Masaryk University
Faculty of Informatics

Part-of-Speech Tagging Using Neural Networks

Master's thesis

Martin Frodl

Advisor: Mgr. Pavel Rychlý, PhD.

Brno, autumn 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Martin Frodl

Advisor: Mgr. Pavel Rychlý, PhD.

Acknowledgement

First of all I would like to thank to my advisor, Mgr. Pavel Rychlý, Ph.D. for guiding me throughout my studies and all of his insightful remarks concerning my thesis, as well as other projects.

I am grateful for the support from the people on whose life this thesis had a not insignificant impact: my parents and Jitka.

Also, I would like to thank Martina, who kindly offered to bind this thesis into a book.

A special thanks goes to the staff of the Happy Teahouse where a substantial part of this text came into existence.

Abstract

Part-of-speech tagging, the process of assigning parts of speech to words in sentences, has a vast field of applications in natural language processing. It constitutes an important intermediate step in other tasks such as syntactic analysis or machine translation. Out of the methods that have been employed in solving this problem, neural networks belong to the rather non-typical ones, being often neglected in textbooks. In this work we provide an overview on some notable attempts that have been made in part-of-speech tagging with neural networks. Based on these, we also propose our own tagger based on similar principles. The tagger provides two rather different training methods that can be chosen freely. The first method employs a set of recurrent multilayer perceptron networks which learn the most likely tags from the word-to-tag probabilities of the words within a context. The second method converts words into feature vectors in a multidimensional space; subsequently, the hyperplanes separating the data in one class from the other ones are searched for using perceptrons. An additional statistical method is available as a baseline to compare the performance. Training on the first 999,998 words in the Brown corpus and evaluating on the rest, the best accuracy was 94.93%, achieved by the first method. The second method was significantly more successful for smaller training sets, nevertheless, long training times prevented us from determining the accuracy for the largest set. Both methods did systematically better than the baseline.

Shrnutí

Značkování slovních druhů, tj. proces přiřazování značek slovních druhů jednotlivým slovům ve větách, má značné množství aplikací v oblasti zpracování přirozeného jazyka. Představuje důležitý mezikrok v dalších úlohách, jakými je např. syntaktická analýza nebo strojový překlad. Neuronové sítě představují jeden z méně typických přístupů ke značkování slovních druhů a bývají v učebnicích nezřídka opomíjeny. V této práci předkládáme základní přehled některých význačných pokusů, které byly v oblasti značkování slovních druhů neuronovými sítěmi učiněny. Na jejich základě pak navrhujeme vlastní tagger založený na analogických principech. Tagger disponuje dvěma učicími metodami, mezi nimiž lze volit. První metoda využívá sadu rekurentních vícevrstvých sítí, které se učí nejpravděpodobnější značky na základě podmíněných lexikálních pravděpodobností vztahujících se ke slovům a značkám v kontextovém okně. Druhá metoda převádí slova na rysové vektory v mnohadimenzionálním prostoru; v tom jsou pak s využitím perceptronů hledány nadroviny oddělující vektory příslušející ke konkrétní značce od ostatních. Pro srovnání výkonnosti je k dispozici také třetí, statistická metoda. Při trénování taggeru na prvních 999 998 slovech Brownova korpusu a vyhodnocení na zbývající části bylo první metodou dosaženo nejlepší přesnosti 94.93%. Výsledky pro druhou metodu byly pro tři nejmenší testovací sady výrazně lepší oproti metodě první, vzhledem k dlouhým časům běhu však nebylo možné přesnost pro největší trénovací sadu. Obě metody dosáhly lepších výsledků než metoda třetí.

Keywords

part of speech, part-of-speech tagging, tagger, neural network, neuron, perceptron, MLP, slovní druhy, značkování slovních druhů, neuronové sítě

Contents

1 Introduction

Part-of-speech ambiguity is a common feature of a majority of the world's languages. Considering the English word *left* as an example, we can observe that in different situations it functions as an adjective (as in *the left mouse button*), an adverb (*turn left*), a noun (*I was sitting to his left*) or a verb, either a past tense form (*he left his wife*) or a past participle (*she was left wondering*). A substantial amount of words is ambiguous. In Brown corpus, an English corpus of about 1,000,000 words each of which was manually assigned to a part-of-speech category, about 16% of the words appear as two or more different part of speech. Programs which assign parts of speech to words in a plain text corpus automatically are commonly termed 'taggers'. These are typically trained on manually tagged corpora beforehand in order to learn certain statistical properties of the target language, such as that the sequence determiner-adjective-noun is more likely to appear than, say, preposition-verb-preposition.

The task of part-of-speech tagging, as the process of assignment part-of-speech categories to words in sentences is called, has a vast range of applications in natural language processing. Assigning parts of speech to words is often followed by a syntactic analysis (parsing), a process whereby the sentence structure is determined, such as finding the subject, predicate and object. This can provide a better insight into the information structure of the sentence, making it possible to tell something about the meaning of the sentence. Part-of-speech disambiguation also plays an irreplaceable role in machine translation: translating the aforementioned examples featuring the word *left* to Czech gives ***levé*** *tlačítko myši*, *zahnout* ***doleva***, *seděl jsem* ***nalevo*** *od něj*, ***opustil*** *svou ženu* and ***zůstala*** *v údivu*, respectively. For the tools making use of part-of-speech tags to yield reasonable results, it is essential that taggers assign the tags with as high an accuracy as possible. Tagging the noun in the subject position of a sentence as e.g. a verb can have a significant negative impact on the resulting parse tree or the translation.

A number of techniques have been applied to part-of-speech tagging over history and even nowadays there is no clear answer to the question of which approach is the best. The most common methods can be grouped into a handful of classes. *Statistical taggers* try to learn various kinds of probabilities from a training corpus and use them to find the most likely tag sequence for each sentence to be tagged. *Rule-based taggers*, as the name indicates, use a set of rules to assign tags; for instance, if a word ambiguous between noun and verb is preceded by *to*, choose the tag 'verb'. *Information-theoretical approaches* try to find such context clues which best indicate a word's most likely part-of-speech tag. Last but not least, *neural networks* of an aptly chosen architecture have been repeatedly employed in the task as well.

The position of researchers towards neural networks has been ambiguous ever since their invention in the mid-20th century. This machine learning technique was motivated by processes which take place in the nervous systems of animals, including the human brain. The parallel with the human brain, a fascinating computing machine of which we still have rather fragmentary knowledge, is probably a major reason why artificial neural networks keep coming to popularity every now and then, even though many machine learning tasks can be performed more effectively by other techniques like support-vector machines. Part-of-speech tagging is a task for which neural networks have been used comparatively seldom. In fact, many textbooks on natural language processing do not consider neural networks when dealing with part-of-speech tagging at all, as is the case with Jurafsky and Martin (2009). The reasons as to why neural networks did not make it into the mainstream of the field can be several, including generally rather long training times or unconvincing performance; also the fact that the models work with many parameters whose values have to be estimated experimentally (such as the number of neurons and the connections between them) makes neural networks a little difficult to work with.

In the following text we are going explore more in detail how neural networks can be applied to part-of-speech tagging and also present our own tool which seeks to accomplish the task. Essential theoretical concepts will be introduced first in order to provide a fundament on which the rest will be built. The chapter immediately following after this one describes basic principles of part-of-speech tagging in general, explaining the difference between discriminative and generative models. Chapter 3 serves as a brief introduction into neural networks which includes the computation mechanisms of neuron units and multilayer networks together with some algorithms used for their training. In chapter 4, we give a historical overview of some notable cases in which neural networks were employed in part-of-speech tagging. Inspired by these, we propose a new tagger based on neural networks, described in detail in chapter 5. Two tagging methods are available, along with a third, statistical method which was used as a baseline. Finally, chapter 6 compares the performances of the individual methods based on experiments that were done using the Brown corpus. In addition, some possible improvements are suggested that could be done in order to make the tagger better in some respects.

2 Basic principles of part-of-speech tagging

The purpose of this chapter is not to give an exhaustive overview on all the techniques that are or ever have been used to solve the part-of-speech tagging task. What we are seeking to do here is provide an essential theoretical background for the things that will be dealt with in the subsequent chapters; in particular, techniques that are going to have at least some relevance to the methods we are proposing in this work are described here. Since employing neural networks in solving the task is a topic of special interest to us, a separate chapter will be devoted to this matter and we are not going to discuss it here.

We are going to use certain conventions in the remaining text. The notation $w_1 \dots w_n$, alternatively written as w_1^n refers to a bare (untagged) sequence of n words or tokens. The terms *word* and *token* are to be understood as interchangeable, so 'words' include also various non-lexical items such as punctuation marks, frequently found in corpora. The set of all words in a vocabulary learnt from a corpus will be denoted as $\mathcal{W}$, its cardinality being W. A sequence of n part-of-speech tags will be written as $t_1 \dots t_n$ or briefly t_1^n. The tagset corresponding to a particular corpus will be denoted as $\mathcal{T}$, the cardinality of this set is T. The notations $\mathcal{W}^n$ and $\mathcal{T}^n$ refer to the sets of all possible word or tag sequences of the length n, respectively. In case that only a subsequence of a word or tag sequence is being dealt with, the shorthand notations $w_k^l = w_k w_{k+1} \dots w_{l-1} w_l$ and $t_k^l = t_k t_{k+1} \dots t_{l-1} t_l$ are going to be used.

