UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Spectral Graph Convolutional Networks for Part-of-Speech Tagging

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Saner Demirel

Erstgutachter:	Prof. Dr. Steffen Staab (Institute for Web Science and Technologies)
Zweitgutachter:	Lukas Schmelzeisen, B.Sc. (Institute for Web Science and Technologies)

Koblenz, im Oktober 2017

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

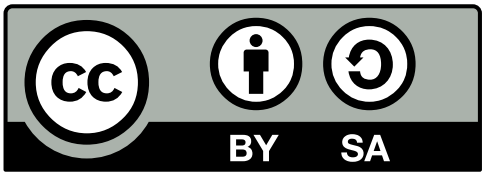
(Ort, Datum) (Unterschrift)

Abstract

Part-of-Speech tagging is the process of assigning words with similar grammatical properties to a part of speech (PoS). In the English language, PoS-tagging algorithms generally reach very high accuracy. This thesis undertakes the task to test against these accuracies in PoS-tagging as a qualitative measure in classification capabilities for a recently developed neural network model, called graph convolutional network (GCN). The novelty proposed in this thesis is to translate a corpus into a graph as a direct input for the GCN. The experiments in this thesis serve as a proof of concept with room for improvements.

Zusammenfassung

Part-of-Speech Tagging bezeichnet die Zuordnung von Wörtern, die ähnliche grammatikalische Eigenschaften aufweisen, zu Wortarten (Abk. PoS). Im englischen erreichen PoS-Tagger im Allgemeinen sehr hohe Genauigkeiten. Diese Arbeit nimmt sich zur Aufgabe die Klassifizierungsfähigkeiten eines neuen Neuralen Netzwerks, das Graph Convolutional Network, qualitativ anhand dieser Genauigkeiten in PoS-Tagging zu messen und zu vergleichen. Der neuartige Ansatz dieser Arbeit ist hierbei die Umwandlung eines Textcorpus in einen Graphen als direkte Dateneingabe für das GCN. Die Experimente dieser Arbeit dienen als Machbarkeitsnachweis mit Raum für Verbesserungen.



Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
GCN	Graph Convolutional Network
GDA	Gradient Descent Algorithm
HMM	Hidden Markov Model
LMS	Least Mean Squares
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
NN	*see ANN*
PLA	Perceptron Learning Algorithm
PoS	Part-of-Speech
ReLU	Rectified Linear Unit
SGC	Spectral Graph Convolution
TnT	Trigrams'n'Tags

1 Introduction to Part-of-Speech Tagging

Part-of-Speech (PoS) tagging, in various literature also found as *word-category disambiguation* or *grammatical tagging*, is the process of tagging a PoS, also called *word class* or *linguistic unit*, by assigning it to a PoS-marker in a text or in a large structured set of texts (*corpus*).

A PoS is a category of words with similar grammatical properties. Because PoS-tags are not limited to words but can also include, for example, punctuation, part-of-word punctuation (such as e.g., i.e. and etc.), numbers, alpha-numerics, compound words (e.g. "San Francisco"), some way of segmentation of the text is needed. As the listed examples suggest — even though commonly applied — segmentation through whitespace might not always be the first choice. In general, a text is segmented semantically, morphologically or syntactically. The process of segmentation is called *tokenization*, which is a preprocessing state or a part of the tagging process. Basically, tokenization is performed as preferred or performed dependent on the chosen tagset.

From the perspective of *Natural Language Processing (NLP)*, the tagging of the particular PoS requires knowledge of the definition and context, i.e. its relationship with adjacent and related words in a phrase, sentence, or paragraph. Tagging can therefore be viewed as a *disambiguation task*: PoS are ambiguous, meaning there are more than one possible PoS-tags, and the goal is to resolve these ambiguities by choosing the proper tag in the context. For example, the word "dogs", which is usually thought of as a plural noun, is instead a verb in the sentence "The sailor dogs the hatch.".

Nowadays PoS-tagging is a part or a preprocessing step of various NLP tasks because of the large amount of information they give about a word and its neighbors (Jurafsky and Martin 2009). Knowledge about what kind of PoS a word is helps in the prediction of neighboring words and about the syntactic structure around the word, which makes PoS-tagging an important component of syntactic parsing, such as in the works of Chen and Manning (2014). Jurafsky and Martin (2009) further describe, that PoS is *"a useful feature for finding named entities like people or organizations in text and other information extraction tasks"* and influences possible morphological affixes and therefore also affects stemming for informational retrieval. Another use is to improve pronunciation in speech synthesis and recognition.

In the English language, eight PoS are taught, which can be divided into two word metaclasses (Francis and Kucera 1979; Jurafsky and Martin 2009), as follows:

content words: noun, verb, adjective, adverb[1]; the open lexical classes

function words: article, preposition, conjunction, pronoun; the closed lex-

[1]Interjections as a 9^{th} PoS do not occur within this thesis and are therefore negligible.

ical classes[2]

Open classes are continually being created or borrowed, while *closed classes* only occasionally occur (Jurafsky and Martin 2009). However, as most established tagsets suggest, there are many more (sub-)classes. This thesis will focus on the well known *Brown* corpus with a tagset of 82 PoS-tags (Francis and Kucera 1979).

Tag ambiguity is responsible for the difficulty of the tagging problem. Though not tagged with the original Brown tagset but with a similarly well established tagset called *Treebank-3*, which has 45 PoS-tags, table 1.1 still depicts the problem of tag ambiguity: Most vocabulary (85%) in the Brown

	Unambiguous (1 tag)	Ambiguous (2+ tags)
Types	45,799 (85%)	8,050 (15%)
Tokens	384,349 (33%)	786,646 (67%)

Table 1.1: Amount of tag ambiguity for word types in the Brown corpus with Treebank-3 tagging

corpus is unambiguous, however, the rest of the words are some of the most common in the English language, thus 67% of word tokens are ambiguous (Jurafsky and Martin 2009).

Usually, the input to a tagging algorithm is a corpus and a tagset, and the output is a sequence of tagged linguistic units (Jurafsky and Martin 2009). This thesis takes a highly new approach on PoS-tagging with a *Neural Network (NN)* by converting the corpus first into a graph as an input for the newly developed *Graph Convolutional Network (GCN)* machine learning algorithm by Kipf and Welling (2016). GCNs are one of the very first in the category of NNs to take graphs as input without prior conversion into another input form and are a very promising concept in the field of machine learning. State-of-the-art algorithms, such as the *Dynamic Feature Induction* algorithm by Choi (2016) used in NLP4J[3], reach around 97% accuracy in PoS-tagging, hence the goal of this thesis is not to improve the accuracy of PoS-tagging but to give a qualitative measure of the capabilities of GCNs in terms of classification.

The remainder of this chapter gives an in-depth description of the Brown corpus design (section 1.1) as a comprehension base for subsequent chapters and further motivates the idea of using a GCN as a PoS-tagger (section 1.2).

[2]Some literature make a distinction between open-closed classes and content-function words, which makes sense for other than the English language. This thesis does not make a distinction.

[3]`https://emorynlp.github.io/nlp4j/`

1.1 The Brown Corpus and Tagset

As described by Francis and Kucera (1979), the Brown corpus consists of 1,014,312 words of running text of edited varieties of English prose. It is divided into 500 samples. A sample begins at the beginning of a sentence regardless of being the beginning of a paragraph or other larger division, and ends after roughly 2000 words. A sentence is a string of words beginning with a capital and ending with a final mark (., ! or ?; abbreviations excluded) followed by space. In some cases, the final mark of a sentence not followed by space, e.g. when quotation marks are used. The tagged version of the corpus provides each individual word a brief PoS-tag which assigns it to a specific word class. Following example extracted from the *Natural Language Toolkit (NLTK)*[4] demonstrates the structure of the tagged corpus, which will become important in chapter 4:

```
The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl
said/vbd Friday/nr an/at investigation/nn of/in Atlanta's/np$
recent/jj primary/nn election/nn produced/vbd ``/`` no/at
evidence/nn ''/'' that/cs any/dti irregularities/nns took/vbd
place/nn ./.
```

Figure 1.1: An example of a tagged sentence taken from the Brown corpus

A complete explanation of every tag is of secondary importance for this thesis. Readers are referred to the complete list of tags and their explanations found in the Brown Corpus Manual by Francis and Kucera (1979).

