in the test sets the number of observations with multiple labels is significantly smaller than in the training set. This may happen because in the training, validation, test split we always remove from the validation and test sets multi-label observations that are already present in the training set. However, this choice is reasonable since multi-

label observations are very few and putting the majority of these in the training set could lead to a more robust multi-label classifier.

Weighting Scheme

As already mentioned, the observations that belong to less represented classes are weighted more than the observations that belong to the most represented ones. The most natural choice for the weights is the inverse of the ratio between the number of observations of the given label and the number of observations of the most represented label. For each class some a-posteriori grid scan around such values has been performed, evaluating how the metrics values changed in the validation set. The best values have been reported in Table 4.7.

Table 4.7: Optimal weight values in the loss function for every label

	Inverse ratios	**Weights**
AA	2.31	2.00
AL	10.68	10.0
CC	1.30	1.20
CR	2.14	2.00
CS	1.00	1.00
GE	36.33	25.00
LL	4.60	2.00
MQ	2.47	2.00
ND	8.80	8.00
OD	18.40	12.00
OR	1.64	1.00
RS	1.24	1.20
SR	2.24	1.00
TE	1.00	1.00

Training and prediction results

The five models have been trained applying the weights above to the cross entropy loss function. As already mentioned, Early Stopping has been applied, setting the maximum number of epochs to 20 and the patience to 4. Figures 4.4 to 4.8 report the trend of the loss function during the training

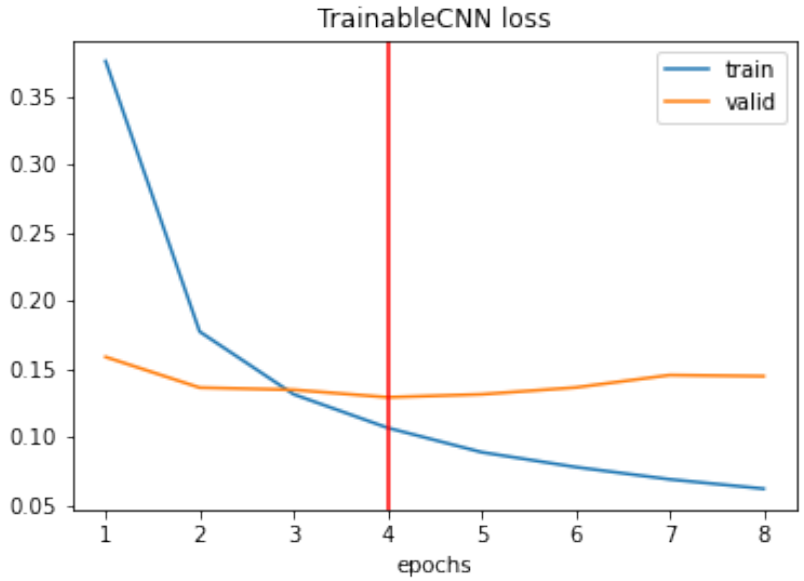

Figure 4.4: Trend of the loss function during the training of the Trainable CNN model

of the five models. In each Figure, the trend of the loss function calculated in the training set is compared with that of the loss function calculated in the validation set. The vertical red line corresponds to the epoch in which the validation loss reaches a minimum. The model trained at the end of this epoch is considered to be the best one and it is the one that will be evaluated using the classification metrics. We can see that all five models converge to a validation loss value around 0.1 and that in all models but ELMoLSTM the training loss value is lower than the validation loss value even before the cutoff epoch.

The training time for each model is reported in Table 4.8, along with the cutoff epoch and the total number of epochs. As we can see for the single layer Embedding models the training time is of the order of tens of minutes. The models using the LSTM feature extractor took more than the ones using the CNN feature extractor. On the other hand, the ELMoLSTM model took nearly 11 and a half hours.

Table 4.9 reports the optimal probability threshold values for each label computed after the error rate minimization over all the possible threshold values. After computing these thresholds, the model assigns every observation i to the class j if the a-posteriori probability $\hat{p}_{ij}$ is above the threshold t_j. Looking at the values on the table we can clearly see that for the same class

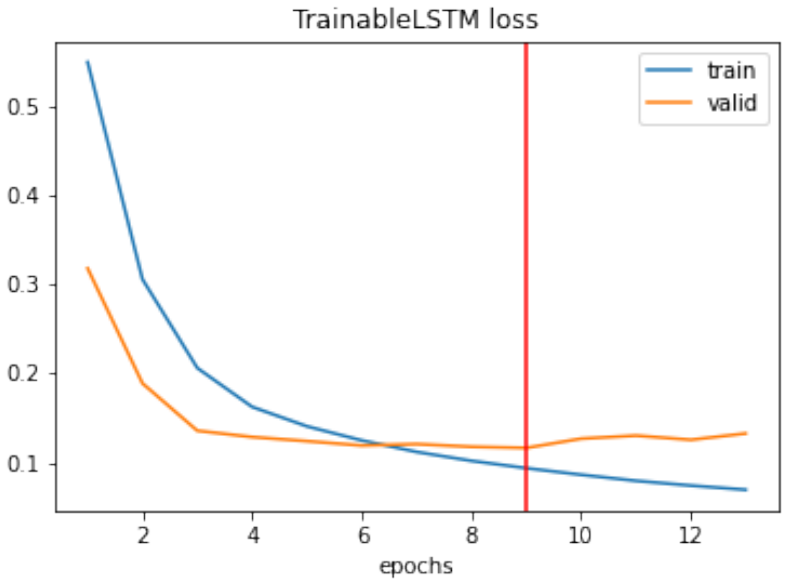

Figure 4.5: Trend of the loss function during the training of the Trainable LSTM model

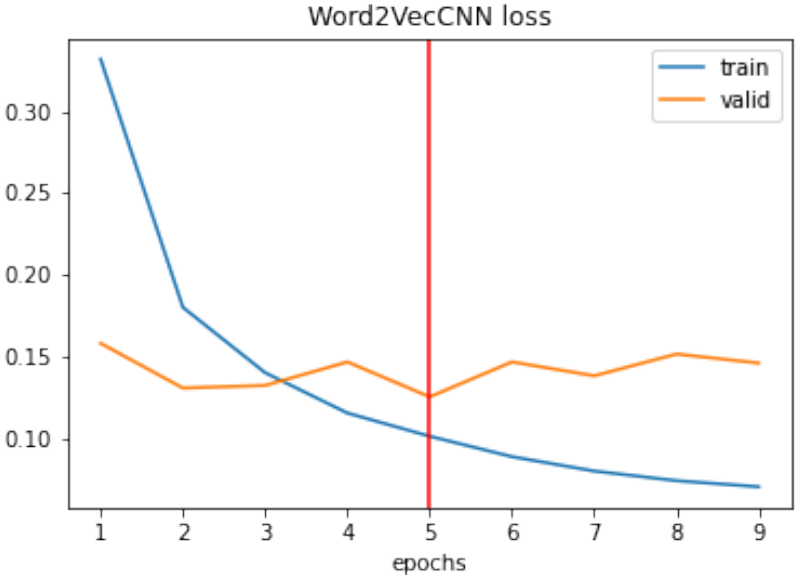

Figure 4.6: Trend of the loss function during the training of the Word2VecCNN model