2.1 Discriminative and generative models

2.1.1 Discriminative models

Given a sequence of tokens $w_1^n = w_1 \dots w_n$, the common goal of all part-of-speech tagging techniques is to find the most likely sequence of tags $t_1^n = t_1 \dots t_n$. Formally put, we are trying to find such a sequence

$\hat{t}_1^n$ among all the possible sequences t_1^n such that

$$\hat{t}_1^n = \arg\max_{t_1^n \in \mathcal{T}^n} P(t_1^n | w_1^n). \tag{2.1}$$

Since it is impossible to find the exact values of $P(t_1^n|w_1^n)$ for each pair of sequences w_1^n, t_1^n, various approximation techniques are used to estimate them. By definition of conditional probability,

$$P(t_1^n|w_1^n) = \frac{P(w_1^n, t_1^n)}{P(w_1^n)}. \tag{2.2}$$

This can be easily seen to be equal to

$$\frac{P(w_1^n, t_1^n)}{P(w_1^n)} = \frac{P(w_1^n, t_1)}{P(w_1^n)} \cdot \frac{P(w_1^n, t_1^2)}{P(w_1^n, t_1)} \cdot \dots \cdot \frac{P(w_1^n, t_1^n)}{P(w_1^n, t_1^{n-1})}, \tag{2.3}$$

which can be expressed in terms of conditional probabilities as

$$P(t_1^n|w_1^n) = P(t_1|w_1^n) \cdot P(t_2|w_1^n, t_1) \cdot \dots \cdot P(t_n|w_1^n, t_1^{n-1}). \tag{2.4}$$

Now, instead of computing the probability $P(t_1^n|w_1^n)$, it suffices to find the values of each factor in 2.4 and calculate the product instead. Even though the task itself is not any more feasible than the original one if insisting on the precise values, it makes it possible to make certain simplifying assumptions. These adopted, estimating the conditional probabilities need not be an insurmountable problem any longer.

Two kinds of assumptions are frequently employed in part-of-speech taggers, both cropping the context that is taken as relevant for the current tag. The first one limits the number of words on which the current tag depends. Typically, only one word is taken into consideration, i.e.

$$P(t_i|w_1^i, t_1^{i-1}) \approx P(t_i|w_i, t_1^{i-1}). \tag{2.5}$$

The other assumption concerns the number of preceding tags which affect the probability of the current tag. Depending on this, the corresponding taggers are termed *unigram* (no previous tags are taken account of), *bigram* (one preceding tag), *trigram* (two preceding tags) or *quadrigram* (three preceding tags). Due to the sparse data problem, longer context are hardly ever made use of in existing taggers. Considering now the trigram assumption as an example, a tag probability is approximated as

$$P(t_i|w_1^i, t_1^{i-1}) \approx P(t_i|w_1^n, t_{i-2}^{i-1}). \tag{2.6}$$

Taking both assumptions together, we yield the following approximation, frequently used in existing taggers:

$$P(t_i|w_1^i, t_1^{i-1}) \approx P(t_i|w_i, t_{i-2}^{i-1}). \tag{2.7}$$

By substituting into equation 2.4 and subsequently 2.1, the most likely tag sequence t_1^n for given w_1^n can be found as

$$\hat{t}_1^n = \operatorname*{arg\,max}_{t_1^n \in \mathcal{T}^n} \prod_{i=1}^{n} P(t_i|w_i, t_{i-2}^{i-1}). \tag{2.8}$$

For practical reasons, it is more convenient to take a logarithm (to an arbitrary base) of the product; as the logarithm is a monotonically increasing function, it does not affect the result of the arg max operator.

$$\begin{aligned} \hat{t}_1^n &= \operatorname*{arg\,max}_{t_1^n \in \mathcal{T}^n} \left[\log \left(\prod_{i=1}^{n} P(t_i|w_i, t_{i-2}^{i-1}) \right) \right] \\ &= \operatorname*{arg\,max}_{t_1^n \in \mathcal{T}^n} \sum_{i=1}^{n} \log P(t_i|w_i, t_{i-2}^{i-1}) \end{aligned} \tag{2.9}$$

Models which estimate the probability of a tag sequence in a way like the one in 2.8 are commonly referred to as *discriminative models*. These constitute one of the two major classes of statistical taggers. The idea behind this technique is fairly straightforward: provided a training set of labeled data, the model strives to learn conditional probabilities of each tag in combination with every possible context; be they estimated from the respective conditional relative frequencies or in a different way.

2.1.2 Generative models

An alternative approach, possibly a little less intuitive for the first sight, is represented by the so-called *generative models*. We have just seen that discriminative models estimate the posterior probability of each sequence of tags given a particular word sequence. Generative models, on the other hand, compute prior probabilities of the possible tag sequences together with likelihoods of the word sequence being 'generated' by the particular tag sequence. The following text should make it clear how to understand this—so far perhaps a bit obscure—terminology.

Let us go back to equation 2.1 again. It can be converted into a different form which can be more suitable for certain uses. We can apply Bayes' rule to the term $P(t_1^n|w_1^n)$, which yields

$$\hat{t}_1^n = \arg\max_{t_1^n \in \mathcal{T}^n} P(t_1^n|w_1^n) = \arg\max_{t_1^n \in \mathcal{T}^n} \frac{P(t_1^n)P(w_1^n|t_1^n)}{P(w_1^n)} \tag{2.10}$$

For a given word sequence w_1^n, the probability $P(w_1^n)$ is effectively a constant. While its exact value is unknown, it is clear that dividing the numerator by this constant does not affect the result of the arg max operator in any way and as such can be left out from the formula:

$$\hat{t}_1^n = \arg\max_{t_1^n \in \mathcal{T}^n} P(t_1^n)P(w_1^n|t_1^n) \tag{2.11}$$

It should be obvious by now why models using this approach are termed 'generative'. For each possible sequence t_1^n, we estimate two probabilities: $P(t_1^n)$, the probability of the given tag sequence no matter what the words in w_1^n are, and $P(w_1^n|t_1^n)$, the probability of the sentence among all possible sentences with the same structure, i.e. those which are 'generated' by t_1^n.

Both probabilities on the right-hand side can be approximated if simplifying assumptions similar to those in 2.5 and 2.6 are made. Adopting, say, the trigram assumption as in 2.6, $P(t_1^n)$ can be converted to the form

$$P(t_1^n) = P(t_1)P(t_2|t_1)P(t_3|t_1^2)\cdots P(t_n|t_{n-2}^{n-1}) = \prod_{i=1}^{n} P(t_i|t_{i-2}^{i-1}). \qquad (2.12)$$

As for the probability $P(w_1^n|t_1^n)$, the assumption most frequently made is that the probability of a word appearing at given position only depends on the corresponding tag, not on the other tags or words in the sequence. (Note that this assumption is not equivalent to the one made in 2.5.) The probability $P(w_1^n|t_1^n)$ can thus be simplified to

$$P(w_1^n|t_1^n) = \prod_{i=1}^{n} P(w_i|t_i). \qquad (2.13)$$

Taking now equations 2.12 and 2.13 together and substituting into equation 2.11, we come to the following formula, central to many existing taggers:

$$\begin{aligned}\hat{t}_1^n &= \arg\max_{t_1^n \in \mathcal{T}^n} P(t_1^n)P(w_1^n|t_1^n) \\ &= \arg\max_{t_1^n \in \mathcal{T}^n} \prod_{i=1}^{n} P(t_i|t_{i-2}^{i-1}) \prod_{i=1}^{n} P(w_i|t_i) \\ &= \arg\max_{t_1^n \in \mathcal{T}^n} \prod_{i=1}^{n} P(t_i|t_{i-2}^{i-1})P(w_i|t_i)\end{aligned} \qquad (2.14)$$

The sum of log probabilities can be used here for computation of the most likely tag sequence instead of the product of probabilities in the