1.2 PoS with Neural Networks on Graphs

With increasingly larger real-world datasets, the common trend goes towards forms of graphs: social networks, knowledge graphs, protein-interaction networks, the World Wide Web, to name a few. However, despite the rapidly advancing research of NNs, little attention was dedicated to the generalization of NN models to work on arbitrarily structured graphs, until recently, when a number of researchers started to revisit this problem (Defferrard et al. (2016), Duvenaud et al. (2015), Kipf and Welling (2016), and Parisot et al. (2017) among others) with promising results.

The focus on the solution to this problem in this thesis is on approaches that use *graph convolutions*, generally known from spectral graph theory. Defferrard et al. (2016) use Chebyshev polynomials with free parameters to approximate smooth filters in the spectral domain learned in a NN-like model. The research of Kipf and Welling (2016), on which the main focus remains, takes a similar approach with simplifications (see section 3.3), al-

[4] `http://www.nltk.org/`

lowing for higher predictive accuracy, reaching state-of-the-art *classification* results on a number of benchmark graph datasets, and faster training times.

It stands to reason that these results are tested against tasks that can be considered as fairly solved[5] in their domain, as is the case with PoS-tagging in the English language. The question arises how a text corpus can be imagined as a graph to act as an input for a GCN. To anticipate the answer, we adduce the *Convolutional Neural Network (CNN)* before its proper introduction in section 3.3.1: GCNs originate from CNNs, which proved themselves as powerful models, primarily on images, that exploit features (e.g. image intensities) and neighborhood information (e.g. pixel grid) to solve problems like image segmentation and classification (Parisot et al. 2017). The task of PoS-tagging is, as implied before, a classification problem and can be compared to image segmentation, and the per-pixel classification afterwards. This context leads to the analogy between an image pixel plus intensity and a PoS with corresponding *feature vectors*, while the generated text graph, depicting neighborhood structure, equates to the pixel grid.

1.3 Thesis Overview

This thesis is structured as follows:

Chapter 2 reviews several general approaches in state-of-the-art PoS-tagging.

In chapter 3, a huge part of this thesis is devoted to understand the basics of machine learning with NNs. This is important for the comprehension of GCNs, which is also covered in the same chapter. This chapter also introduces the used notation.

Chapter 4 covers the core contribution of this thesis. This chapter describes how the PoS-Graph is constructed, how it is fed as an input to the GCN and which hyperparameters we optimize.

The evaluation of the implementation is found in chapter 5, where its performance in practice is analyzed.

Finally, chapter 6 concludes the thesis and discusses future work.

[5] Giesbrecht and Evert (2009) point out that this statement is very arguable, since most PoS-taggers only reach high accuracies in artificial conditions. Also, a per-word accuracy of 97%, for example, still means that there is a probability of around 50% to find one or more tagging errors in a 20-word sentence, which is the length of the average Brown corpus sentence.

2 Related Work

Many PoS-tagging algorithms participate in a head-to-head race on which algorithm can be considered the ”best”. The answer to this question is not straightforward, as it highly depends on the taken measurements. Therefore, this chapter provides a general brief overview over four types of PoS-tagging approaches and gives examples of well evaluated state-of-the-art algorithms for English corpora. Motivation or justification of their origin will not be given, since they are highly differentiable to the approach taken in this thesis and have only secondary significance to the contribution of this thesis. Instead, the interested reader is hereby referred to the primary sources. Since GCNs emerged very recently and research is still actively ongoing while this thesis is written, there are no scientific works that actively do PoS-tagging with GCNs — at least not to the author's knowledge.

The used notation in this chapter is for this chapter only. It is dropped as soon as the chapter ends.

2.1 Types of PoS Taggers

Generally, the types of approaches in PoS-tagging can be divided into two categories: rule and stochastic based, where the former tags based on rules and the later based on probability models (Tian and Lo 2015). We proceed to present these types in the following. For a comparative analysis of the types, we refer to Giesbrecht and Evert (2009) and Tian and Lo (2015).

2.1.1 Unigram Tagger

A *unigram* PoS tagging algorithm is rule based and assigns a marker to a word that is most likely based on the training corpus. It computed and stored the likelihood of a tag for each word before and assigns, for instance, the word ”dogs” to the tag *noun* if the tag has the highest count for the specific word in the training examples. Unknown words are automatically tagged as *noun*, i.e. the context of the word is ignored. This approach is simple and fast and achieves acceptable accuracy when the training corpus is large enough (Tian and Lo 2015).

2.1.2 Hidden Markov Model Based Tagger

A *Hidden Markov Model (HMM)* based tagger also marks a word with a tag with the highest likelihood. However, a HMM tagger computes a tag sequence for a sentence as a whole instead of having a tag for each word separately. A HMM tagger chooses a tag sequence $\mathbf{t} = t_1 \dots t_n$ that maximizes the joint probability

$$P(\mathbf{t}, \mathbf{w}) \quad = \quad P(\mathbf{t})P(\mathbf{w}|\mathbf{t}), \tag{2.1}$$

with $\mathbf{w} = w_1 \dots w_n$ as a sequence of words. The computation of $P(\mathbf{t})$ proved itself as impractical, which is why it is generally simplified (Tian and Lo 2015). *Trigrams'n'Tags (TnT)* by Brants (2000) proposes the assumption that the tag of a word is determined by the tags of the previous two words, by using second order Markov models. The *tree tagger* by Schmid (1995) leverages decision trees to get more reliable estimates in Markov models.

2.1.3 Maximum Entropy Based Tagger

Maximum entropy based tagger provide a principled way to incorporate complex features to maximize the entropy of a probability model, originally proposed by (Ratnaparkhi et al. 1996):

$$P(\mathbf{t}|\mathbf{w}) \approx \prod_{i=1}^{N} P(t_i|C_i), \tag{2.2}$$

where $C_1 \dots C_n$ are the corresponding contexts for each word w appearing in the sentence, including previous assigned tags before the word w (Tian and Lo 2015). *Feature-functions* are a real-valued concept in maximum entropy, which encode elements of C useful for predicting the tag of a word by representing constraints (Tian and Lo 2015). The *Stanford Tagger* (Toutanova et al. 2003) improves the original algorithm by considering two more types of feature-functions affected by stylization of characters (e.g., whether the first letter of a word is capitalized or not).

2.1.4 Transformation Based Tagger

Transformation based tagger (Brill 1995) tag words based on linguistic knowledge in form of rules. A stochastic-based tagger is used first to get an initially tagged corpus to correct the errors afterwards. The rules needed to correct are then learned automatically from a training corpus (Tian and Lo 2015).

3 Machine Learning and Artificial Neural Networks

An *Artificial Neural Network (ANN)*, is a connectionist computational system, which is composed of a collection of simple processing units, or *neurons*, connected to each other, loosely modeled after the biological brain. Haykin (1998) defines NNs in resemblance of the brain in two respects: first is the ability to acquire knowledge from real-world data — be it images, sound, text or time series — through a *learning process*, second is the ability to store the acquired knowledge by strengthening or *weighting* the interneuron connections, or *synapses*. We split the first two sections of this chapter accordingly: the first section formalizes the learning problem and discusses different paradigms of learning that have arisen to deal with different situations and different assumptions. In the second section, we further describe the inner workings of a NN. The remaining section assumes comprehension of NNs and is dedicated to the formalization of GCNs.