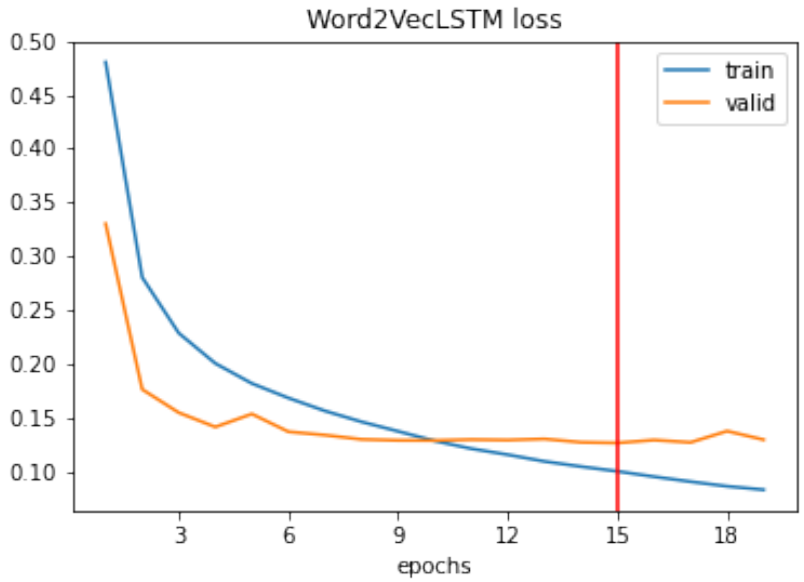

Figure 4.7: Trend of the loss function during the training of the Word2VecLSTM model

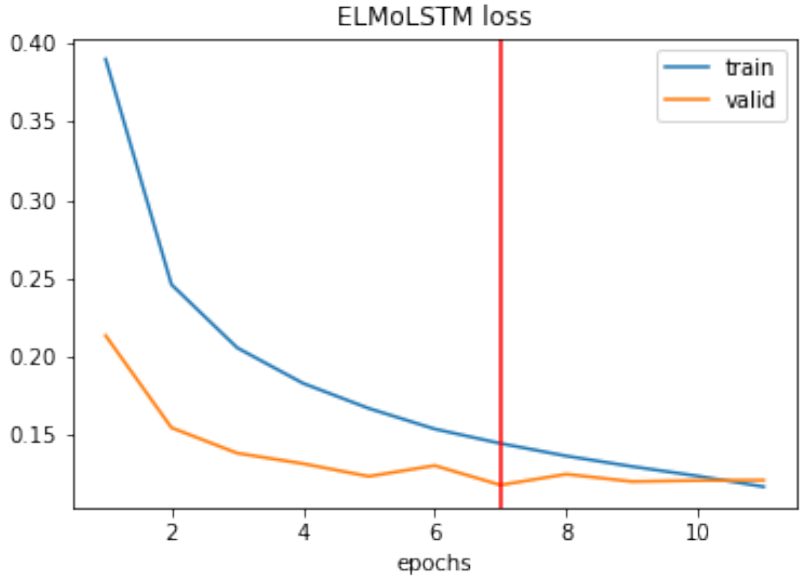

Figure 4.8: Trend of the loss function during the training of the ELMoLSTM model

Table 4.8: Training time and number of epochs for each model

	Cutoff epoch	**Total Epochs**	**Training time**
Word2VecCNN	5	9	7 minutes
TrainableCNN	4	8	13 minutes
TrainableLSTM	9	13	31 minutes
Word2VecLSTM	15	19	42 minutes
ELMoLSTM	7	11	11 hours 25 minutes

some models have very different optimal thresholds. If we take, for example, the class SR we note that two models have a threshold value for this class very low (< 0.3) with respect to the others (roughly 0.74 taking the mean of the three values). Higher threshold values lead to bigger precision, as most of the predicted observations will be predicted correctly. But having higher threshold values leads to lower recall, as we will have very few observations which are predicted to be positive. On the contrary, low threshold values lead to more observations that are predicted to be positive and this leads to lower precision and higher recall.

Table 4.9: Threshold values for each topic category

	Trainable CNN	**Trainable LSTM**	**Word2Vec CNN**	**Word2Vec LSTM**	**ELMo LSTM**
AA	0.168	0.493	0.620	0.414	0.597
AL	0.459	0.409	0.410	0.412	0.383
CC	0.374	0.470	0.576	0.508	0.478
CR	0.379	0.611	0.418	0.357	0.726
CS	0.378	0.416	0.217	0.535	0.526
GE	0.585	0.471	0.337	0.511	0.712
LL	0.619	0.234	0.687	0.339	0.335
MQ	0.637	0.544	0.626	0.362	0.449
ND	0.291	0.289	0.390	0.506	0.480
OD	0.624	0.763	0.096	0.814	0.548
OR	0.595	0.467	0.517	0.434	0.455
RS	0.554	0.471	0.220	0.472	0.607
SR	0.755	0.238	0.787	0.294	0.669
TE	0.450	0.426	0.399	0.485	0.555

Classification models comparison

Table 4.10 reports the validation precision values for each topic category. The model with the highest average precision on the validation set is the ELMoLSTM, but the difference with the other models is not large. In the majority of classes the ELMoLSTM model shows a better precision. However, for the CR and RS class the TrainableLSTM model has way better precision than the other models. In general, all the models have good performances on each class excluding the class OD and the class GE, which are the less represented ones.

Table 4.10: Validation set precision for each topic category

	Trainable CNN	**Trainable LSTM**	**Word2Vec CNN**	**Word2Vec LSTM**	**ELMo LSTM**
AA	0.865	0.860	0.827	**0.878**	0.866
AL	0.828	0.803	0.848	0.799	**0.855**
CC	0.853	0.868	0.834	0.872	**0.903**
CR	0.899	**0.916**	0.864	0.907	0.895
CS	0.832	0.842	**0.860**	0.809	0.831
GE	0.556	0.553	0.577	0.559	**0.652**
LL	**0.909**	0.904	0.891	0.896	0.879
MQ	0.807	0.817	0.805	**0.840**	0.836
ND	0.871	0.874	0.864	0.878	**0.879**
OD	0.584	0.600	0.632	0.554	**0.784**
OR	0.919	0.915	**0.919**	0.918	0.914
RS	0.798	**0.853**	0.815	0.821	0.788
SR	0.898	0.900	0.916	0.923	**0.936**
TE	0.884	0.890	0.903	**0.910**	0.904
Weighted average	0.861	0.872	0.864	0.871	**0.873**

Table 4.11 reports the validation recall values for each topic category. We can see that the best model in terms of validation set recall is the TrainableCNN, but the difference with the other models is not very large. Looking at the recall values for each label no model stands out with respect to the others. In general we can see that all models have a good recall score, except for the class OD and the class GE.