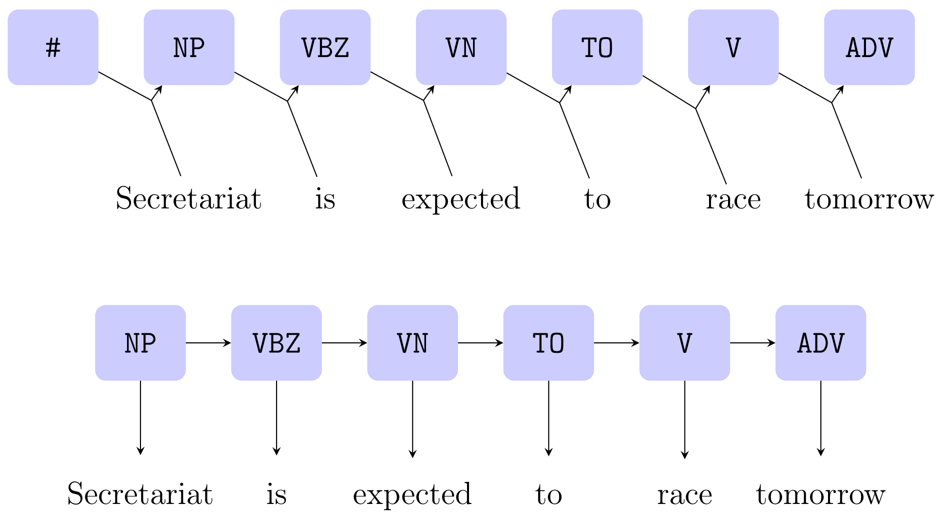

Figure 2.1: Illustration of the fundamental difference between discriminative and generative models, adapted from Jurafsky and Martin (2009). In discriminative models, the probability of a tag sequence is obtained by multiplying the probabilities of individual tags in given contexts. In generative models, on the other hand, two kinds of probabilities are used: transition probabilities, telling how likely a given sequence (in this case, pair) of tags is to appear, and emmission probabilities, stating how likely each single word is to appear given the corresponding tag.

same way as it was done in equation 2.9:

$$\hat{t}_1^n = \operatorname*{arg\,max}_{t_1^n \in \mathcal{T}^n} \sum_{i=1}^{n} \log P(t_i|t_{i-2}^{i-1}) \cdot \log P(w_i|t_i) \tag{2.15}$$

2.1.3 Viterbi algorithm

However simple the formula 2.15 for the most likely tag sequence may seem in theory, it does not give straightforward instructions how to find $\hat{t}_1^n$ within a reasonable time frame. A naïve approach taking all possible tag sequences of the length n would fail to accomplish this task (except for a couple values of n close to zero) due to the exponential complexity. Fortunately, a dynamic programming algorithm for solving Hidden Markov Models was developed by Andrew Viterbi in 1967, which guarantees to find the optimal solution in a significantly shorter time. The algorithm is shown in figure 2.2. It basically consist in creating a table in which the log probabilities of each tag are computed for each w_i. The probabilities calculated for each word are reused in computing the probabilities of the word that follows. Once the table has been filled, the path which maximizes the overall probability of a tag sequence (the Viterbi path) is found and the most likely tag sequence is returned.

Notation

Let $\mathcal{W} = \{w_1, \ldots, w_{|\mathcal{W}|}\}$ be the set of all possible observations, $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}$ the set of all possible states. Let further Tr be a transition matrix $|\mathcal{T}| \times |\mathcal{T}|$ where $Tr_{ij} = \log P(t_j|t_i)$ and Em an emission matrix $|\mathcal{T}| \times |\mathcal{W}|$ where $Em_{ij} = \log P(w_j|t_i)$. Let $init : \mathcal{T} \to \mathbb{R}$ be a mapping such that $init(i) = \log P(t_i|\#)$, i.e. initial log probability of t_i. Let $(\hat{w}_1, \ldots, \hat{w}_n) \in \mathcal{W}^n$ be a sequence of observations.

Algorithm

▷ For each state t_i, set $A_{i1} = init(i) + Em_{i1}$, $B_{i1} = 0$.

▷ For $k = 2, \ldots, n$:

- ⋆ For each state t_i, compute the values A_{ij}, B_{ij} as follows:

$$A_{ij} = \max_k (A_{k,j-1} + Tr_{ki} + Em_{im}) \tag{2.16}$$

$$B_{ij} = \arg\max_k (B_{k,j-1} + Tr_{ki} + Em_{im}) \tag{2.17}$$

  where m is a number satisfying the equality $\hat{w}_k = w_m$.

▷ Set $z_n = \arg\max_k A_{kn}$ and then $\hat{t}_n = t_{z_n}$.

▷ For $k = n-1, \ldots, 1$:

- ⋆ Set $z_k = B_{z_{k+1},k+1}$ and then $\hat{t}_k = s_{z_k}$.

▷ Return $\hat{t}_1^n = \hat{t}_1 \ldots \hat{t}_n$.

Figure 2.2: Viterbi algorithm

3 Neural networks

Neural networks are a machine learning technique incorporating principles that have been observed in biological nervous systems. A biological neuron is a cell consisting of a *body*, multiple smaller protrusions called *dendrites* and a single long protrusion called an *axon*. On the surface of the dendrites are found numerous spots called *synapses*, whereby axons of other neurons are connected to the neuron. Every now and then, the axons fire an electric impulse which affects the permeability of the cell membrane in such a way that the voltage inside the neuron body increases slightly. The more activations come from other neurons, the higher the voltage grows. At the base of the axon is a center which 'measures' the voltage permanently. As long as the voltage value lies below a certain threshold, nothing particular happens. Once this threshold has been exceeded, the neuron fires an *action potential*, meaning that a wave of high voltage starts propagating towards the end of the axon. This axon can be attached to the dendrites of one or more other neurons, where the process is repeated in an analogous way.

This simplified mechanism of function of biological neurons is simulated by artificial neuron networks. Like their biological counterparts, also artificial neurons are units which contain one or more inputs and a single output. Depending on how much non-zero inputs the neuron receives, it either remains inactive, giving a zero output, or generates an action potential, represented by a non-zero output. Multiple neurons can be connected to one another, producing an (artificial) neural network. The following sections will describe the functioning of neural networks in more detail. This chapter draws mainly on Šíma and Neruda (1996), to which book the reader should refer for more information.

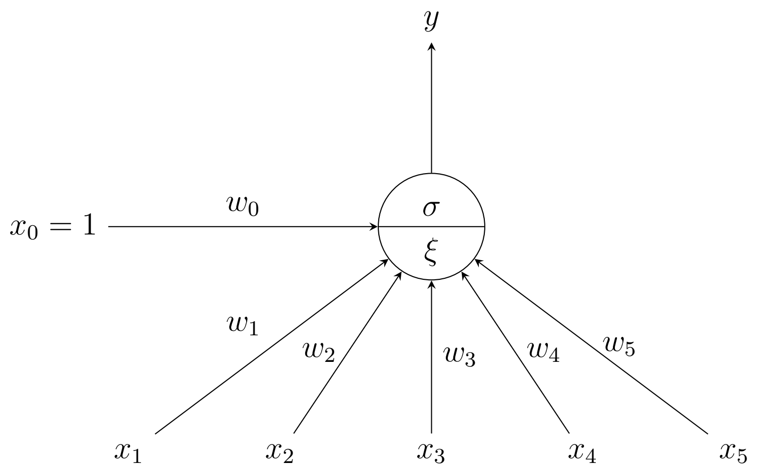

Figure 3.1: An example neuron unit with five inputs and bias

3.1 Perceptron

The basic constituent unit of most neural networks is called a *perceptron* or simply an *(articifical) neuron.* It can be viewed as a simplified model of a biological neuron. Much like its biological counterpart, a perceptron is sort of a 'black box' with some fixed number of inputs (sometimes referred to as *dendrites*) and a single output (also known as the *axon*). Each input has a corresponding weight indicating the extent to which the particular connection influences the resulting output.

The rough idea described above can be formalized as follows. Let $\mathbf{X} = (x_1, \ldots, x_n)$ be the vector of input values and $\mathbf{w} = (w_1, \ldots, w_n)$ the vector of the corresponding weights. To calculate the output, we need to compute the weighted sum $\xi = \mathbf{wX} = \sum_{i=1}^{n} w_i x_i$ (sometimes called the *inner potential* of the neuron) first. The weighted sum is subsequently passed to the activation function σ, yielding the output $y = \sigma(\xi) = \sigma(\sum_{i=1}^{n} w_i x_i)$.

The activation function σ may be of various kinds, depending on the task the neuron is designed to perform. In the case of (two-class) classification, *threshold activation functions* are widely used. A threshold