3.1 The Learning Problem

Learning from data is applied whenever it is difficult to express an analytical solution to a problem through a traditional procedural system, but where it is possible to provide an empirical solution. Thus, the basic premise of learning from data is to uncover an underlying process through observation. To keep structural integrity, we introduce a common terminology that is retained throughout the thesis. Whenever several terms to an idea are introduced, we use them synonymously, detached from the fact that other (machine learning) publications may assign special meaning to some of these terms. The following notation for sections 3.1 and 3.2 mirrors that of Abu-Mostafa et al. (2012). Other publications may differ in used designation and notation.

3.1.1 Components of Learning

A learning problem consists of the following main components: an input example or *feature vector* $\mathbf{x}$, where each entry $x^{(i)}$ is called a *feature*, an output example $\mathbf{y}$, which may be also called *label* or *target* vector, with entries $y^{(i)}$, an unknown target function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the output space, a (*labeled*) data set $\mathcal{D}$ of input-output or *training examples* $(\mathbf{x}_1, \mathbf{y}_1), \cdots, (\mathbf{x}_m, \mathbf{y}_m)$, where $\mathbf{y}_j = f(\mathbf{x}_j)$ for $j = 1, \ldots, m$, and the learning algorithm that uses the data set $\mathcal{D}$ to pick a formula $g : \mathcal{X} \to \mathcal{Y}$ that approximates f. Note that the superscript "(i)" in the notation is simply an index into the entries of the vector, and has nothing to do with exponentiation. A second superscript sitting on the right to the first one denotes the index of the vector: $x^{(ij)}$ is the ith entry to the vector $\mathbf{x}_j$. To generalize, the input and the output example are stated as vectors, for easier understanding though, the output example can be thought of as a vector

with one entry or as a simple scalar. $\mathcal{H}$ is called the hypothesis set, a set of candidate formulas $h \in \mathcal{H}$ the *learning algorithm* $\mathcal{A}$ chooses g from. The choice of g is good when it is able to faithfully replicate f. This is done by choosing a g that is the best match for f based on the given training examples, in hope that it continues to match f on new input data, which is also called *generalization*. The data set $\mathcal{D}$ and the unknown target function f are dictated by the learning problem itself, the *learning model*, which is the combination of the learning algorithm $\mathcal{A}$ and the hypothesis set $\mathcal{H}$, however, is not, but is free to be chosen (Abu-Mostafa et al. 2012).

3.1.2 Types of Learning

The most studied and utilized learning paradigm is *supervised learning*, which we have covered so far. Other variations may mostly differ in the nature of the provided data set. In this subsection, we introduce the most important variations.

Supervised learning algorithms experience a training data set with explicit examples of what the correct output should be for given inputs (Abu-Mostafa et al. 2012). In other words, supervised learning involves observing several randomly selected feature vectors $\mathbf{x}$ and the associated or *annotated* label $\mathbf{y}$, and learning to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating the unknown joint distribution $P(\mathbf{x}, \mathbf{y})$ (Goodfellow et al. 2016; Subramanya and Talukdar 2014). This is usually known as a *regression* task. Similar to regression is the task of classification, introduced in chapter 1, where an algorithm is asked to specify which of the k categories an input belongs to (instead of predicting a numerical value). For this, the algorithm is prompted to produce the function $f : \mathbb{R}^d \to \{1, \ldots, k\}$ (Goodfellow et al. 2016). Note that even though the k-classes are denoted as scalars, they could also act as vectors.

In *unsupervised learning*, the training data is unlabeled and only contains information about the features. It can be viewed as the task of finding useful properties of the structure and patterns of a given dataset. Roughly speaking and in contrast to supervised learning, unsupervised learning involves observing several randomly selected feature vectors $\mathbf{x}$, and attempting to implicitly or explicitly learn the unknown probability distribution $P(\mathbf{x})$, or finding interesting properties of that distribution (Goodfellow et al. 2016; Subramanya and Talukdar 2014). Another role of unsupervised learning is *clustering*, which divides datasets into clusters of similar examples. However, the correct clustering, in comparison to the same but supervised task, becomes less obvious, and even the number of clusters may be ambiguous (Abu-Mostafa et al. 2012). Unsupervised learning can be considered as a standalone technique or as a way to create higher-level representation of data to become a precursor to supervised learning (Abu-Mostafa et al. 2012).

Semi-supervised learning combines both supervised and unsupervised learning. The training set is only partly labeled and largely unlabeled.

The goal is to approximate the target function f given a training set $\mathcal{D} = (\mathcal{D}_l, \mathcal{D}_u)$, where $\mathcal{D}_l = \{(\mathbf{x}_j, \mathbf{y}_j)\}$ for $j = 1, \ldots, m_l$ represents the labeled samples and $\mathcal{D}_u = \{\mathbf{x}_j\}$ for $j = 1, \ldots, m_u$ represents the unlabeled samples (Subramanya and Talukdar 2014). The assumed number of training samples is $N \equiv N_l + N_u$. As in supervised learning, each labeled training sample $(\mathbf{x}_j, \mathbf{y}_j)$ is predicted from the joint distribution $P(\mathbf{x}, \mathbf{y})$, and as in unsupervised learning, the unlabeled samples are predicted from the probability distribution $P(\mathbf{x})$, while both distributions are kept unrevealed to the learning algorithm (Subramanya and Talukdar 2014).

Reinforcement learning does not force the training data to contain the correct output for each input. It instead uses a feedback loop between the learning system and its experiences in the interaction with an environment. Again in comparison to supervised learning, where the examples were of the form `(input, correct output)`, reinforcement learning examples do not contain the target output, but instead include some possible output with a grading of how good this output is, in the form of `(input, some output, grade for this output)`, reinforcing the better actions and eventually learning what to do in similar situations (Abu-Mostafa et al. 2012).

3.2 Training a Neural Network

To be processed by the neurons of a network, all real-world data must be translated into numerical values beforehand, contained in vectors. Neurons are organized in several *layers* and each layer is a row of neurons. The output of a layer is the input of the subsequent layer, starting from an initial *input layer* that receives the data[6]. Hidden layers are intermediate layers between the input and the output layer, which usually enables the network for more complex computation and models. There is no general definition for a *Deep Neural Network (DNN)* but it is safe to assume that a neural network is considered *deep* if it at least consists of two hidden layers or more. NNs working in this manner, namely *forwarding* the signal from the input space to the output space through the several layers, are also summarized as *feedforward* (neural) networks. To further understand NNs, we first introduce the simplest type, the *perceptron*, and move to newer ones from there on.

3.2.1 Understanding the Basics: The Perceptron

A perceptron is usually portrayed with two layers, as seen in figure 3.1, with one computational unit in the second or — in this case — *output* layer. The perceptron takes several inputs from the input layer and produces a single binary output. The units are combined with coefficients, or *weights*, which

[6]Technically, the input layer contains neurons that only forward the input without any computation.

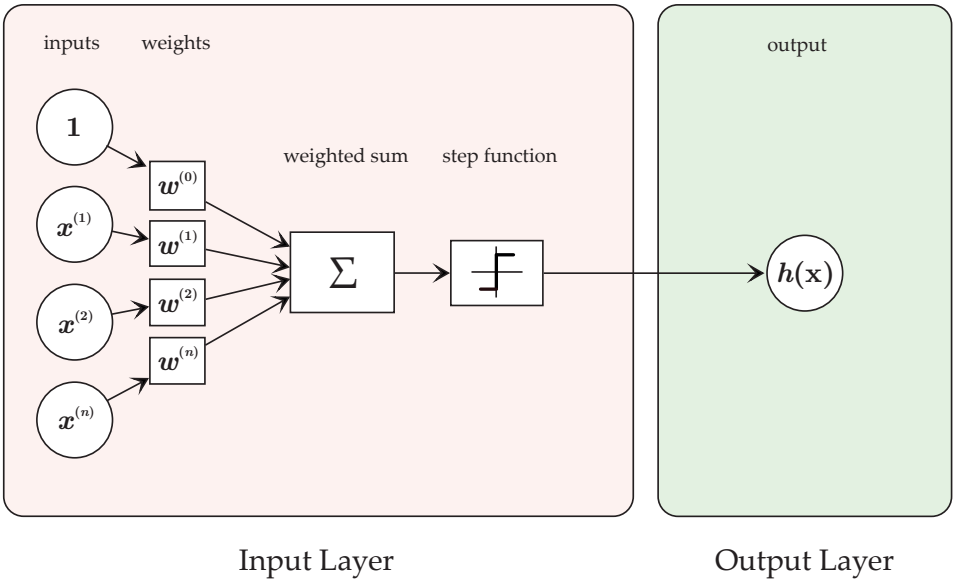


Figure 3.1: A diagram of a simplistic neural network: the perceptron

either amplify or dampen the input of the next layer according to their importance to the respective output (Nielsen 2015). The input-weight product is then passed through the perceptron's *activation function* — the *binary step*; a given threshold that determines whether the signal is propagated further, with 1 as the perceptron's output if the product is greater than the threshold, and 0 if it is not (Haykin 1998; Nielsen 2015). For clarification and anticipation, the difference between a perceptron and a neuron as a computational unit lies in the activation function. Usually and in contrast to a perceptron's, the neuron's activation function always propagates the signal, but to a certain extent, further explained in section 3.2.2.