Table 4.12 reports the validation F1 values for each topic category. Since the F1-score is the harmonic mean between Precision and Recall, it is a more meaningful metric for model evaluation and model comparisons. The best

Table 4.11: Validation set recall for each topic category

	Trainable CNN	**Trainable LSTM**	**Word2Vec CNN**	**Word2Vec LSTM**	**ELMo LSTM**
AA	0.838	0.871	**0.873**	0.829	0.827
AL	**0.862**	0.856	0.838	0.856	0.844
CC	0.865	0.882	**0.893**	0.873	0.863
CR	0.886	0.903	**0.911**	0.899	0.894
CS	**0.888**	0.868	0.858	0.864	0.841
GE	0.606	**0.636**	0.455	0.576	0.455
LL	0.897	0.895	0.889	0.876	**0.903**
MQ	**0.873**	0.846	0.851	0.834	0.816
ND	0.934	0.934	0.934	**0.946**	0.927
OD	0.577	0.462	**0.615**	0.462	0.513
OR	0.937	**0.948**	0.936	0.937	0.930
RS	**0.888**	0.855	0.870	0.859	0.883
SR	0.916	**0.921**	0.917	0.906	0.898
TE	**0.933**	0.930	0.918	0.919	0.914
Weighted average	**0.895**	0.892	0.891	0.883	0.877

model in terms of validation set F1 is the TrainableLSTM, but this table clearly shows that the five models are not significantly different in terms of performance values.

Table 4.13 reports the Hard Accuracy values for the validation set. We can clearly see that all the five models can predict correctly the label in observations that have only one label, and they struggle with observations that have more than one. As for the accuracy metric we cannot see significant differences between the models.

Although, in general, the TrainableLSTM model performances look higher than the other classifiers ones, the difference between this model and other competitors is not huge. In Table 4.14 the average metrics computed in the test set for all models has been reported. As we can see, the TrainableLSTM model confirms the best performances in this scenario.

4.4 Discussion

For all NLP applications, including text classification, the procedure is twofold. First, the goal is to get a good text representation for each observation. The

Table 4.12: Validation set f1 for each topic category

	Trainable CNN	**Trainable LSTM**	**Word2Vec CNN**	**Word2Vec LSTM**	**ELMo LSTM**
AA	0.851	**0.866**	0.849	0.853	0.846
AL	0.845	0.829	0.843	0.827	**0.849**
CC	0.859	0.875	0.863	0.873	**0.882**
CR	0.893	**0.910**	0.887	0.903	0.895
CS	**0.859**	0.855	**0.859**	0.836	0.836
GE	0.580	**0.592**	0.508	0.567	0.536
LL	**0.903**	0.899	0.890	0.886	0.891
MQ	**0.839**	0.831	0.828	0.837	0.826
ND	0.901	0.903	0.898	**0.911**	0.902
OD	0.581	0.522	**0.623**	0.503	0.620
OR	0.928	**0.931**	0.927	0.928	0.922
RS	0.841	**0.854**	0.842	0.839	0.833
SR	0.907	0.910	**0.917**	0.915	**0.917**
TE	0.907	0.909	0.910	**0.914**	0.909
Weighted average	0.877	**0.882**	0.877	0.876	0.875

Table 4.13: Hard accuracy in the validation set

	Trainable CNN	**Trainable LSTM**	**Word2Vec CNN**	**Word2Vec LSTM**	**ELMo LSTM**
One label	0.845	**0.864**	0.849	0.858	0.861
Two labels	**0.451**	0.377	0.432	0.350	0.340
Three or more labels	**0.093**	0.070	0.070	0.070	0.023
Weighted average	0.804	**0.814**	0.806	0.806	0.807

following step it to build a machine learning model to solves as best as possible the task at hand. We saw in Chapter 3 that Language Models can be trained to generate semantic word representations, which can be then used as inputs of a supervised machine learning models. The ELMo Language Model has undoubtedly some strong points with respect to some competitors. It offers the possibility of learning contextual word embedding vectors that depend on the whole sentence, with a bidirectional approach. Moreover, the word representations in the first biLM layer are computed by character

Table 4.14: Test set aggregate metrics

	Trainable CNN	**Trainable LSTM**	**Word2Vec CNN**	**Word2Vec LSTM**	**ELMo LSTM**
Hard Accuracy	0.823	**0.839**	0.831	**0.839**	0.837
Precision	0.852	0.865	0.861	**0.870**	0.867
Recall	0.920	**0.923**	0.918	0.912	0.908
F1-score	0.884	**0.893**	0.888	0.890	0.887

convolution. This allows not to depend on a vocabulary and to generate meaningful representations of words that are not included in the training set. Finally, the word representations are task dependent, which makes this model suitable for using transfer-learning techniques. Theoretically, for a text classification task the best solution consists in a neural network that is trained to explore both the fine-tuning of the ELMo model and to transform the ELMo outputs to a vector that represents each text, using for example a LSTM layer followed by a MLP one.

Looking at the results from the experiments, we saw that using LSTM layers for extracting document-level features leads to better performances than using CNN layers. The ELMoLSTM model results, on average, can be comparable to the ones of the TrainableLSTM model, which seems to be the best model in our example. In practice the difference in performances with simpler solutions, like models using Trainable embedding layers or pre-trained Word2Vec representations, is not very large, especially in a typical business scenario, where costs and implementation time should be evaluated before carrying out an algorithmic solution.

In the dataset lots of examples turned out to be semantically similar and this might turn the task to be suitable even for simpler models. The ELMoLSTM model adds a remarkable complexity, leading to higher training duration of each epoch. However, we can see that that ELMoLSTM model took less epochs to reach convergence, comparing to the ones of the TrainableLSTM and the Word2VecLSTM models. This may suggest that in more challenging conditions, like having less observations, or more lexical variety or solving a more challenging task (i.e. question answering or language translation) the model using the ELMo representations would have performed better with respect to models using simpler word representations.

From a business point of view, one of the most important aspects of a classification model is the training time. Training the ELMoLSTM model took way more than the other models, nearly 12 hours with a GPU support.