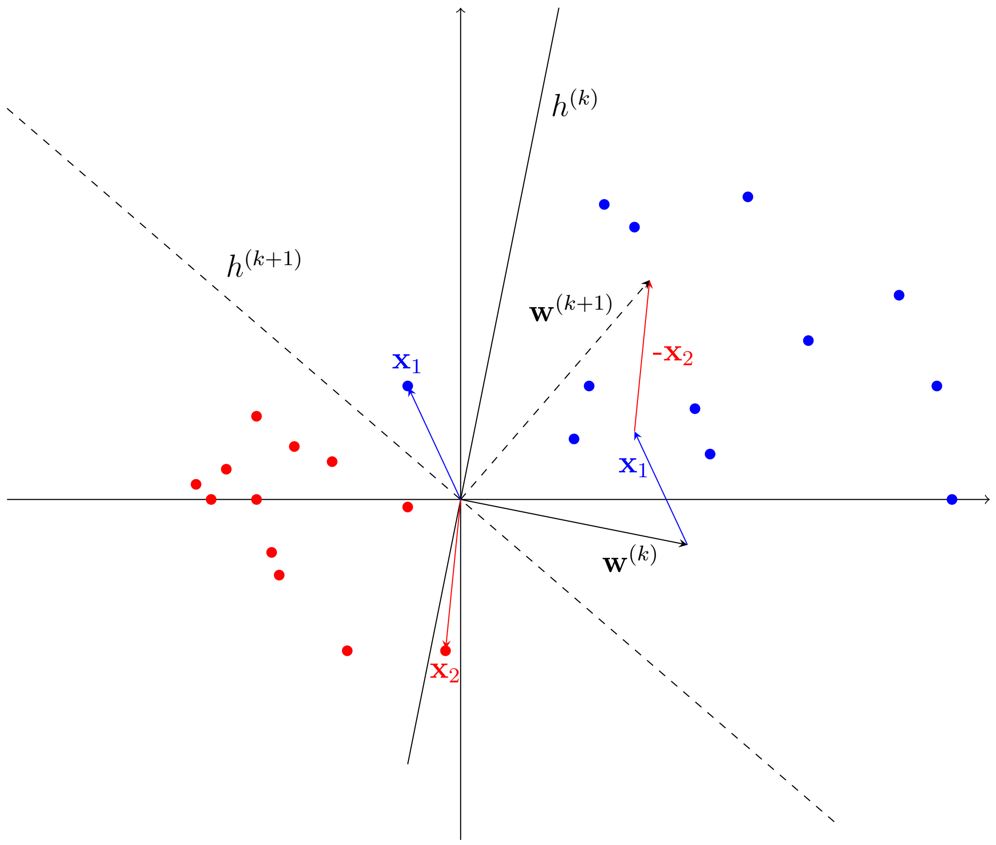

Figure 3.2: Geometric interpretation of perceptron classification and learning. The (two-dimensional) sample vectors are split into two groups, labeled either as 0 (red points) or 1 (blue points). The current weight vector $\mathbf{w}^{(k)}$ defines a hyperplane $h^{(k)}$ which splits the vector space into two half-spaces. Samples in one half-space (the 'left' one in the graph) are classified as 0, samples in the other one as 1. This configuration gives rise to two classification errors: one false negative ($\mathbf{x}_1$) and one false positive ($\mathbf{x}_2$). The new weight vector $\mathbf{w}^{(k+1)}$ is obtained from the old one by adding all the vectors corresponding to the false negative samples (here only $\mathbf{x}_1$) and subtracting the false positive vectors ($\mathbf{x}_2$ in our case). (In fact, the error vectors are multiplied by the parameter ε first; here $\varepsilon = 1$.) It is the normal vector of a new separating hyperplane $h^{(k+1)}$, which already classifies all the samples correctly.

activation function has a general form

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq h, \\ 0 & \xi < h. \end{cases} \tag{3.1}$$

where h is an arbitrary (but fixed) real-valued parameter. Like in a biological neuron, the 'action potential' (i.e. non-zero output) is generated whenever the weighted sum of inputs exceeds a pre-defined threshold value and vice versa.

A variant of an artificial neuron is a neuron with bias. Such a neuron contains an extra formal input x_0 whose value is always 1. This input has a corresponding weight w_0. When the weighted sum is computed, the product $w_0 x_0 = w_0$ is also included in it, effectively adding w_0 to the weighted sum of the remaining inputs. Note that if $w_0 = -h$ where h is the threshold mentioned above, the activation function σ can be equivalently defined as

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0, \\ 0 & \xi < 0. \end{cases} \tag{3.2}$$

where $\xi = \sum_{i=0}^{n} w_i x_i$ (the sum starts from 0 instead of 1). This approach is more convenient for many uses, notably for learning, as will be shown in a short while. Henceforth, any mention of the term 'neuron' is to be understood as a neuron with bias unless specified otherwise.

3.2 Perceptron learning algorithm

Considering what has been said so far, a neuron can be viewed as an n-ary function from $\mathbb{R}^n$ to the set $\{0, 1\}$. For each n-dimensional real vector, it returns either 0 or 1, the choice being made as defined in 3.2. Note that $\xi = w_0 x_0 + \ldots + w_n x_n = 0$ is an equation of an $(n-1)$-dimensional hyperplane which separates the space $\mathbb{R}^n$ into two half-spaces. The first half-space contains precisely those vectors $\mathbf{X}$ for which $\xi(\mathbf{X}) \geq 0$; the other half-space those for which $\xi(\mathbf{X}) < 0$. This hyperplane is fully defined by the vector $\mathbf{w} = (w_0, \ldots, w_n)$: the components $(w_1, \ldots, w_n)$ constitute the normal vector of the hyperplane, while the terms $-\frac{w_0}{w_1}, \ldots, -\frac{w_0}{w_n}$ define its intersection points with the corresponding axes.

Thus, in order to maximize the accuracy of classification, it is necessary to find a suitable hyperplane that will correctly separate the data of one class from the other ones. This can be accomplished by employing the *perceptron algorithm*, a supervised learning algorithm which adapts the weights based on labeled training data.

Let $\mathcal{T} = \{(\mathbf{x}_1, d_1), \ldots, (\mathbf{x}_p, d_p)\}$ be the set of labeled training samples, where $\mathbf{x}_k = (x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^n, k = 1, \ldots, p$, is the k-th input vector and $d_k \in \{0, 1\}$ the desired value of the corresponding output. (Note that the vector $\mathbf{x}_k$ does not contain x_{k0}, which is always set to 1, so it need not be passed in the input vector.) Let further $\mathbf{X}_k = (x_{k0}, \ldots, x_{kn}) = (1, x_{k1}, \ldots, x_{kn})$ for each $k = 1, \ldots, p$.

In the course of the algorithm, the weight vector $\mathbf{w}$ is modified step by step, generating the sequence $\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots$ At the very beginning, $\mathbf{w}^{(0)}$ is initialized with randomly generated values close to 0. In every subsequent step, the weights are computed as

$$\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} + \varepsilon \cdot \sum_{k=1}^{p} (d_k - \sigma(\mathbf{w}^{(t-1)} \mathbf{X}_k)) \cdot \mathbf{X}_k, \tag{3.3}$$

where ε a constant expressing the learning rate $(0 < \varepsilon \leq 1)$. In particular, if the neuron output $y_k = \sigma(\mathbf{w}^{(t-1)} \mathbf{X}_k)$ equals the desired output d_k, no change in the weight vector is made, i.e. $\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)}$. If this is not the case, the vector $\mathbf{w}$ is modified in such a way that the Euclidean distance between $\mathbf{X}_k$ and the half-space to which should belong is decreased.

It was shown already by Frank Rosenblatt that the algorithm always converges, provided that the sets of examples labeled with 0, resp. 1 are *linearly separable*. That is to say, if a hyperplane exists such that all examples labeled as 0 are located on one side of it and all examples labeled as 1 on the other, the algorithm guarantees to find it in a finite time.

The condition of linear separability imposes a strong limitation on the data to which the perceptron algorithm can be applied. Not only it fails to learn functions as simple as the logical XOR, but—more importantly—the algorithm is not robust enough to cope with outliers. A single outlier in the vicinity of the other cluster is enough to make the algorithm run ad infinitum without success. Since real-life data are generally not linearly separable, other techniques must be employed to build a succesful classifier.

3.3 Fast perceptron training

The basic perceptron algorithm, as formulated in the previous section, raises a couple of questions which ought to be discussed first in order to boost the algorithm's effectivity (most importantly, the rate of convergence). First, what values should the weights be set to during the initialization? And second, how to choose the optimal value for the parameter ε, i.e. learning rate? Should it be constant, or is it better for it to change in each iteration?

One method to tackle these issues was proposed by Gkanogiannis and Kalamboukis (2009). In their approach, the centers of both classes of samples are computed first in the following manner:

$$\mathbf{C}_0 = \frac{1}{|C_0|} \sum_{\mathbf{x}_i \in C_0} \mathbf{x_i} \qquad (3.4) \qquad\qquad \mathbf{C}_1 = \frac{1}{|C_1|} \sum_{\mathbf{x}_i \in C_1} \mathbf{x_i} \qquad (3.5)$$

The vector $\mathbf{w} = |\mathbf{C}_1 - \mathbf{C}_0|$ is then used as the initial weight vector $(w_1, \dots w_n)$. The method to find the initial weight vector is illustrated