Exemplary to *linear classification*, we introduce the perceptron learning model. We define the input space of a perceptron as $\mathcal{X} = \mathbb{R}^d$, where $\mathbb{R}^d$ is the d-dimensional Euclidian space, and the output space as $\mathcal{Y} = \{0, 1\}$. The functional output form $h(\mathbf{x})$, that all hypotheses $h \in \mathcal{H}$ share, gives different weights $w^{(i)}$ from the *weight vector* $\mathbf{w}$ to the different features $x^{(i)}$ of the feature vector $\mathbf{x}$. The weighted features are then summed and checked upon the threshold:

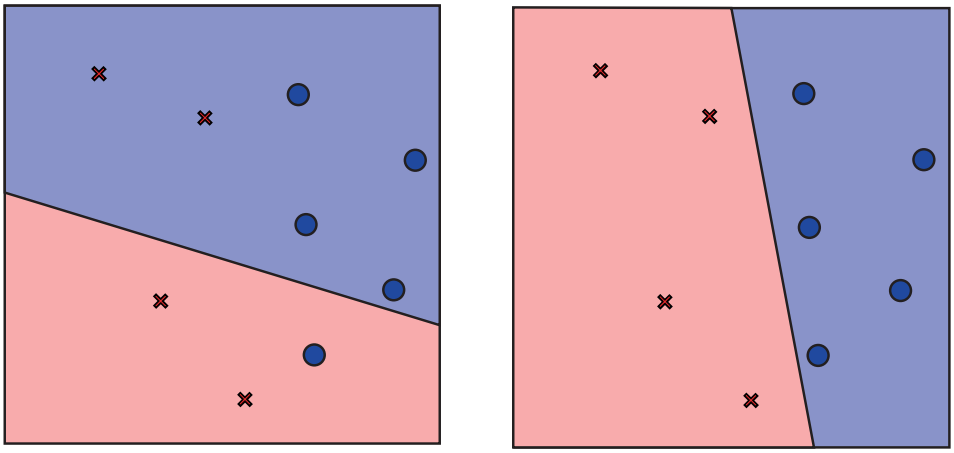
$$\text{output} \quad = \quad \begin{cases} 0 & \text{if } \sum_{i=1}^{d} w^{(i)} x^{(i)} \leq \text{ threshold} \\ 1 & \text{if } \sum_{i=1}^{d} w^{(i)} x^{(i)} > \text{ threshold} \end{cases}$$

More compactly, this formula can be written as

$$h(\mathbf{x}) \quad = \quad \text{sign}\left(\left(\sum_{i=1}^{d} w^{(i)} x^{(i)}\right) + b\right), \tag{3.1}$$

where function $\text{sign}(s) = 1$ if $s > 0$, and $\text{sign}(s) = 0$ if $s \leq 0$ formalizes the if-condition; *bias* $b \equiv -\text{threshold}$ to move the threshold to the left side of the equation. The bias can be thought of as a measure of how easy it is for the perceptron to "fire". A big positive bias makes the perceptron output a 1 easily; vice versa, a big negative bias makes it difficult (Nielsen 2015). Further simplifying the perceptron formula, we treat the bias b as a weight $w^{(0)} = b$, extending the weight vector to $\mathbf{w} = [w^{(0)}, w^{(1)}, \ldots, w^{(d)}]^\top$, where $^\top$ denotes the transpose of a vector. Accordingly, we also have to extend the feature vector to $\mathbf{x} = [x^{(0)}, x^{(1)}, \ldots, x^{(d)}]^\top$, while $x^{(0)} = 1$. Lastly, we write $\sum_{i=0}^{d} w^{(i)} x^{(i)}$ as $\mathbf{w}^\top \mathbf{x}$, which is the equivalent matrix product to the sum. With these changes, we can rewrite equation 3.1:

$$h(\mathbf{x}) \quad = \quad \text{sign}(\mathbf{w}^\top \mathbf{x}) \tag{3.2}$$



(a) Missclassified data (b) Correctly classified data

Figure 3.2: Classification of linearly separable data in a 2D input space The separating line is defined by the chosen weights and biases of the hypothesis. As seen in (a), training examples might be misclassified (blue points in red region and vice versa). The final hypothesis of the perceptron that separates the training data correctly is seen in (b), with $\circ$ being 1 and $\times$ being 0 *(figure analogous to Abu-Mostafa et al. (2012)).*

In regards to the learning model, the perceptron provides $\mathcal{H}$, outputting different hypotheses h. The *Perceptron Learning Algorithm (PLA)* searches for the optimal choice of a weight vector in $\mathcal{H}$ to produce the final hypothesis $g \in \mathcal{H}$. To explain how the choices are made, we assume that the input data is *linearly separable* in a two dimensional case, as illustrated in figure 3.2. Linear separability implies the existence of a $\mathbf{w}$, which converges to a correct $h(\mathbf{x}_j) = \mathbf{y}_j$ on all training examples (Abu-Mostafa et al. 2012; Mitchell 1997). The algorithm would fail to converge when the training examples

are not linearly separable, as there is no satisfying weight vector $\mathbf{w}$ that can separate the data (Mitchell 1997). The search for $\mathbf{w}$ works as described in algorithm 3.1. The binary separation is corrected by the update rule in the

Algorithm 3.1 The Perceptron Learning Algorithm

```
1:  procedure PLA(x, y, T)
2:      w ← 0                          ▷ weight vector initialized with zero vector
3:      for t = 0 to T do              ▷ T being the maximum count of iterations
4:          update ← false
5:          for j = 0 to m do          ▷ iterate through all training examples
6:              h_j ← sign(w^⊤ x_j)        ▷ apply perceptron equation
7:              if h_j ≠ y_j then          ▷ find misclassified example
8:                  w ← w + y_j x_j            ▷ update rule for weight
9:                  update ← true
10:         if update == false then
11:             break              ▷ stop when no misclassified examples found
12:     return w
```

direction of a correctly classified $\mathbf{x}_j$. The algorithm continues with further iterations until there are no longer misclassified examples in the data set or until the maximum count of iterations of the outer loop is reached (Abu-Mostafa et al. 2012).

3.2.2 From Perceptron to Neuron

Whenever we need to make small adjustments to the weights or biases to get the desired behavior, the output of the perceptron might completely flip, which can lead to aforementioned convergence issues (Nielsen 2015). To overcome this problem, neurons were introduced. Neurons can be seen as modified perceptrons, especially in regards to the aforementioned activation function; small changes in weights and biases cause small changes in the overall output of a neuron, which allows gradual improvements towards the final hypothesis (Nielsen 2015).