This means that if we need to perform multiple training to tweak some hyperparameters, it might take days to get the final model. As expected, the ELMoLSTM model is the most resource-consuming among the investigated ones and this strongly affects the timing performance. Indeed the ELMoLSTM model got the highest training duration. However similar duration can be observed when using the transfer-learning technique on the ELMo word embedding representations for all kinds of datasets. On the other hand, the Word2Vec Language Model doesn't build semantic representations for words that are not included in the vocabulary. When working with a dataset that includes very domain specific words, before using the Word2Vec representations for our task, we might need to re-perform the Language Model training including our dataset and some samples from the Wikipedia corpus, and this operation is very costly.

Another remarkable aspect consists in the fact that the labels were set by humans based on their subjective judgment whose implicit criteria may impact the dataset quality. Human influence is crucial in determining the dataset quality, especially if the label setting operation is done by an heterogeneous sample of human evaluators. The introduction of some noise in the dataset is unavoidable in non-exclusive multi-label classification tasks, and this may have an impact on the classifier performance. Therefore, in this pre-processing setting, significantly higher performances than these might be hardly reached.

In the end, in a business scenario different criteria can be applied when choosing the most suitable model for the task at hand. Often in this evaluation different factors have to be taken into account. Costs, resource consumption, implementation time and training time affect the final algorithmic choice. For this text classification task, the classifier using the ELMo Language Model (combined with an LSTM feature extractor) leads to performances similar to the other classifiers, which are simpler but still reliable solutions. Its significant computational time represents a weakness when considered together with its performances in this task scenario. But, as stated before, its use could be more suitable for more complex tasks and it may lead to a significant increase in performances with respect to simpler models. The choice of the most suitable algorithm in real-word applications is always the result of many trade-offs.

Chapter 5

Conclusions

The goal of this work was twofold. On one hand, it served to present some existing literature regarding the Natural Language Processing field and how Deep Learning models can be useful to solve the word embedding and the text classification tasks. On the other hand, it showed how the presented models can be applied to the real-world application of classifying an Italian dataset with the additional constraint that each observation could be assigned to multiple labels.

The theoretical concepts presentation started in Chapter 1 from some basic and well known concepts in the literature about Text Classification, text pre-processing methods and feature design for text classification problems. Then, in Chapter 2 the focal point was to introduce the topic of supervised text classification and the main deep learning models that can be used to solve this task. We saw that Convolutional Neural Networks, in particular the multiple filter size variant, and Recurrent Neural Networks, in particular the LSTM Networks, have been used in the past on similar NLP tasks for their ability to extract useful information on sequences of data. Finally, Chapter 3 focuses on some strategies for the problem of word embedding. First, the difference between the automatic embedding tuning and the pre-training of word representations has been described. Then, two popular word embedding algorithms have been presented. Word2Vec is a class of algorithms based on Language Modeling that can be used to compute semantic representations for words in a fixed vocabulary. ELMo, on the other hand, belongs to a class of algorithms called Generalized Language Models which objective is to compute contextual word representations.

Chapter 4 goes to the heart of a text classification application, explaining in deep the various phases that lead to the choice of the best solution for a given task. In particular, the analyzed documents were a set of Italian contracts that could be classified into a set of overlapping categories. All the

necessaries operations of pre-processing and data analysis were taken into consideration and in particular some considerations needed to be made about the number of words distribution, the class imbalance and the procedure to split data in the training, validation and test sets. Five different deep learning models have been trained to solve the task, and then their performances have been compared using some classification metrics and measuring the time required to perform the training. Looking at the results, the model using the ELMo word representations performs similarly to the competitors, while taking significantly more time to be trained. The trainable embedding layer seems the best embedding strategy for the considered problem. The last consideration regards the LSTM network which can be preferable over the CNN for the feature extraction operation.

Overall this work showed the power of the Deep Learning approach applied to the Text Classification task. Some popular models in the literature have been successfully applied in a challenging real-world scenario. The detailed analysis of the trained models has been performed, in order to decide if each model could be suitable for solving similar problems in a business scenario.

Appendix A

Appendix for chapter 2

A.1 Supervised classification

Loss functions

When training a machine learning model, we are trying to make the predictions of the model $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ as accurate as possible w.r.t. the true labels $\boldsymbol{y}$. For this scope, we introduce the notion of *loss function*, quantifying the loss suffered when predicting $\hat{\boldsymbol{y}}$ while the true label is $\boldsymbol{y}$. Formally, a loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ assigns a numerical score (a scalar) to a predicted output $\hat{\boldsymbol{y}}$ given the true expected output $\boldsymbol{y}$. The estimated parameters Θ of the learned function are the ones which minimize the loss L over the training examples:

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i) \tag{A.1}$$

Sometimes, to avoid *overfitting* the training data, we add to the loss a *regularization* term $R(\Theta)$ that takes as input the parameters and returns a scalar that reflects their "complexity", which we want to keep low. By adding R to the objective, the optimization problem needs to balance between low loss and low complexity:

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i) + \lambda R(\Theta) \tag{A.2}$$

Different combinations of loss functions and regularization criteria result in different learning algorithms, with different inductive biases.

The most common loss function when dealing with classification problems with conditional probability outputs is the Cross Entropy. In the binary case when the classifier's output is transformed using the sigmoid function defined

in Equation 2.3 it is interpreted as the conditional probability $\hat{y} = \sigma(f(\boldsymbol{x}) = P(y = 1|\boldsymbol{x})$. Thus, the classifier is trained to maximize the conditional probability for each training example $(\boldsymbol{x}_i, y_i)$. The binary cross entropy loss is defined as:

$$L_{logistic}(\hat{y}_i, y_i) = -y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i) \tag{A.3}$$

When solving a multi-label classification problem, since it can be viewed as solving multiple binary classification problems, the binary cross entropy loss can be computed for every label and then the final value of the loss function is the average of this values. The same reasoning can be used when solving a multi-class classification problem, but in this case the outputs of the classifier are transformed using the softmax function defined in Equation 2.4.

Gradient-based optimization

In order to train the classifier, we need to solve the optimization problem in Equation A.2. The common solution is to use a gradient based method. The most simple one is the Stochastic Gradient Descent algorithm (SGD), by Bottou (1998). A variant of the SGD algorithm is the Adam algorithm. For a complete motivation of the algorithm explanation we refer to the original article (Kingma and Ba, 2014). The Adam algorithm, or one of its variants, is the most used nowadays when training deep learning models.

SGD and all its variants (including Adam) work by repeatedly sampling a batch of training examples and updating the parameter estimation using the information given by the gradient computed on the sample. The training process is divided in epochs, where an *epoch* refers to one cycle through the full training dataset. Usually, training a neural network takes more than a few epochs.