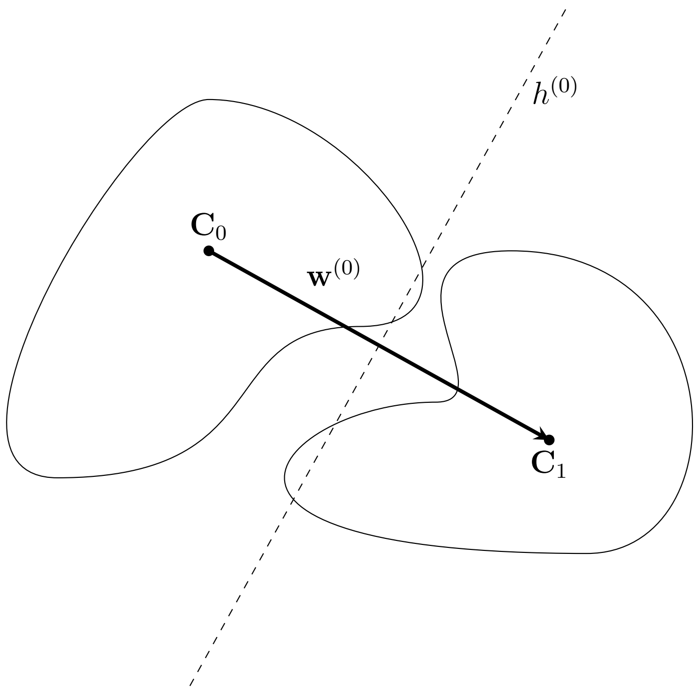

Figure 3.3: Initialization of weights according to Gkanogiannis and Kalamboukis (2009). The centres $(\mathbf{C}_0, \mathbf{C}_1)$ were computed for both classes of samples. The initial weight vector $\mathbf{w}^{(0)}$ was then set to $\mathbf{C}_1 - \mathbf{C}_0$. Finally, the bias w_0 was set to $\mathbf{C}_0 + \frac{|\mathbf{C}_1 - \mathbf{C}_0|}{2}$ which ensures that the initial separating hyperplane $h^{(0)}$ passes halfway between $\mathbf{C}_0$ and $\mathbf{C}_1$. In this setup, a substantial number of samples is already assigned to the correct class, even though a handful of samples are still classified incorrectly.

in Figure 3.3.

As regards the initial bias w_0, the authors suggest using the Scut method, originally developed by Yang (2001). Roughly speaking, it means to go through the training samples one by one, shift the separating hyperplane to pass through the particular sample by choosing an appropriate bias value and compute the 'quality' of such a configuration (e.g. accuracy or F-measure). The best scoring bias value is then used as the actual bias in the initialization.

Apart from the initialization, the authors also suggest a modified stepwise learning rule for the adaptation of the weights, see Figure 3.4. In the traditional approach, the output values are computed for each sample and the weights are subsequently updated in such a way that the separating hyperplane is rotated in the direction of the misclassified samples. The extent to which the weights are modified is co-determined by the parameter ε, which has to be assessed experimentally. The modified variant is both similar and different. It also determines the incorrectly classified samples of both types—false positives (FP) and false negatives (FN)—using the current weights first. After that, the respective centers **FP** and **FN** are computed in the following fashion:

$$\mathbf{FP} = \frac{1}{|FP|} \sum_{x_k \in FP} x_k \tag{3.6}$$

$$\mathbf{FN} = \frac{1}{|FN|} \sum_{x_k \in FN} x_k \tag{3.7}$$

The error vector $\mathbf{e}$ is obtained by subtracting $\mathbf{FP} - \mathbf{FN}$. (Recall that in the traditional approach, this would be obtained as $\sum_{x_k \in FP} x_k - \sum_{x_k \in FN} x_k$ instead.) Afterwards—identically in both approaches—a multiple of the error vector is added to the current weight vector $\mathbf{w}^{(k)}$:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \varepsilon \cdot \mathbf{e}^{(k)} \tag{3.8}$$

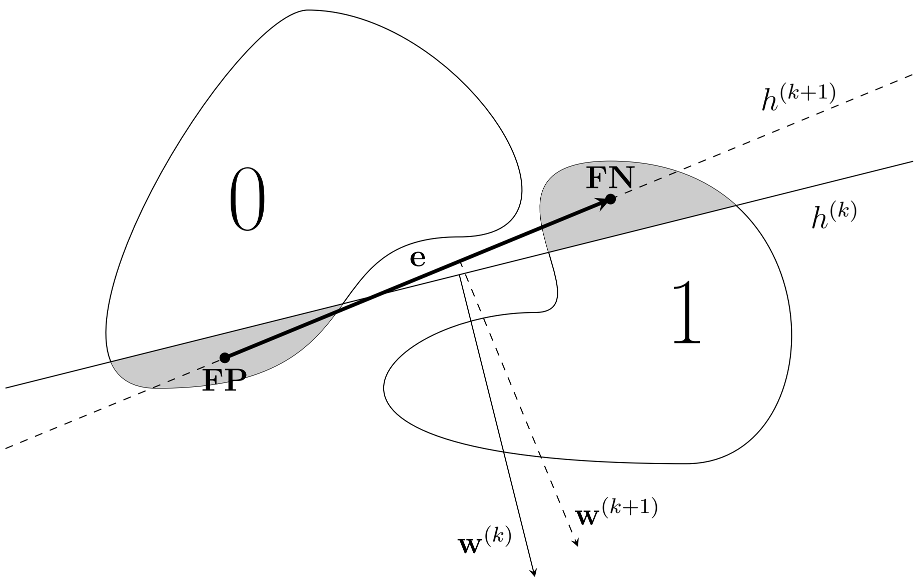

Figure 3.4: Illustration of one step in fast perceptron training according to Gkanogiannis and Kalamboukis (2009). We are trying to find a hyperplane (in this case, line) which separates two classes of examples from each other. The separating line corresponding to the current weight vector $\mathbf{w}^{(k)}$ is labeled as $h^{(k)}$. Apparently, a certain number of samples from the class 0 are incorrectly classified as 1 (“false positives” in the left grey area) and contrarily, other samples from the class 1 are misclassified as 0 (“false negatives” in the right grey area). The centers **FP**, resp. **FN** are computed for all false positives, resp. false negatives, following which the error vector $\mathbf{e}$ can be obtained. Finally, the weights are adapted in such a way that both **FP** and **FN** lie on the new separating hyperplane $h^{(k+1)}$.

The major benefit of the new approach is that the value of ε can be calculated precisely. It is done by adopting the assumption that a hyperplane passing through the points **FP** and **FN** will lead to an improvement in the classification rate. Since these points are the centers of the respective misclassified sets, it follows that about half of the previously misclassified samples will now be classified correctly. (Naturally, new erroneously classified samples may emerge instead.) It can be seen that setting $\varepsilon^{(k)}$ to

$$\varepsilon^{(k)} = -\frac{\mathbf{w}^{(k)} \cdot \mathbf{e}^{(k)}}{||\mathbf{e}^{(k)}||^2} \tag{3.9}$$

and the bias w_0 to

$$w_0 = -\mathbf{w}^{(k+1)} \cdot \mathbf{FP}^{(k)} = -\mathbf{w}^{(k+1)} \cdot \mathbf{FN}^{(k)} \tag{3.10}$$

will accomplish the required task. For a theoretical justification why exactly these values, consult the original article (Gkanogiannis and Kalamboukis, 2009).

3.4 Multi-layer feedforward networks

In the case of a single neuron unit described above, the output value was directly used as the final result indicating the class assigned to the input vector. An alternative option to make use of the output is to pass it as an input of another neuron instead. In this way it is possible to create a network of interconnected neurons, in certain aspects analogous to biological neural networks. Numerous different topologies have been used in practice for different purposes. One of the most widely used type of neural networks are the so-called *multi-layer feedforward networks*. Since these have been employed in part-of-speech tagging as well, they are of a particular importance for this thesis. It is therefore worth saying a few words about their structure, computation and mechanism of learning first.

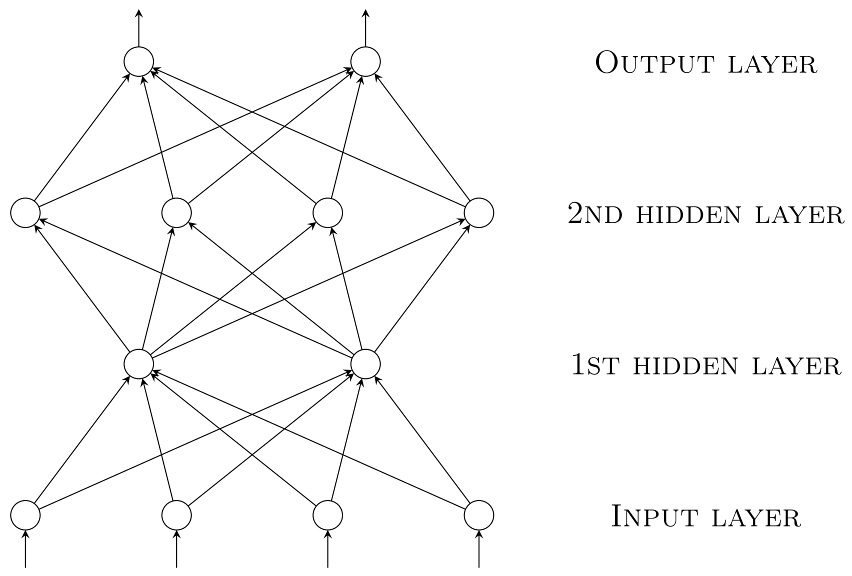

Figure 3.5: An example MLP network with two hidden layers

As the name hints, multi-layer feedforward networks, also known as *multi-layer perceptrons* (MLPs) consist of a certain number of neurons which are split into several disjoint sets, so called *layers*. Every MLP contains an input layer, to which the input vector is fed, and an output layer, which produces the vector of output values. In addition, the network may contain one or more hidden layers, located between the input and the output layer. Each neuron is connected to all the neurons in the adjacent layers: the outputs of the neurons from the previous layer serve as inputs and the output is passed to input of the neurons in the next layer.