As does a perceptron, a neuron has several units $x^{(1)}, x^{(2)}, \ldots$ in the input layer, which forward floating-point numbers, weights $w^{(1)}, w^{(2)}, \ldots$ and a bias b. The output $h(\mathbf{x})$, however, is defined by $\sigma(\mathbf{w}^\top \mathbf{x})$ (cf. equation 3.2), where σ is the *sigmoid* or *logistic activation function* of a neuron. The *linear* binary step only produced 0 or 1 as an output, whereas the sigmoid function allows floating-point numbers bounded by 0 and 1, taking on a curvature that is basically a smoothed out version of the binary step (Nielsen 2015). The sigmoid function σ is defined as

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}, \tag{3.3}$$

resulting in

$$h_{\text{NEURON}}(\mathbf{x}) \quad = \quad \frac{1}{1+\exp(-(\mathbf{w}^\top \mathbf{x}))} \tag{3.4}$$

by inserting $(\mathbf{w}^\top \mathbf{x})$ as z. The output of the sigmoid function can be interpreted as a probability of an event occurring, reducing the problems of convergence on non-linear separable data.

Similarly to perceptrons, training a NN means that we try to find a hypothesis $h(\mathbf{x})$[7] that is close to the target $\mathbf{y}$ by learning the correct weights $\mathbf{w}$ on the given data set. To formalize this supervised approach, we define a *cost function* or *error function* $E(\mathbf{w})$ that measures for each weight how close the hypotheses $h(\mathbf{x}_j)$ are to their corresponding labels $\mathbf{y}_j$ (Mitchell 1997):

$$E(\mathbf{w}) \quad = \quad \frac{1}{2}\sum_{j=1}^{m}(h(\mathbf{x}_j)-\mathbf{y}_j)^2 \tag{3.5}$$

This familiar function is known as the *least-squares* cost function in *linear regression*, and gives rise to the *Least Mean Squares (LMS)* algorithm to minimize the cost function via *gradient descent*.

3.2.3 Optimization: Gradient Descent

As Mitchell (1997) describes, the *Gradient Descent Algorithm (GDA)* lets us quantify the quality of any particular weight vector $\mathbf{w}$. The goal of *optimization* is to minimize the error function $E(\mathbf{w})$ by starting with an arbitrary inital weight vector and repeatedly performing the update[8]

$$w^{(i)} \quad \leftarrow \quad w^{(i)} + \Delta w^{(i)} \tag{3.6}$$

where

$$\Delta w^{(i)} \quad = \quad -\eta \frac{\delta E(\mathbf{w})}{\delta w^{(i)}} \tag{3.7}$$

on it. Here, η is a constant called *learning rate*, which determines the step size in the GDA. To help understanding the GDA, the general idea with explanation is visualized in figure 3.3. In order to implement this algorithm, we need to efficiently calculate the gradient at each step by figuring out what the partial derivative $\frac{\delta E(\mathbf{w})}{\delta w^{(i)}}$ is. By differentiating $E(\mathbf{w})$ from equation 3.5

[7]For the sake of brevity, we drop the subscript tag as soon as one is introduced, such as NEURON in $h_{\text{NEURON}}(\mathbf{x})$. Unless otherwise stated, we use the latest introduced definition in the continuity of this thesis.

[8]The left arrow denotes an operation (in a computer program), in which we override the left side of the arrow with the right side.

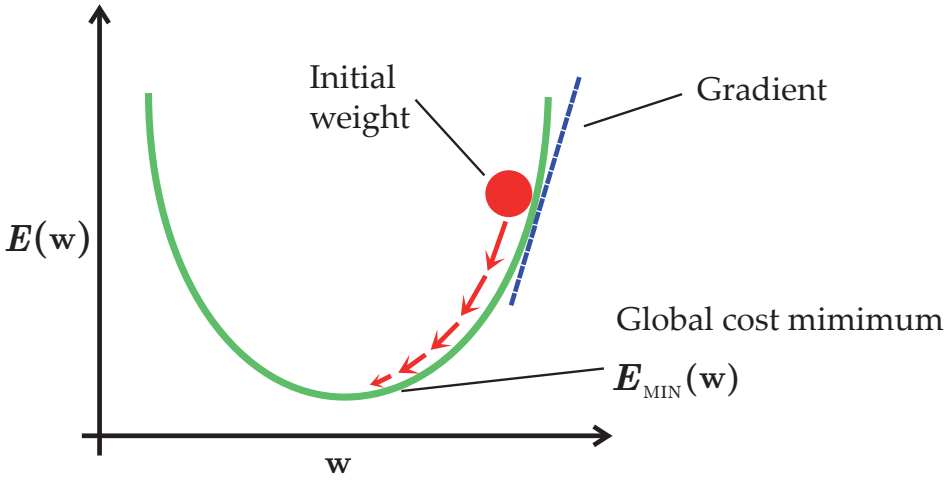


Figure 3.3: Visualization of the GDA on a convex graph
Error of different hypotheses in a two-dimensional example. The space of all hypotheses is the **w**-axis. The vertical axis indicates the error of the weight vector hypothesis, relative to a fixed set of training examples. The dotted line shows the gradient of the error graph. The error graph itself summarizes the desirability of every weight in the hypothesis space by showing the local minimum, or in this case even global minimum of the graph with a straight line, which corresponds to the desired hypothesis with minimum error. The big dot illustrates the initial chosen weights $w^{(i)}$. The arrows indicate the direction, in which the weights must be updated to produce the steepest descent along the graph.

in a few mathematical relations, formulated in Mitchell (1997), we have the following:

$$\frac{\delta E(\mathbf{w})}{\delta w^{(i)}} = \sum_{j=1}^{m} (h(\mathbf{x}_j) - \mathbf{y}_j)(-x^{(ij)}) \tag{3.8}$$

Substituting equation 3.8 into equation 3.7 yields the weight update rule for the GDA:

$$\Delta w^{(i)} = \eta \sum_{j=1}^{m} (h(\mathbf{x}_j) - \mathbf{y}_j)\, x^{(ij)} \tag{3.9}$$

Training with the GDA is summarized as follows:

1. Pick an initial random weight vector.
2. Compute the initial hypotheses for all training examples, then compute $\Delta w^{(i)}$ for each weight according to equation 3.9.

3. Update each weight $w^{(i)}$ by adding $\Delta w^{(i)}$, then repeat.

This variation of the GDA, which is also known as *batch* GDA, has two difficulties: first, converging to a local minimum can be slow; second, there is no guarantee that the global minimum will be found if there are multiple local minima (Mitchell 1997). Another variation of the GDA, the *stochastic* GDA, intends to mitigate these problems. Instead of updating weights after summing over all training examples, the stochastic GDA updates weights incrementally, followed by the calculation of the error for each individual example (Mitchell 1997).

3.3 Graph Convolutional Networks

The advancements by Kipf and Welling (2016) generalize a CNN (Krizhevsky et al. 2012) to learn on arbitrary graph-structured data in a semi-supervised fashion, first introduced in chapter 1 as a GCN. We take a small detour to CNNs, explaining them on a surface level first, to pave the way for a better understanding of GCNs.

3.3.1 Convolutional Neural Networks

CNNs are able to extract statistical patterns in high-dimensional datasets by revealing local features, which are shared across the data domain (Defferrard et al. 2016). To identify these features, Krizhevsky et al. (2012) use *filters* or *kernel*. known from the field of computer vision, to *convolve* regular grids, such as matrices. A filter can be imagined as a window function, i.e. focusing only on values inside the window at a time. For instance, we take an image as a representation of a matrix with color values for each pixel. The *stride* is the number of pixels by which we move our window over the input matrix, each time performing a mathematical operation with neighboring pixels inside the window. This convolution step outputs a *feature map*, usually smaller in dimension than the original image. The size of the feature map is predefined by three parameters:

- **Depth:** the depth of the feature map is the amount of different filters used on the input. Using three different filters on an image, for example, produces three stacked 2D-feature maps.
- **Stride:** as defined before.
- **Zero-padding:** to apply the filter on bordering elements of the input matrix, the outer border is "padded" with zeroes.