A.2 Feed Forward Architecture

Optimization in neural networks

Backpropagation

When performing optimization over the set of possible neural networks parameters, the loss function is not convex. Still, gradient-based methods can be applied and they work very well in practice. For complex networks the gradient computation can be laborious and error prone. Fortunately, gradients

can be efficiently and automatically computed using the *backpropagation algorithm* (Rumelhart et al., 1986). The backpropagation algorithm is a fancy name for methodically computing the derivatives of a complex expression using the chain-rule, while caching intermediate results. Once the gradient computation is taken care of, the network is trained using SGD or another gradient-based optimization algorithm, like Adam.

Dropout

Multi-layer networks can be large and have many parameters, making them especially prone to overfitting. A regularization term can be added to the loss function like we saw in Equation A.2. Another effective technique for preventing neural networks from overfitting the training data is *dropout*. It is designed to prevent the network from learning to rely on specific weights. It works by dropping (setting to 0) some of the neurons in each training example during an iteration of the gradient-based optimization. The dropout technique is very effective in deep neural networks to avoid overfitting and it is vastly used also in NLP applications.

Early stopping

A major challenge in training neural networks is how long to train them. When training a large network, there will be a point during training when the model will stop generalizing and start learning the statistical noise in the dataset. The challenge is to train the network long enough such that it is capable of learning the mapping from inputs to outputs avoiding overfitting.

A possible approach is to train the model for a large number of epochs, evaluating its performance on a validation set and saving the model after each epoch. If the performance of the model on the validation set starts to degrade, then the training procedure is stopped and only the best model is kept. This strategy is known as *early stopping*.

When using early stopping, first we need to decide the performance metric to monitor during training. It is common to use the loss on validation set as the metric to monitor, but we may also use a specific metric, for example the accuracy metric in the case of a classification model. Secondly, we need to decide when to stop training. It can be stopped as soon as the performance metric computed on the validation set gets worse. However some delay or *patience* in stopping is always a good idea. The validation error can still go further down after it has begun to increase (Prechelt, 1996).

Early stopping is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its

simplicity (Goodfellow et al., 2016).

A.3 Convolutional Neural Networks

Basic convolutional architecture

Convolutional layers

The convolution operation in NLP is the 1D-convolution, since the inputs are 1 dimensional (sequences). Call $\boldsymbol{w}_i$ the embedding vector of the i-th word. A 1D convolution of width k works by moving a slicing window of size k over the sentence, and applying the same filter to each window in the sequence. A filter is a dot-product between the window and a weight vector $\boldsymbol{u}$, which is often followed by a nonlinear activation function g. Define the operator $\boldsymbol{x}_i = \oplus(\boldsymbol{w}_{i:i+k-1})$ to be the concatenation of the vectors $\boldsymbol{w}_i, ..., \boldsymbol{w}_{i+k-1}$. We then apply the filter to each window vector, having as output the scalar value $p_i = g(\boldsymbol{x}_i \cdot \boldsymbol{u})$. It is customary to use l different filters which can be arranged into the matrix $\boldsymbol{U}$, and a bias vector $\boldsymbol{b}$ is often added. The 1D convolution function can be expressed as:

$$\boldsymbol{p}_i = g((\boldsymbol{x}_i \cdot \boldsymbol{U} + b). \tag{A.4}$$

Each vector $\boldsymbol{p}_i$ is a collection of l values that represents the i-th window. Ideally, each dimension captures a different kind of indicative information. For a sequence of length n with a window of size k there are $n-k+1$ positions in which to start the sequence, and we get $n - k + 1$ vectors. Alternately, we can pad the sentence with $k - 1$ padding symbols to each side, resulting in $n + k + 1$ vectors. Figure A.1 shows an example of a convolutional layer without sequence padding.

Pooling layers

Applying the convolution operation over the text representations results in m vectors of dimension l. These vectors are then combined (*pooled*) into a single vector $\boldsymbol{c}$ of dimension l representing the entire sequence.

The most common pooling operation is *max pooling*, taking the maximum value across each dimension. Calling $\boldsymbol{c}_{[j]}$ the j-th element of the vector $\boldsymbol{c}$, max pooling is defined as:

$$\boldsymbol{c}_{[j]} = \max_{1<i\leq m} \boldsymbol{p}_{i[j]}. \tag{A.5}$$

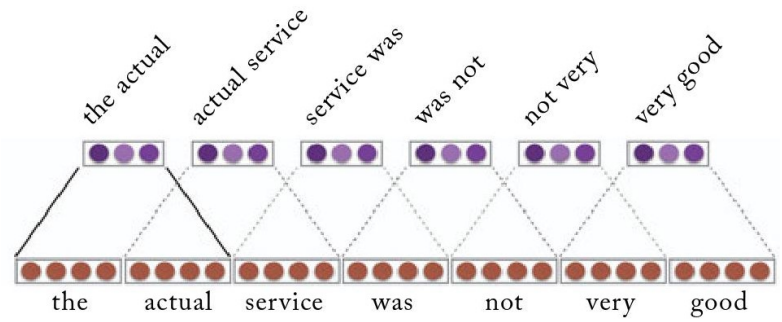

Figure A.1: A convolution with a window of size $k = 2$ and output $l = 3$

The effect of max pooling is to get the most salient information across various window positions. Ideally, each dimension will specialize in a particular sort of predictors, and the max operation will pick the most important predictor of each type.

A common alternative to max pooling in *average pooling*, taking the average value of each index instead of the max:

$$\boldsymbol{c}_{[j]} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{p}_{i[j]}. \tag{A.6}$$

For most tasks max-pooling is used, but there is no guarantee that it is the best pooling operation in practice. One can experiment average pooling or other variations and compare the results.

Appendix B

Appendix for chapter 3

B.1 Language Modeling

Neural language models

Nonlinear neural network models allow conditioning on increasingly large context sizes and they support generalization across different contexts. The presented model was popularized by Bengio et al. (2003). The input to the neural network is the k-gram of words $w_1, ..., w_k$, and the output is a probability distribution over the next word. The k context words $w_1, ..., w_k$ are treated as a word window: each word w is associated with an embedding vector $v(w) \in \mathbb{R}^d$, and the input vector $\boldsymbol{x}$ is a concatenation of the k word vectors:

$$\boldsymbol{x} = [v(w_1); ...; v(w_k)]. \tag{B.1}$$

The input $\boldsymbol{x}$ is then fed to a MLP with one or more hidden layers, like in Equation 2.12. Then, the softmax activation function is applied to the output of the network. The model vocabulary V includes the set of possible words, along with special symbols for unknown words. The k words are used as features, and the word that follows is used as the target label for the classification task. The model can be trained using the cross-entropy loss. This works well, but it requires the use of a costly softmax operation which can be prohibitive for large vocabularies. Language Models can be trained on *raw text*: for training a k-order language model we just need to extract $(k+1)$-grams from the text, and treat the $(k+1)$th word as the supervision signal. The model architecture is shown in Figure B.1.