As with a single perceptron unit, various activation functions can be used. For reasons that will be shown shortly, it is required that it be differentiable. As the threshold function cannot be used (it is discontinuous in 0), other functions of a similar shape are often employed instead. Two frequently used examples are the logistic sigmoid (3.11) and the hyperbolic tangent (3.12):

$$\sigma(\xi) = \frac{1}{1 + e^{-\lambda\xi}} \tag{3.11}$$

$$\sigma(\xi) = a \tanh(b\xi) \tag{3.12}$$

These functions behave in an analogous way to the threshold function in that they return a value close to the maximum (1 in this case) in for sufficiently large inputs; similarly, a value close to the minimum (0, resp. -1) is returned for inputs low enough. Between these two extremes, the functions grow continuously, their growth rate being determined by the parameter λ (in the logistic sigmoid), resp. b (in the hyperbolic tangent).

3.5 Back-propagation learning algorithm

Let us consider an arbitrary multilayer perceptron network. Let N be the set of all the neurons it consists of, $IN \subseteq N$ the set of the neurons in the input layer and $OUT \subseteq N$ the set of the neurons in the output layer. The individual neurons are indexed by numbers from 1 to $|N|$. The weight of the connection from a neuron i to a neuron j is denoted by w_{ji}, the set of all the weights by W. The notation $i^{\rightarrow}$ refers to the set of all the neurons to which a connection from i exists, and contrarily, $i^{\leftarrow}$ is the set of all the neurons which are connected to the input of i. Let $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{d_1}), \ldots, (\mathbf{x}_p, \mathbf{d_p})\}$ be the set of labeled training samples, where $\mathbf{x}_k = (x_{k1}, \ldots, x_{kn}) \in \mathbb{R}^{|IN|}$ is the k-th input vector and $\mathbf{d}_k = (d_{k1}, \ldots, d_{km}) \in \mathbb{R}^{|OUT|}$ the desired output vector. Let further $\mathbf{X}_k = (x_{k0}, \ldots, x_{kn}) = (1, x_{k1}, \ldots, x_{kn})$ for each $k = 1, \ldots, p$.

At the beginning of the training, weights of each neuron are initialized to random values close to zero. In every learning step, the network is activated with each training input $\mathbf{x}_k$; the outputs are propagated to the output layer, producing an output vector $\mathbf{y}_k = (y_{k1}, \ldots, y_{km})$.

Once all the output vectors $\mathbf{y}_k$ have been obtained, they need to be compared to the desired output vectors $\mathbf{d}_k$ to determine how well the network does and in what way the weights should be modified to improve the performance. To do so, we define the squared error function $E(\mathbf{w})$ of the network's weight vector as

$$E(\mathbf{w}) = \sum_{k=1}^{p} E_k(\mathbf{w}) \tag{3.13}$$

where $E_k(\mathbf{w})$ is the partial error function of the network for the k-th training sample defined as

$$E_k(\mathbf{w}) = \frac{1}{2} \sum_{j \in OUT} (y_{kj} - d_{kj})^2. \tag{3.14}$$

The more training samples generated incorrect outputs, the higher the value of the squared error function is, and vice versa. If the network produces correct outputs for all the samples, the squared error is zero. The problem of finding the best weights for a network is therefore equivalent to the problem of finding the global minimum of the function E. One way to find the global minimum is to compute the gradient $\nabla E(\mathbf{w}) = (\frac{\partial E}{\partial w_{ji}})_{w_{ji} \in W}$ in the point $\mathbf{w}$, corresponding to the current vector of all the weights w_{ji} in the network. The gradient is a vector which determines the direction of the fastest growth of the error function in the specified point, as well as the steepness of the slope in the respective direction. If the gradient is subtracted from the current weight vector, we obtain a new vector which is likely to lie 'below' the original one in terms of the value of E. Therefore, the vector $\mathbf{w}$ is updated in the t-th iteration ($t = 1, 2, \ldots$) as follows:

$$\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} - \varepsilon(t) \cdot \nabla E(\mathbf{w}^{(t-1)} \tag{3.15}$$

The parameter ε determines the learning rate much like in the case of the basic perceptron learning algorithm. Here it specifies the coefficient by which the gradient vector is multiplied prior to its subtraction. Its value can change in every iteration and is usually set to values between 0 and 1.

What remains to be done now is to find the value of $\nabla E(\mathbf{w})$ pertaining to the given weight vector $\mathbf{w}$. As was shown a while ago, it is a vector of partial derivatives $(\frac{\partial E}{\partial w_{ji}})_{w_{ji} \in W}$. Each derivative $\frac{\partial E}{\partial w_{ji}}$ for a particular weight w_{ji} can be computed as

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}. \tag{3.16}$$

Using the chain rule, each term $\frac{\partial E_k}{\partial w_{ji}}$ can be rewritten as

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \frac{\partial y_j}{\partial \xi_j} \frac{\partial \xi_j}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i. \tag{3.17}$$

The value y_i can be obtained directly from the network, so two factors of the product in 3.17 remain to be calculated. Starting with the easier part, the derivative $\sigma_j'(\xi_j)$ depends on the activation function σ_j that the corresponding neuron uses. In the case of the two most common functions, i.e. the logistic sigmoid and the hyperbolic tangent, their respective derivatives yield

$$\begin{aligned} \sigma_j'(\xi_j) &= \left(\frac{1}{1+e^{-\lambda_j \xi_j}}\right)' = \frac{\lambda_j e^{-\lambda_j \xi_j}}{(1+e^{-\lambda_j \xi_j})^2} \\ &= \lambda_j \frac{1}{1+e^{-\lambda_j \xi_j}} \frac{e^{-\lambda_j \xi_j}}{1+e^{-\lambda_j \xi_j}} \\ &= \lambda_j y_j (1-y_j) \end{aligned} \tag{3.18}$$

and

$$\sigma_j'(\xi_j) = (a \cdot \tanh(b \cdot \xi_j))' = \frac{b}{a}(a-y_j)(a+y_j). \tag{3.19}$$

The value of the factor $\frac{\partial E_k}{\partial y_j}$ is defined differently depending whether the neuron j lies in the output layer or not. For output neurons, it is computed directly as the difference of the actual and desired output:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_j \tag{3.20}$$

For neurons in the remaining layers, it is defined as

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma'_r(\xi_r) \cdot w_{rj}, \tag{3.21}$$

where r ranges over all the neurons to which a connection from j exists, here denoted by $j^{\rightarrow}$.

It can be seen that the errors of the output neurons are computed first. Following this, the error is used in computation of the weights of their inputs and subsequently the errors of the neurons in the previous layers are computed. In this manner, the error propagates through the layers to the input layer, modifying the weights along the way. For this reason, the learning algorithm is commonly called the *backpropagation algorithm*.

The algorithm does not guarantee to find the global minimum because the hill climbing technique can cause the weight vector to get stuck in numerous local minima of the error function. To overcome this issue, several modifications are commonly made to the basic backpropagation algorithm. First of all, the chance of getting stuck in a local minimum grows higher as the learning rate ε approaches zero; an appropriate choice of the value of ε is therefore very important for the network's learning ability. Secondly, a *momentum* can boost the performance sometimes. When updating the weight vector in each iteration, not only is the ε-multiple of the gradient subtracted from it, but the vector is also shifted in the direction of the previous correction:

$$\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} - \varepsilon \cdot \nabla E(\mathbf{w}^{(t-1)}) + \alpha \cdot (\mathbf{w}^{(t-1)} - \mathbf{w}^{(t-2)}) \tag{3.22}$$

The size of the momentum is determined by the parameter α, $0 < \alpha \leq 1$. The benefit of employing a non-zero momentum is in that it prevents the weight vector from moving in a zigzag way too much, aiming to the minimum in a straighter way.

Another mechanism that can be included in the trainer is the *weight decay*. It is basically a way to prevent weight from growing too large in order to avoid overfitting. The weight decay rate λ is a number between 0 and 1 determining how fast the weights will 'decay'. The weight vector $\mathbf{w}^{(t)}$ is computed in every iteration as

$$\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} - \varepsilon \cdot \nabla E(\mathbf{w}^{(t-1)}) - \varepsilon\lambda\mathbf{w}^{(t-1)}. \tag{3.23}$$

Now that the essential theoretical framework has been built, we can move on to some practical applications. In the following chapter, a historical overview will be shown which describes particular examples of neural networks that have been used to solve the part-of-speech tagging problem.

4 Historical overview

4.1 Nakamura and Shikano (1989)

One of the earliest attempts to employ neural networks in word category prediction was made by Nakamura and Shikano (1989). In their paper, the authors presented a tool for estimating the most likely part-of-speech tags based on the left context of a chosen size. Such a tool was not a tagger in the usual sense: rather than disambiguating among the possible tags for each word, it learned the probability distributions of individual n-gram tag sequences and stored them in a compact way – the trained network. The performance of this approach was comparable to that of previously used statistical methods; moreover, the number of free parameters did not increase exponentially with growing length of context, a notable improvement in comparison to the statistical approach.