The convolution of a matrix is a linear process. To introduce non-linearity, an activation function is applied on each element (or each pixel in an image). The *Rectified Linear Unit (ReLU)* is defined as

$$\sigma_{\text{ReLU}}(z) \quad = \quad \max(0, z), \tag{3.10}$$

replacing all negative values in the feature map by zero (cf. equation 3.3).

Spatial Pooling (also called *subsampling* or *downsampling*) reduces the dimensionality of the rectified feature map while retaining the most important information. Similar to a filter, a window moves over the feature map and takes, for example, the largest element within the window. This is called *Max Pooling*, but there are also techniques such as averaging the values or to take the sum of all elements in the window.

At last, a *fully connected layer*, which is just another term in this context for a neural network, takes the high-level feature extractions of the convolution and pooling layers as an input. The sum of output probabilities from the connected layer is 1, ensured by using the *softmax* activation function in the output layer. The softmax equation, defined as

$$\sigma_{\text{Softmax}}(\mathbf{z}) \quad = \quad \frac{e^{z^{(i)}}}{\sum_i e^{z^{(i)}}}, \tag{3.11}$$

takes a vector $\mathbf{z}$ of arbitrary values and maps it to a vector of values between zero and one, as defined in the softmax equation 3.11. After training, the connected layer is able to classify the input matrix into various classes based on the training dataset (Krizhevsky et al. 2012).

3.3.2 Generalizing CNNs to Graphs

The generalization of CNNs to graphs is not straightforward as convolution and pooling operators only work on regular grids (Defferrard et al. 2016). However, a CNN can be formulated in terms of spectral graph theory, with graph signal processing (Shuman et al. 2012) as a basis, which uses harmonic analysis for a signal defined on irregular graph structures, resulting in a *Spectral Graph Convolution (SGC)* or *localized graph filter* (Parisot et al. 2017).

As described by Kipf and Welling (2016), the goal for a GCN is to learn a function of signals/features on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. $\mathcal{G}$ takes a feature vector $\mathbf{x}_i$ for every node i, summarized as an input feature matrix (*signal*) $X \in \mathbb{R}^{N \times C}$ and a graph structure description as an input, usually in the form of an adjacency matrix A, and produces an output feature matrix (*convolved signal*) $Z \in \mathbb{R}^{N \times F}$. N denotes the number of nodes, while C and F denote the number of input and output features, respectively.

A neural network layer can be defined as the function

$$H^{l+1} \quad = \quad f(H^l, A), \tag{3.12}$$

and a simple layer-wise propagation rule as

$$f(H^l, A) \quad = \quad \sigma\Big(AH^lW^l\Big), \tag{3.13}$$

where $H^0 = X$, $H^L = Z$ (with $l = 0, \ldots, L$ denoting the layer count), W^l is a weight matrix for layer l and $\sigma(\cdot)$ is an activation function, for example ReLU (equation 3.10). There are two major improvements to equation 3.13: first is $\tilde{A} = A + I_N$, I_N being the identity matrix to add a self-connection to every node; the second improvement is to normalize A. Without, multiplication with A would change the scale of the feature vectors, as Kipf and Welling (2016) explain, and to get rid of this problem, we introduce the diagonal node degree matrix D. The multiplication of the inverse of D with A corresponds to taking the average of neighboring node features. Empirically estimated, it is better to use a symmetric normalization, as this no longer results in only averaging the neighboring nodes, i.e. $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ (Kipf and Welling 2016).

Both these enhancements bring us to the final propagation rule

$$f(H^l, A) \quad = \quad \sigma\Big(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^l W^l\Big), \tag{3.14}$$

with $\tilde{D}$ as the diagonal node degree matrix of $\tilde{A}$.

We introduced the propagation rule of a multi-layer GCN, which draws similarities to the propagation rule of regular NNs, in the preceding paragraphs. The following gives a very shallow overview of how the propagation rule is originated from a first-order approximation of localized spectral filters on graphs. The reader is encouraged to consult the primary and secondary sources (Defferrard et al. 2016; Kipf and Welling 2016; Parisot et al. 2017; Shuman et al. 2012) for an in-depth discussion of how the propagation rule is approximated, as its genesis is only of peripheral importance for understanding the contribution of this thesis.

The *convolution theorem* states that a convolution in one domain equals the point-wise multiplication in another domain (Arfken et al. 2005). This is exploited by SGCs for a Fourier transform in the frequency domain. As Parisot et al. (2017) explain, the graph Fourier transform is defined by analogy to the Euclidian domain from the eigenfunctions of the Laplace operator. The eigenfunctions of the eigendecomposition of the graph Laplacian associated with low frequencies vary slowly across the graph, which translates to the realization that vertices connected by an highly weighted edge have similar values in the corresponding locations of these eigenvectors (Parisot et al. 2017). A spatial signal can be defined as a graph Fourier transform on a graph, being a base for spectral convolutions of a signal with a filter. By taking the preceding information and following the mathematical transformations of Kipf and Welling (2016), we arrive at the propagation rule in equation 3.14.

4 Implementation

In this chapter, we will explain the main scientific contributions of this thesis. As briefly described in chapter 1, we will process the Brown text corpus with its tagset into a graph and feed it into a GCN to perform supervised node classification.

Kipf and Welling (2016) provide an implementation of their GCN[9] to accompany their research paper, which makes use of the machine learning library TensorFlow (Abadi et al. 2016) with the high-level Keras API (Chollet et al. 2015) on top of it. We will use this implementation as our base mostly as is. The preset hyperparameters in the default implementation showed the best results in the experiments of Kipf and Welling (2016). We start with these hyperparameters, which are also used for examples and explanations throughout this chapter unless stated otherwise, and optimize them as needed.

We structure this chapter by illustrating how we construct the graph and then show by way of example, how a 2-layer GCN implementation is used for supervised classification on the built graph.

4.1 Constructing the Graph

In figure 1.1, we exemplarily showed a tagged sentence from the Brown corpus. The NLTK (Bird et al. 2009) provides a preprocessed version of said corpus and makes words and corresponding tags easily accessible. We also want to equip the words with features, which we get from pre-trained *word (feature) vectors* (also called *word embeddings*) provided by *GloVe* (Pennington et al. 2014). Machine learning algorithms are in general unable to process strings in raw form and therefore require a conversion of strings to numbers. The GloVe algorithm does exactly this by producing word vectors. Word vectors are vector representations for words in a word-word co-occurrence matrix, which indicates how frequently words co-occur with one another in a given corpus. When positioned in the word vector space, words with common contexts in the corpus are located in close proximity to one another.

We assume that these vector representations are suitable as features for our graph as word vectors usually improve NLP tasks (Ling et al. 2015). We proceed to build our data as shown in algorithm 4.1. There a two immediate limitations within this approach, which are caught by the algorithm: first, it is not guaranteed that every word in the brown corpus is included in the GloVe lookup table. Line 5 in the algorithm excludes these words. Second, because GloVe is treated as a lookup table, we cannot guarantee that the context correct word vector is chosen as features in the graph. In this context, the algorithm treats every word undifferentiated from other possible senses.

[9] `https://github.com/tkipf/keras-gcn`

Algorithm 4.1 Construction of the Brown Graph

Input: Brown word-tag tuple from NLTK **as** *brownTuple*
Input: GloVe word feature vectors look-up table **as** *gloveTable*
Output: Content file **as** *contentFile*
Output: Neighborhood file **as** *nhFile*

```
1: prevWord ← NULL
2: for every brownTuple do
3:     currWord ← word in brownTuple          ▷ extract current word into variable
4:     currTag ← tag in brownTuple            ▷ extract current tag into variable
5:     if currWord in gloveTable then
6:         currFeatures ← features for currWord in gloveTable
7:         write in contentFile : currWord + currFeatures + currTag
8:         if prevWord not NULL then
9:             write in nhFile : prevWord + currWord
10:        prevWord ← currWord
```

However, as every word is correctly tagged, we assume that this problem is negligible when we train the GCN.