The parameters of the model are associated with individual words and words in different positions share parameters, making them share statisti-

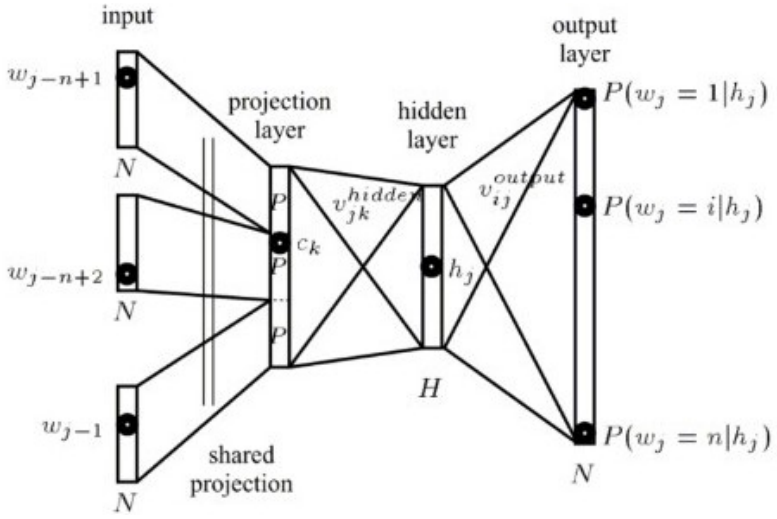

Figure B.1: A neural language model with one projection layer and one hidden layer

cal strength. The hidden layers are in charge of finding informative word combinations and can, in theory, learn informative sub-parts of the $k$grams and do it in a context-dependent way. Another appealing property of the model is the ability to generalize across contexts. By observing that words like *white, blue, red, black, etc.* appear in similar contexts, the model will bee able to assign a reasonable score to the event *blue car* even though it never observed this combination in training, because it observed *red car* and *black car*. The combination of these properties makes it very easy to increase the size of conditioning contexts without suffering much from data sparsity and computational efficiency.

Each of the vocabulary words is associated with one d-dimensional row vector, a row of the embedding matrix $\boldsymbol{E}$, and a column vector in the matrix $\boldsymbol{W}$ of the last MLP layer. During the final score computation, each column in $\boldsymbol{W}$ is multiplied by the context representation, and this produces the score of the corresponding vocabulary item. Intuitively, this should cause words that are likely to appear in similar contexts to have similar vectors. Following the distributional hypothesis, words with similar meanings will have similar vectors. A similar argument can be made about the rows of the matrix $\boldsymbol{E}$, which are the word embedding vectors. Hence, language modeling can be treated as an unsupervised approach for computing word embedding vectors as a byproduct of the training procedure.

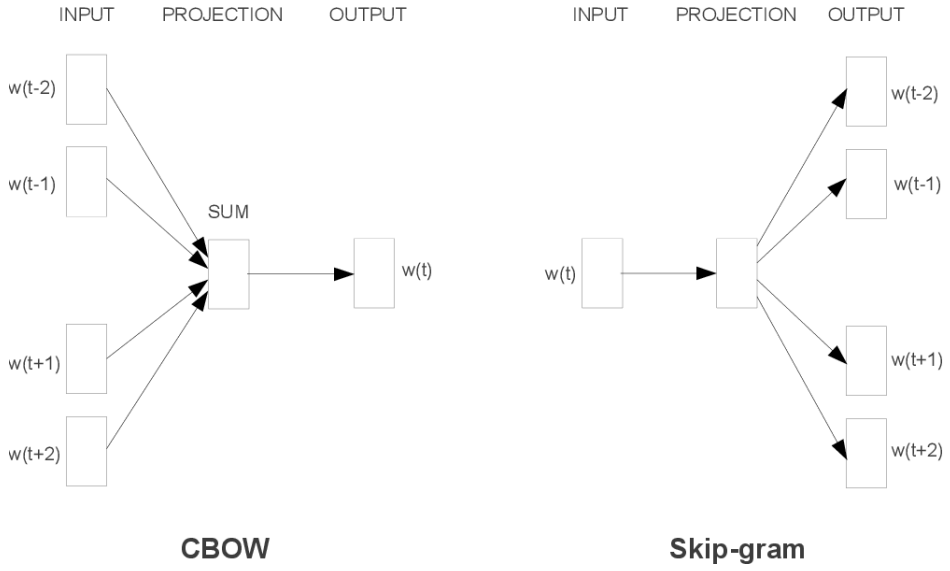

Figure B.2: The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

B.2 Word2Vec

CBOW vs Skip-Gram

For a multi-word context $c_1, ..c_k$, the CBOW variant of Word2Vec defines the score vector to be the sum of the context components, and the context-word score to be the dot product between the word and the context vectors:

$$\begin{aligned} \boldsymbol{c} &= \sum_{i=1}^{k} \boldsymbol{c}_i \\ s(w; c_1, ..c_k) &= \boldsymbol{w} \cdot \boldsymbol{c} \end{aligned} \tag{B.2}$$

Note that the CBOW variant loses the order information between the contexts elements. In the Skip-gram variant, the k elements of the context c_i are assumed to be independent, and they are treated as k different contexts: $(w, c_1), ..., (w, c_k)$. The scoring function is defined as in CBOW, but now each context is a single embedding vector. Hence, the likelihood of the word-context pairs is the product of the k likelihoods:

$$
\begin{aligned}
s(w; c_i) &= \boldsymbol{w} \cdot \boldsymbol{c}_i \\
P(w; c_1, ..c_k) &= \prod_{i=1}^{k} P(w; c_i)
\end{aligned}
\tag{B.3}
$$

As we can guess from the two definitions, CBOW computes the conditional probability of a target word given the context words surrounding it. On the other hand, the skip-gram does the exact opposite, by predicting the probability of context words given the central target word. The two different architectures are shown in Figure B.2.

According to the authors, the CBOW variant is much faster while the Skip-Gram one does a better job for infrequent words.

Optimization

A Word2Vec model can be trained with hierarchical softmax and/or negative sampling. To approximate the conditional log-likelihood a model seeks to maximize, the hierarchical softmax method uses a Huffman tree (Huffman, 1952) to reduce calculation. The negative sampling method, on the other hand, approaches the maximization problem by minimizing the log-likelihood of sampled negative instances. According to the authors, hierarchical softmax works better for infrequent words while negative sampling works better for frequent words and better with low dimensional vectors.

Ringraziamenti

Ci sono tante persone che hanno fatto parte della mia vita. Ho condiviso con loro tanti momenti, tante storie che meriterebbero tutte di essere raccontate.