The network Nakamura and Shikano used was a 4-layered perceptron network trained via a slightly modified version of the backpropagation algorithm. Its basic constituent unit was so-called *bigram network*, which approximated the probability distribution of tag-to-tag transitions: given a particular tag on the input (represented by setting the corresponding input neuron to 1), the network generated a vector of numbers, each of them higher or lower according to the rate at which the corresponding tag occured after the input tag in the training data.

The subsequent cluster analysis of values of the neurons of the lower hidden layer, when run on the individual word categories, shown that the network displays similar behaviour for categories that form linguistically meaningful groups (e.g. forms of the verb *be*, subjective pronouns, categories which can come before nouns etc.). The ability to extract various additional linguistic information from texts has been observed numerous times in multiple types of neural networks—let us mention self-organizing semantic maps by Ritter and Kohonen (1989) as a notable example—and demonstrates the strength of this approach.

4.2 Schmid (1994)

Drawing on the work by Nakamura and Shikano (1989) mentioned in the previous section, Schmid (1994) constructed the first notable part-of-speech tagger based on neural networks, named NetTagger.

The network used in NetTagger is a classical MLP network, reproduced here in figure 4.1. The input layer collects various information about the current token and its context. These are subsequently propagated through several hidden layers to the output layer. Each neuron of the output layer represents one part-of-speech tag in the particular tagset. From the output neurons, the one with the highest output value is returned as the result, suggesting the most likely tag for the current token.

Let l, resp. r be the size of the left, resp. right context, i.e. the number of words preceding, resp. following the current token that are taken into consideration. Let further T be the total number of tags in given tagset. Then the input layer consists of neurons $in_{ij}, -l \leq i \leq r, 1 \leq j \leq T$. For $i \geq 0$, each neuron in_{ij} is initialized to the value $P(tag_j|word_i)$, i.e. the lexical part-of-speech probability of the corresponding word. For instance, if the word to the right of the current word is `book` and the tag `V` has the index 5, the neuron in_{+15} will be fed with the value $P(\texttt{V}|\texttt{book})$. The lexical probabilities are estimated using the Maximum Likelihood Principle, i.e. by dividing the number of co-occurrences of $word_i$ with tag_j divided by total number of occurences of $word_i$ in the training corpus.

For $i < 0$, the network output for the word w_i has already been computed in one of the previous runs. Therefore, the output values are used instead of the lexical probabilities.

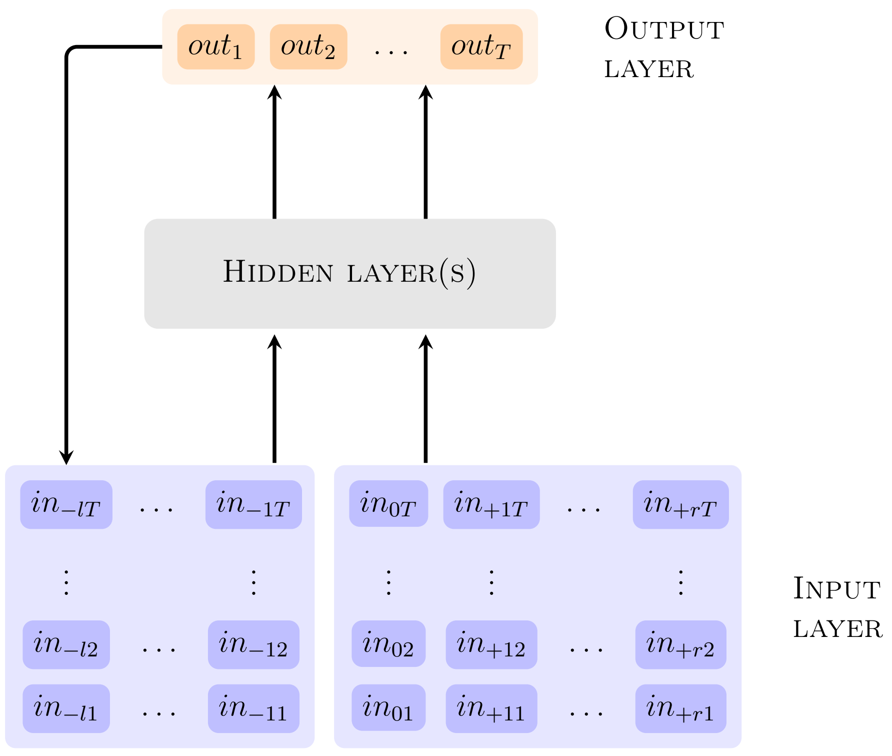

Figure 4.1: Topology of the MLP network used in Schmid's NetTagger

The network is trained on a corpus of manually tagged sentences. Each sentence is converted into a set of training samples representing one word each. The input vector is generated in exactly the same way as in the active phase (i.e. lexical probabilities of the words within context). The desired output is 0 for all output neurons except the correct one, whose expected output is set to 1.

The reccurence brought about by re-using the outputs pertaining to previous words is reflected in the adaptive phase too. Instead of feeding the input neurons representing the left context directly with the network output (which is far from correct in the early learning phase), a weighted average of the desired output and the actual output is used instead.

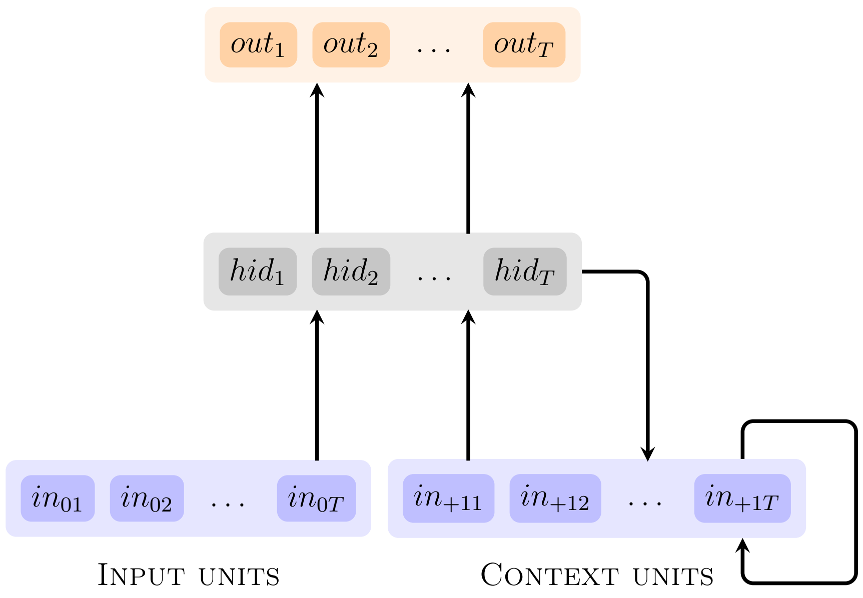

Figure 4.2: Topology of one of the MLP networks used by Marques and Lopes (1996)

4.3 Marques and Lopes (1996)

Another early attempt to make use of neural networks in part-of-speech tagging was made by Marques and Lopes (1996) for Portuguese. In many aspects it resembled the one described by Schmid two years earlier (see above). As in the previous cases, it was also based on multilayer perceptrons. A total of three topologies were tested, all of them employing a context of length 1 – the following word.

In the first topology, only input and output layer were used. Each output neuron corresponded to one particular tag. The input layer was divided into two blocks, the first one representing the word under consideration, and the second one the word to its right. The input neurons in each block represented probabilities of individual tags for the corresponding word. In the second topology, the network was enlarged by adding an extra hidden layer between the input and output layer; the number of these units was unspecified in the article.

Finally, the third topology used was based on Elman (1990). In this case, each neuron in the context block was recurrently connected to itself via an identity link, i.e. the weight was fixed to 1. In addition, each neuron in the hidden layer was connected via an identity link to one neuron in the context block. This recurrence, not dissimilar to the one used by Schmid, introduces a notion of (short-time) memory to the network. This topology is sketched in Figure 4.2.

All three topologies were tested with several in several configurations, using either external resources to generate probability vectors, or computing these from the training corpus. Perhaps a bit surprisingly, the simple network without a hidden layer systematically achieved better scores than either of the two topologies that contained a hidden layer.

4.4 Pérez-Ortiz and Forcada (2001)

The method proposed by Pérez-Ortiz and Forcada (2001) is different from the former approaches in that it does not require a manually disambiguated corpus for training. Instead, it learns from an ambiguously tagged corpus, i.e. corpus in which each word is assigned to a set of one or more part-of-speech tags that are applicable for the particular word form (regardless of the context). Such a corpus can be easily created from a raw corpus e.g. by running a morphological analyser for each word. The sets of possible tags are also referred to as *ambiguity classes*. Since the number of ambiguity classes in a typical corpus is substantially lower compared to the number of different word forms, converting words to ambiguity classes and throwing away any other information can be a good way to simplify the complex task of part-of-speech tagging. This is exactly what the first step of the proposed method looks like.