In figures 4.1 and 4.2, we show snippets of the generated files. The way the files are constructed is predetermined by how the GCN is implemented, which was with the classification of citation networks in mind. The structure is easily transferable to the task of PoS-tagging. Word features can be found in between the word on the left and the PoS-tag on the right in the content file. The number of features per word is defined by the GloVe word vector size, which is the same for every word. The neighborhood file indicates that for every word, we write a line with the word itself separated with a whitespace from the next word. Note that a "word" in this context could also stand for characters other than alphabetic ones or punctuation (cf. chapter 1), for instance. Also note that in the actual implementation of our algorithm, we mapped the word strings to index numbers, for reasons mentioned in the beginning of this section.

```
⋮
implementation 0.68404 -0.5835 ... 1.6217 0.015818 0.41272 NN
of 0.70853 0.57088 ... -0.22562 -0.093918 -0.80375 IN
automobile -0.41195 0.069058 ... 0.16543 0.89073 -0.060983 NN
title -1.2383 0.99487 ... -0.97966 0.20244 -0.36069 NN
law -1.2328 -0.11042 ... 0.10342 0.20543 0.36536 NN
⋮
```

Figure 4.1: A snippet from the generated content file

The GCN treats the words as nodes with features and the neighborhood file as (undirected) edges between words in a graph and constructs a binary, symmetric adjacency matrix. When visualized, the graph can be imagined as a chain with a maximum of two edges per node, i.e. an edge from the word in focus to the previous and to the next word in the corpus. This means that *every* word in the corpus is treated as a node, independent of the fact that the same word with the same tag is multiply included in the graph. We reason this graph structure as follows: when chained, every word keeps its correct label. For PoS-tagging, other graph structures might lead to loss of information, e.g. one node for every word would make it unclear which tag is supposed to be used and falsify the classification process. We assume that multiply occurring nodes do not have a negative impact.

```
        ⋮
... implementation
implementation of
of automobile
automobile title
title law
law ...
        ⋮
```

Figure 4.2: A snippet from the generated neighborhood file

4.2 Examplary Multi-Layer GCN Used for Training

In the following, we outline a 2-layer GCN for supervised classification, based on Kipf and Welling (2016). The symmetric adjacency matrix $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is calculated in a preprocessing step to simplify the forward model:

$$Z = f(X, A) = \sigma_{\text{Softmax}}\Big(\hat{A}\,\sigma_{\text{ReLU}}\Big(\hat{A}XW^{(0)}\Big)\,W^{(1)}\Big), \tag{4.1}$$

where $W^{(0)} \in \mathbb{R}^{C\times H}$ is an input-to-hidden weight matrix for a hidden layer with H feature maps and $W^{(1)} \in \mathbb{R}^{H\times F}$ is a hidden-to-output weight matrix. The softmax activation function (equation 3.11) is applied row-wise. For supervised classification, the *cross-entropy error* is evaluated over all examples:

$$E_{\text{CrossEnt}} \quad = \quad -\sum_{l=1}^{L}\sum_{f=1}^{F} Y_{lf} ln Z_{lf}, \tag{4.2}$$

with L as the number of node indices and Y as the correct label at position lf.

The neural network weights $W^{(0)}$ and $W^{(1)}$ are trained using an optimizer, such as *Adam* (Kingma and Ba 2014) or batch GDA, on the dataset for every training iteration.

4.2.1 Experimental Setup

We run our experiments on a virtual machine operating Ubuntu 16.04.3 LTS with 72 GB RAM, using TensorFlow 0.12.0 and Keras 1.2.0. What follows is a list of exhaustive hyperparameters with their default values and alternative values. We tried to test every reasonable configuration and look for improvements. However, we do not adjust all hyperparameters, as improvements were not affected by these in our experiments.

Parameter	**Default**	**Alt. Val.**	**Definition**
Layers	2	3	Count of hidden layers + output layer
Hidden Units	16	32	Count of hidden units per hidden layer
Features	50	*none*	Count of features per word. Dependent on the chosen word vectors.
Dataset size	*n/a*	4000, 40000, 80000	Count of examples in content list
Splitsets	*n/a*	80/20	Ratio between training and validation set.
Optimizer	Adam (0.01)	*none*	Used Optimizer with learning rate
Epochs	200	400	Count of training iterations
Early stopping window size	10	20	Consecutive epochs without decrease in validation loss
Dropout rate for all layers	0.5	–	Dropout rate to prevent overfitting
L2 regularization for first layer	$5 \cdot 10^{-4}$	–	Applied on weights to prevent overfitting

Table 4.1: Table of hyperparameters for our experimental setup

Table 4.1 shows the list of hyperparameters. The keyword "*none*" indicates no changes to the default value, whereas "–" is used whenever the hyperparameter is removed and "*n/a*" when the default values are not applicable or unknown. The set size is chosen according to the memory, further explained in chapter 5. The splitsets describe, how the dataset is split into training, *validation* and test data and the ratio between these. We only focus on the ratio between training and validation data, which we chose by Pareto principle. Usually, when supervised learning is performed, the training set is split into two, where one of the sets is the validation set, which is, simply put, unlabeled data (Yang et al. 2016). Informally described, the model we train is applied to the validation data, which is a way to measure how well the trained model performs to tweak the hyperparameters. Validation loss is hereby the cross-entropy error applied on the validation set. The dataset size and splits deviate from the defaults by Kipf and Welling (2016), as we handle much larger datasets. The dropout (Srivastava et al. n.d.) and L2 regularization (Schmidhuber 2014) are ways to prevent a model from overfitting. The weights are initialized randomly with a specific probability distribution, described in (Glorot and Bengio 2010).

5 Results

In order to evaluate our results, we first want to refer to limitations of the GCN that negatively impact our experiments, described by Kipf and Welling (2016). One major problem of the GCN is the memory requirement, which grows linearly in the number of edges. Our full generated dataset with a file-size of 445MB consists of 1071201 edges and exceeds the RAM-size of 72GB when loaded into the GCN. We had to limit the amount of nodes and therefore edges in the dataset. The reduction is depicted in table 4.1. We chose the biggest dataset that fit into the RAM and took a reasonable long computation time. We assume that the chosen dataset sizes are sufficient enough to represent our model. We also experiment on a smaller tagset provided by NLTK with 12 different PoS-tags[10], as a smaller tagset — empirically stated — improves classification results.

Results are summarized in table 5.1 for the chosen sizes of datasets and another row of the small dataset with small tagset. Reported numbers denote mean classification accuracy in percent for 10 consecutive runs.

Parameter	**Small**	**Small w/ 12 tags**	**Medium**	**Large**
Default	16.12	39.94	15.29	13.35
Alternative	20.45	39.54	16.47	15.85

Table 5.1: Summary of results in terms of classification accuracy (in percent).

5.1 Evaluation

During testing, every dataset was terminated prematurely by the early stopping window size, either because the validation loss did not decline or because of a divide-by-zero error, which causes infinite validation loss. The generation of output predictions for the input samples causes a divide-by-zero when calculating the cross-entropy error. It is not entirely clear why and how this happens.

As we can see in table 5.1, a change of the listed hyperparameter improves the accuracy, alas only slightly. The biggest contributor is the removal of the hyperparameters affecting overfitting. Generally speaking, a bigger dataset is supposed to better the accuracy, while the opposite happens in our case. The main reason for this is the described early termination caused by the divide-by-zero error. The biggest dataset terminated after 12-20 epochs, which is not enough ”time”, speaking in terms of epochs, to improve the accuracy. However, even removing the window did not result in better accuracy, which

[10] `http://www.nltk.org/book/ch05.html`

is also true for the other dataset sizes. It can be speculated that the datasets converge in local minima, unable to escape for improvements. Changing the setsplit randomly did not either result in betterment.

For comparative reasons, we also experimented with the smaller tagset. Even though it suffers under the same problems as the other datasets, its results still shed some light on the effectiveness of the model with almost 40% accuracy. Still, the low accuracy prohibits us from instancing other measures, such as *found & actual*, *precision & recall* and the *F-measure*, which considers the latter tuple, to analyze the tagger coverage.