Voglio ringraziare innanzitutto la mia famiglia. I miei genitori, Marisa e Giovanni, meritano di stare in cima a questa lista. Mi hanno cresciuto, mi hanno trasmesso dei valori sani e mi hanno insegnato, come direbbe mia mamma, "a vivere al mondo". Li voglio ringraziare per aver sempre creduto in me, anche nei miei momenti più difficili e per avermi dato dei buoni consigli quando ne avevo bisogno. Non dimentico mia sorella, Elena, che ringrazio per tutti i bei momenti passati. Ti ho vista crescere e sarò sempre orgoglioso di te.

Un ringraziamento va poi a tutti i miei parenti. Sento sempre la loro vicinanza, anche vedendoli solo una volta ogni tanto. Un pensiero speciale va a mia nonna Lucia. Avrei voluto tanto condividere con te le gioie della mia vita, spero che da lassù riesci a vedermi comunque.

Ringrazio tutti coloro che in passato sono stati compagni di avventure. In tutto il mio percorso scolastico ho avuto modo di conoscere tante persone e tutti, a loro modo, mi hanno trasmesso qualcosa. La stessa cosa vale per i miei insegnanti da cui ho imparato, oltre alle conoscenze scolastiche, anche e soprattutto il rispetto per gli adulti.

Lo sport ha avuto in passato un ruolo fondamentale nella mia vita. Ringrazio tutte le persone con cui ho avuto a che fare nel mondo del basket, prima come giocatore e poi come istruttore. Allenarsi duramente per prepararsi al meglio, ma anche saper accettare i risultati che si ottengono sono due cose che insegna lo sport ma che servono anche nella vita.

La parrocchia è uno degli ambienti che mi piace frequentare. Ringrazio chi mi ha dato l'opportunità di formarmi come cristiano e come persona e poi di diventare animatore per tanti ragazzi. Grazie alla parrocchia ho conosciuto alcune persone che frequento ancora e con cui ho stretto un legame forte di amicizia.

Ringrazio infine tutti i miei amici, quelle persone con cui uscire per divertirsi, ma anche condividere esperienze di vita. Un amico è una persona con

cui ti senti a tuo agio, non importa che ci si veda tutti i giorni o una volta ogni tanto. Il primo che mi viene in mente è Alessandro, che ho conosciuto per caso ai tempi delle superiori e con cui ho piano piano scoperto di avere tante cose in comune. Ringrazio Cristian, Simone, Cristian e Nicola: amici dai tempi delle medie con cui ho passato dei bei momenti e spero ce ne saranno tanti altri. A questi andrebbe aggiunto il buon Matteo, che ho conosciuto il primo giorno di scuola media e che è ancora al mio fianco dopo tanti anni. Ringrazio le persone che fanno parte della mia attuale compagnia di amici: lo stesso Matteo, Alessandro, Riccardo, Fabio, Mirko, Andrea, Aurora, Alice, Benedetta, Elena, Francesco, Giorgia, Marta, Mattia, Michelle. Il legame che mi unisce a voi non mi viene facile da spiegare adesso.

Sono sicuro che non ho nominato qualcuno, e un po' mi dispiace. Tutte le persone che hanno dimostrato di volermi bene sono persone speciali e a cui sono molto grato.

Bibliography

[Aggarwal and Zhai 2012] Aggarwal, Charu C. ; Zhai, Cheng X.: *Mining Text Data*. Springer Publishing Company, Incorporated, 2012. – ISBN 1461432227

[Bengio et al. 2003] Bengio, Yoshua ; Ducharme, Réjean ; Vincent, Pascal ; Janvin, Christian: A Neural Probabilistic Language Model. In: *J. Mach. Learn. Res.* 3 (2003), M^barz, Nr. null, S. 1137–1155. – ISSN 1532-4435

[Bird et al. 2009] Bird, Steven ; Klein, Ewan ; Loper, Edward: *Natural Language Processing with Python*. 1st. O'Reilly Media, Inc., 2009. – ISBN 0596516495

[Bottou 1998] Bottou, Léon: On-line learning and stochastic approximations. In: *In On-line Learning in Neural Networks*, Cambridge University Press, 1998, S. 9–42

[Cho et al. 2014] Cho, Kyunghyun ; Merriënboer, Bart van ; Gulcehre, Caglar ; Bahdanau, Dzmitry ; Bougares, Fethi ; Schwenk, Holger ; Bengio, Yoshua: Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar : Association for Computational Linguistics, Oktober 2014, S. 1724–1734. – URL `https://www.aclweb.org/anthology/D14-1179`

[Cleverdon 1984] Cleverdon, Cyril: Optimizing Convenient Online Access to Bibliographic Databases. In: *Inf. Serv. Use* 4 (1984), Juni, Nr. 1–2, S. 37–47. – ISSN 0167-5265

[Collobert and Weston 2008] Collobert, Ronan ; Weston, Jason: A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In: *Proceedings of the 25th International*

Conference on Machine Learning. New York, NY, USA : ACM, 2008 (ICML '08), S. 160–167. – URL http://doi.acm.org/10.1145/1390156.1390177. – ISBN 978-1-60558-205-4

[Devlin et al. 2019] Devlin, Jacob ; Chang, Ming-Wei ; Lee, Kenton ; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota : Association for Computational Linguistics, Juni 2019, S. 4171–4186. – URL https://www.aclweb.org/anthology/N19-1423

[El Hihi and Bengio 1995] El Hihi, Salah ; Bengio, Yoshua: Hierarchical Recurrent Neural Networks for Long-Term Dependencies. In: *Proceedings of the 8th International Conference on Neural Information Processing Systems*. Cambridge, MA, USA : MIT Press, 1995 (NIPS'95), S. 493–499

[Elman 1990] Elman, Jeffrey L.: Finding structure in time. In: *COGNITIVE SCIENCE* 14 (1990), Nr. 2, S. 179–211

[Goldberg 2017] Goldberg, Yoav: *Neural Network Methods for Natural Language Processing*. Morgan and Claypool Publishers, 2017 (Synthesis Lectures on Human Language Technologies)

[Goodfellow et al. 2016] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. The MIT Press, 2016. – ISBN 0262035618

[Harris 1954] Harris, Zellig: Distributional structure. In: *Word* 10 (1954), Nr. 23, S. 146–162

[Hinton et al. 1986] Hinton, Geoffrey E. ; Mcclelland, James L. ; Rumelhart, David E.: Distributed Representations. In: Rumelhart, David E. (Hrsg.) ; Mcclelland, James L. (Hrsg.): *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Cambridge, MA : MIT Press, 1986, S. 77–109

[Hirschberg and Manning 2015] Hirschberg, Julia ; Manning, Christopher D.: Advances in natural language processing. In: *Science* 349 (2015), Nr. 6245, S. 261–266

[Hochreiter and Schmidhuber 1997] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-Term Memory. In: *Neural Comput.* 9 (1997), November, Nr. 8, S. 1735–1780. – URL https://doi.org/10.1162/neco.1997.9.8.1735. – ISSN 0899-7667

[Huffman 1952] Huffman, David A.: A Method for the Construction of Minimum-Redundancy Codes. In: *Proceedings of the Institute of Radio Engineers* 40 (1952), September, Nr. 9, S. 1098–1101

[Jurafsky and Martin 2009] Jurafsky, Dan ; Martin, James H.: *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition.* Upper Saddle River, N.J. : Pearson Prentice Hall, 2009. – URL http://www.amazon.com/Speech-Language-Processing-2nd-Edition/dp/0131873210/ref=pd_bxgy_b_img_y. – ISBN 9780131873216 0131873210

[Kadhim 2018] Kadhim, Ammar: An Evaluation of Preprocessing Techniques for Text Classification. In: *International Journal of Computer Science and Information Security,* 16 (2018), 06, S. 22–32

[Kim et al. 2016] Kim, Yoon ; Jernite, Yacine ; Sontag, David ; Rush, Alexander M.: Character-Aware Neural Language Models. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI Press, 2016 (AAAI'16), S. 2741–2749

[Kingma and Ba 2014] Kingma, Diederik P. ; Ba, Jimmy: *Adam: A Method for Stochastic Optimization.* 2014. – URL http://arxiv.org/abs/1412.6980. – cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015

[Manning et al. 2008] Manning, Christopher D. ; Raghavan, Prabhakar ; Schütze, Hinrich: *Introduction to Information Retrieval.* USA : Cambridge University Press, 2008. – ISBN 0521865719

[Mikolov et al. 2013a] Mikolov, Tomas ; Chen, Kai ; Corrado, Greg ; Dean, Jeffrey: Efficient Estimation of Word Representations in Vector Space. In: Bengio, Yoshua (Hrsg.) ; LeCun, Yann (Hrsg.): *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, URL http://arxiv.org/abs/1301.3781, 2013

[Mikolov et al. 2013b] Mikolov, Tomas ; Sutskever, Ilya ; Chen, Kai ; Corrado, Greg S. ; Dean, Jeff: Distributed Representations

of Words and Phrases and their Compositionality. In: Burges, C. J. C. (Hrsg.) ; Bottou, L. (Hrsg.) ; Welling, M. (Hrsg.) ; Ghahramani, Z. (Hrsg.) ; Weinberger, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 26, Curran Associates, Inc., 2013, S. 3111–3119. – URL https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[Nash 1984] Nash, Stephen G.: Newton-Type Minimization Via the Lanczos Method. In: *SIAM Journal on Numerical Analysis* 21 (1984), Nr. 4, S. 770–788. – URL http://www.jstor.org/stable/2157008. – ISSN 00361429

[Pascanu et al. 2013] Pascanu, Razvan ; Mikolov, Tomas ; Bengio, Yoshua: On the difficulty of training recurrent neural networks. Atlanta, Georgia, USA : PMLR, 17–19 Jun 2013, S. 1310–1318. – URL http://proceedings.mlr.press/v28/pascanu13.html

[Pennington et al. 2014] Pennington, Jeffrey ; Socher, Richard ; Manning, Christopher D.: Glove: Global Vectors for Word Representation. In: *EMNLP* Bd. 14, 2014, S. 1532–1543

[Peters et al. 2018] Peters, Matthew ; Neumann, Mark ; Iyyer, Mohit ; Gardner, Matt ; Clark, Christopher ; Lee, Kenton ; Zettlemoyer, Luke: Deep Contextualized Word Representations. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana : Association for Computational Linguistics, Juni 2018, S. 2227–2237. – URL https://www.aclweb.org/anthology/N18-1202

[Prechelt 1996] Prechelt, Lutz: Early Stopping-But When? In: Orr, Genevieve B. (Hrsg.) ; Müller, Klaus-Robert (Hrsg.): *Neural Networks: Tricks of the Trade* Bd. 1524. Springer, 1996, S. 55–69. – URL http://dblp.uni-trier.de/db/conf/nips/nips1996.html#Prechelt96. – ISBN 3-540-65311-2

[Radford 2018] Radford, A.: Improving Language Understanding by Generative Pre-Training, 2018

[Rumelhart et al. 1986] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning Representations by Back-propagating Errors. In: *Nature* 323 (1986), Nr. 6088, S. 533–536. – URL http://www.nature.com/articles/323533a0

[Schuster and Paliwal 1997] Schuster, M. ; Paliwal, K.K.: Bidirectional Recurrent Neural Networks. In: *Trans. Sig. Proc.* 45 (1997), November, Nr. 11, S. 2673–2681. – URL https://doi.org/10.1109/78.650093. – ISSN 1053-587X

[Schuster et al. 2019] Schuster, Tal ; Ram, Ori ; Barzilay, Regina ; Globerson, Amir: Cross-Lingual Alignment of Contextual Word Embeddings, with Applications to Zero-shot Dependency Parsing. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota : Association for Computational Linguistics, Juni 2019, S. 1599–1613. – URL https://www.aclweb.org/anthology/N19-1162

[Sebastiani 2002] Sebastiani, Fabrizio: Machine learning in automated text categorization. In: *ACM Comput. Surv.* 34 (2002), Nr. 1, S. 1–47. – URL http://portal.acm.org/citation.cfm?id=505283. – ISSN 0360-0300

[Srivastava et al. 2015] Srivastava, Rupesh K. ; Greff, Klaus ; Schmidhuber, Jürgen: Training Very Deep Networks. In: Cortes, C. (Hrsg.) ; Lawrence, N. (Hrsg.) ; Lee, D. (Hrsg.) ; Sugiyama, M. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 28, Curran Associates, Inc., 2015, S. 2377–2385. – URL https://proceedings.neurips.cc/paper/2015/file/215a71a12769b056c3c32e7299f1c5ed-Paper.pdf

[Srividhya and Anitha 2010] Srividhya, V ; Anitha, R: Evaluating preprocessing techniques in text categorization. In: *International journal of computer science and application* 47 (2010), Nr. 11, S. 49–51

[Vaswani et al. 2017] Vaswani, Ashish ; Shazeer, Noam ; Parmar, Niki ; Uszkoreit, Jakob ; Jones, Llion ; Gomez, Aidan N. ; Kaiser, Ł u. ; Polosukhin, Illia: Attention is All you Need. In: Guyon, I. (Hrsg.) ; Luxburg, U. V. (Hrsg.) ; Bengio, S. (Hrsg.) ; Wallach, H. (Hrsg.) ; Fergus, R. (Hrsg.) ; Vishwanathan, S. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 30, Curran Associates, Inc., 2017, S. 5998–6008. – URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

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

[Zhang and Wallace 2015] Zhang, Ye ; Wallace, Byron: *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*. 2015