To summarize, the model fails to generalize and vastly underfits the data, since the underlying probability distribution is not properly estimated. The targeted adjustments of hyperparameters do not bring expected enhancements but merely bring consolation in an unsalvageable situation — at least when we speak of the current state of the model. In the following section, we will discuss ideas to improve the model.

5.2 Discussion

Our believe is that the biggest issue in the model is the graph itself. The chain graph does not seem to be exactly fit for the GCN. We thought of other graph structures that built on the chain, basically adding more edges to an already too large perceived graph, which is not desired. In their paper, Kipf and Welling (2016) state that GCNs are not limited to the assumption that edges merely encode similarity of nodes. However, this statement seems not to consider the task at hand. A structure of the graph that displays similarity of nodes could improve our model.

Even though our model failed to impress, we still believe that the smaller tagset acts as proof-of-concept and that the model will work when the graph construction is adjusted.

6 Conclusion and Future Work

We demonstrated how a GCN is potentially suited for solving the PoS-tagging task, when a text corpus is converted into a graph. Our model drastically underfits the data. We see the problems of our approach but also the room for optimization, which is arguably the biggest in the graph structure, as discussed in section 5.2. As we write this chapter, a very similar approach to our model to a similar task was proposed by (Marcheggiani and Titov 2017), which uses a GCN for *semantic role labeling* with very promising results. We see this as an additional affirmation to our proof-of-concept and are motivated for further development of our model. We are currently analyzing, how far their proposed concept is applicable to ours.

In more general terms, GCNs proved to be a good and easily applicable way to provide NN features to graph structures. We believe that there are other, more difficult NLP tasks that could be solved with GCNs, such as *Word sense disambiguation*, where the trend of recent research goes to graph-based approaches.

Acknowledgments

In remembrance of my grandfather.
I hereby express my sincerest gratitude to my advisor, Lukas Schmelzeisen, for making me his first advisee, for his unlimited patience, for his flexibility in terms of appointments and schedules, for his guidance during my research of course, for the many long, educational discussions we had, and for his overall help for making this thesis even possible. I could not have wished for someone better. I also thank my advisor, Steffen Staab, for the possibility of writing this thesis and for our encouraging communication. I am grateful to my family for making a good job of distracting me from writing this thesis — at times way too often for my own good. Finally, I thank my brother and my friend Fawzi Masri for proof-reading my thesis in the last seconds.

References

Abadi, Martín et al. (2016). „TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems“. In: *CoRR* abs/1603.04467.

Abu-Mostafa, Yaser S., Magdon-Ismail, Malik, and Lin, Hsuan-Tien (2012). *Learning From Data*. AMLBook.

Arfken, George B., Weber, Hans J., and Harris, Frank E. (2005). *Mathematical Methods for Physicists, Sixth Edition: A Comprehensive Guide*. 6th ed. Academic Press.

Bird, Steven, Klein, Ewan, and Loper, Edward (2009). *Natural Language Processing with Python*. 1st ed. O'Reilly Media, Inc.

Brants, Thorsten (2000). „TnT: a statistical part-of-speech tagger“. In: *Proceedings of the sixth conference on Applied natural language processing*, pp. 224–231.

Brill, Eric (1995). „Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging“. In: *Computational linguistics* 21.4, pp. 543–565.

Chen, Danqi and Manning, Christopher (2014). „A Fast and Accurate Dependency Parser using Neural Networks“. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, pp. 740–750.

Choi, Jinho D. (2016). „Dynamic Feature Induction: The Last Gist to the State-of-the-Art“. In: *Proceedings of the 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics*. NAACL'16, pp. 271–281.

Chollet, François et al. (2015). *Keras*. `https://github.com/fchollet/keras`.

Defferrard, Michaël, Bresson, Xavier, and Vandergheynst, Pierre (2016). „Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering“. In: *CoRR* abs/1606.09375.

Duvenaud, David K, Maclaurin, Dougal, Iparraguirre, Jorge, Bombarell, Rafael, Hirzel, Timothy, Aspuru-Guzik, Alan, and Adams, Ryan P (2015). „Convolutional Networks on Graphs for Learning Molecular Fingerprints“. In: *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., pp. 2224–2232.

Francis, W. Nelson and Kucera, Henry (1979). *Brown Corpus Manual: A Standard Corpus of Present-Day Edited American English*. Brown University Linguistics Department.

Giesbrecht, Eugenie and Evert, Stefan (2009). „Is part-of-speech tagging a solved task? An evaluation of POS taggers for the German web as corpus“. In: *Proceedings of the fifth Web as Corpus workshop*, pp. 27–35.

Glorot, Xavier and Bengio, Yoshua (2010). „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the Thir-*

teenth International Conference on Artificial Intelligence and Statistics, pp. 249–256.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Haykin, Simon (1998). *Neural Networks: A Comprehensive Foundation*. 2nd ed. Prentice Hall PTR.

Jurafsky, Daniel and Martin, James H. (2009). *Speech and Language Processing*. 2nd ed. Prentice-Hall, Inc.

Kingma, Diederik P. and Ba, Jimmy (2014). „Adam: A Method for Stochastic Optimization“. In: *CoRR* abs/1412.6980.

Kipf, Thomas N. and Welling, Max (2016). „Semi-Supervised Classification with Graph Convolutional Networks“. In: *CoRR* abs/1609.02907.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. (2012). „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*. NIPS'12. Curran Associates Inc., pp. 1097–1105.

Ling, Wang, Dyer, Chris, Black, Alan W, and Trancoso, Isabel (2015). „Two/Too Simple Adaptations of Word2Vec for Syntax Problems“. In: *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1299–1304.

Marcheggiani, Diego and Titov, Ivan (2017). „Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling“. In: *CoRR* abs/1703.04826.

Mitchell, Thomas M. (1997). *Machine Learning*. 1st ed. McGraw-Hill, Inc.

Nielsen, Michael A. (2015). *Neural Networks and Deep Learning*. `http://neuralnetworksanddeeplearning.com/`. Determination Press.

Parisot, Sarah, Ktena, Sofia Ira, Ferrante, Enzo, Lee, Matthew, Moreno, Ricardo Guerrerro, Glocker, Ben, and Rueckert, Daniel (2017). „Spectral Graph Convolutions on Population Graphs for Disease Prediction“. In: *arXiv preprint arXiv:1703.03020*.

Pennington, Jeffrey, Socher, Richard, and Manning, Christopher D. (2014). „GloVe: Global Vectors for Word Representation“. In: *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543.

Ratnaparkhi, Adwait et al. (1996). „A maximum entropy model for part-of-speech tagging“. In: *Proceedings of the conference on empirical methods in natural language processing*. Vol. 1, pp. 133–142.

Schmid, Helmut (1995). „Improvements in part-of-speech tagging with an application to German“. In: *In proceedings of the acl sigdat-workshop*.

Schmidhuber, Jürgen (2014). „Deep Learning in Neural Networks: An Overview“. In: *CoRR* abs/1404.7828.

Shuman, David I., Narang, Sunil K., Frossard, Pascal, Ortega, Antonio, and Vandergheynst, Pierre (2012). „Signal Processing on Graphs: Extending High-Dimensional Data Analysis to Networks and Other Irregular Data Domains“. In: *CoRR* abs/1211.0053.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *J. Mach. Learn. Res.* 15.1, pp. 1929–1958.

Subramanya, Amarnag and Talukdar, Partha Pratim (2014). *Graph-Based Semi-Supervised Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning 29. Morgan & Claypool.

Tian, Yuan and Lo, David (2015). „A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports“. In: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pp. 570–574.

Toutanova, Kristina, Klein, Dan, Manning, Christopher D., and Singer, Yoram (2003). „Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network“. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*. Vol. 1. NAACL '03. Association for Computational Linguistics, pp. 173–180.

Yang, Zhilin, Cohen, William W., and Salakhutdinov, Ruslan (2016). „Revisiting Semi-Supervised Learning with Graph Embeddings“. In: *CoRR* abs/1603.08861.