Once the sentences have been transformed into sequences of ambiguity classes, a two-phase training of the tagger begins. An Elman network is used, i.e. network containing recurrent identity links from the hidden layer to a part of the input layer. The network is trained first to predict the ambiguity class of the next word given the current ambiguity class and a left context, remembered in a context unit. When the training is complete, the network is activated for each word by passing its ambiguity class to the input and the hidden state vector is saved. The hidden state vector of each word is subsequently used for training another perceptron (probably intended to denote a MLP), the purpose of which is to choose the correct tag for the word f positions to the left (i.e. the decision about a word's tag is postponed to a moment when f following words have been read as well). Each part-of-speech tag has assigned its own output neuron; its desired output value during the training is non-zero (e.g. one) if the tag is contained in the ambiguity class and zero otherwise.

This approach was shown to achieve the accuracy of about 54.5%, significantly worse if compared to supervised methods learning from disambiguated corpora, but having an important advantage in that the training data can be obtained very easily, without need to expend considerable resources to create a manually tagged training corpus.

4.5 Other taggers drawing on Schmid

The comparative success of Schmid's NetTagger inspired a number of authors to develop taggers based on the same principle, more or less modified. Apart from Marques and Lopes (1996), which was already described in the previous section, it is worth mentioning at least the following ones:

4.5.1 Ma and Isahara (1997)

Ma and Isahara (1997) proposed a tagger for Thai which is able to work with a variable length of context. For specified lengths of the left and right context l and r (respectively), an MLP pretty much same as in NetTagger is constructed; in addition, further analogous networks are built which differ only in the length of context, which becomes shorter in every subsequent network. All networks are subseqently trained independently with the same data (respecting the length of context required). A network's output is interpreted as a tag t_i if the corresponding output neuron out_i returns the value 1 and all the other output neurons return 0. If this is not the case, the network's output is defined to be *unknown*.

In the active phase, the data to be tagged are transformed into input vectors and passed to the input of each network. Afterwards, the resulting output vectors of individual networks are combined by using either of the logic elements AND or OR. The AND operator produces the tag t_i if this was the result returned by every network in the system, *unknown* otherwise. OR, on the other hand, returns the tag provided by the network which works with the longest context whose output was not *unknown*. If no such network exists, *unknown* is returned instead.

4.5.2 Olde et al. (1999)

The tagger described in Olde et al. (1999) is a module which constitutes a part of a bigger automated tutoring system, AutoTutor. Two topological variants of MLP were tested during its development. As usually, each output neuron $out_1, \ldots, out_T$ represented one part-of-speech tag. The input layer comprised three blocks corresponding to the word being tagged and its neighbours, each of the size T (cardinality of the tagset). The input vector consisted of estimated probabilities of individual tags for given words, exactly like in NetTagger. The second topology contained four extra input neurons indicating whether the current word starts a sentence or whether it is on the second, third or last position in the sentence. Unlike NetTagger, the network did not incorporate recurrent links of any kind.

4.5.3 Poel et al. (2008)

A 2008 paper by Poel et al. (2008) describes another MLP-based part-of-speech tagger for the Dutch language. Again, the input consists of tag probability vectors generated for each word within the context window. Out of the window sizes examined, the best accuracy was achieved for sizes 3-2, 3-3 and 4-3 (the numbers specify the length of the left and right context, respectively). The best scoring sizes of the hidden layer were 250 and 370. An important change to the methods presented so far was made in handling unknown words. Instead of constructing a suffix guesser like in Schmid (1994), each candidate tag was simply assigned an equal probability. From the total of 72 tags in the tagset, the 9 which occured most frequently with unknown words (and covered about 98% of these) were chosen as candidates. These tags were identified by a 10-fold cross validation on the training set.

4.6 Collins (2002)

We have seen earlier that generative models compute the score (i.e. log joint probability) of given pair w_1^n, t_1^n as the sum of certain parameters generated for each tag in the sequence with respect to the context. In 2.15, two parameters were utilized for each tag: $\log P(w_i|t_i)$ and $\log P(t_i|t_{i-2}^{i-1})$.

This notion can be generalized to be able to take also other kinds of parameters into account. Following Collins (2002), let us define a *history* to be a 4-tuple $\langle t_{i-1}, t_{i-2}, w_1^n, i\rangle$ where t_{i-1}, t_{i-2} stand for the preceding two tags, w_1^n is the sequence of tokens to be tagged and finally i is our current position in the sentence. Let us further define a *feature* $\phi_f : \mathcal{H} \times \mathcal{T} \rightarrow \{0,1\}$ to be an arbitrary indicator function assigning each possible combination of history h and tag t either zero or one. Consider the following examples to get a better idea what a feature may look like:

$$\phi_1(h,t) = \begin{cases} 1 & \text{if } w_i = \texttt{the} \text{ and } t = \texttt{DET} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$\phi_2(h,t) = \begin{cases} 1 & \text{if } t_{i-1} = \texttt{ADJ} \text{ and } t_{i-2} = \texttt{DET} \text{ and } t = \texttt{N} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$\phi_3(h,t) = \begin{cases} 1 & \text{if } w_i \text{ has suffix } \texttt{ive} \text{ and } t = \texttt{ADJ} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

$$\phi_4(h,t) = \begin{cases} 1 & \text{if } w_i \text{ is capitalized and } i > 1 \text{ and } t = \texttt{NP} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

In general, features can make use of any kind of information contained in the context $h = \langle t_{i-1}, t_{i-2}, w_1^n, i\rangle$ including the form of w_i or adjacent words, their prefixes or suffixes, tags assigned to the previous words so far, position in the sentence as well as combinations of all above. What all features have in common is that they indicate a co-occurence of a particular tag and a particular context.

Apart from these so-called *local features*, Collins introduces also *global features* Φ_f, each of which represents the number of contexts for which the corresponding local feature ϕ_f equals 1. This can be expressed through the following equation:

$$\Phi_f(w_1^n, t_1^n) = \sum_{i=1}^{n} \phi_f(h_i, t_i) \tag{4.5}$$

where $h_i = \langle t_{i-1}, t_{i-2}, w_1^n, i\rangle$.

Each feature ϕ_f has associated its own weighting factor α_f, which roughly represents how tightly the particular context information is linked to the feature's tag. Taking the feature ϕ_2 from example 4.2 as an example, the value α_2 will be the higher the more frequent the sequence `DET ADJ N` is in the training corpus and vice versa.

Given a token sequence w_1^n, the respective score for a tag sequence t_1^n can be obtained as the weighted sum of all the features ϕ_f computed at each position of the sequence. That is, for each $i = 1, \ldots, n$ we define the history $h_i = \langle t_{i-1}, t_{i-2}, w_1^n, i\rangle$; subsequently, the values $\phi_f(h_i, t_i)$ are computed for each feature ϕ_f in our feature set. These are then weighted by the corresponding values α_f and summed, yielding the overall score of the tagged sequence:

$$score(w_1^n, t_1^n) = \sum_{i=1}^{n} \sum_{f} \alpha_f \phi_f(h_i, t_i) = \sum_{f} \alpha_f \Phi_f(w_1^n, t_1^n) \tag{4.6}$$

Note that if the feature set only contains tag trigram features (like ϕ_2 in 4.2) and word-tag features (like ϕ_1 in 4.1) and the respective α_f's are set to their corresponding log probabilities, the model is equivalent to the generative model of standard HMMs. Hence it can be seen that this approach has the ability to benefit from all the advantages of the HMM approach; in addition, it can utilize further contextual information that is inaccessible by the classical way.

To tag a sequence w_1^n, it then suffices to consider all (reasonable) tag sequences τ_1^n and take the one with the highest score, i.e.

$$\hat{t}_1^n = \operatorname*{arg\,max}_{\tau_1^n \in \mathcal{T}^n} score(w_1^n, \tau_1^n) = \operatorname*{arg\,max}_{\tau_1^n \in \mathcal{T}^n} \sum_{f} \alpha_f \Phi_f(w_1^n, \tau_1^n) \tag{4.7}$$

The best scoring sequence can be effectively found by the Viterbi algorithm in the same way it is done in HMM taggers.

The values α_f for each feature are computed by running the *averaged perceptron algorithm*, here shown in figure . Initially set to zero, they are used to find the best-scoring tag sequence $\hat{t}_1^n$ for each sentence w_1^n. Every

Notation

Let $\mathcal{F}$ denote the set of all features $\phi_f, f = 1, \ldots, |\mathcal{F}|$ we are working with. Let further $\mathcal{S}$ be the set of training sentences $s_i = (w_1^{n_i}, t_1^{n_i}), i = 1, \ldots, |\mathcal{S}|$. The parameter E will represent the (arbitrarily chosen) number of training epochs. Finally, the notation $\alpha_f^{(e),i}$ will refer to the value of the weight α_f in the e-th epoch immediately after the sentence s_i has been processed.

Algorithm

```
▷ Initialize the vector $\vec{\alpha}^{(0),i} = (\alpha_1^{(0)}, \ldots, \alpha_{|\mathcal{F}|}^{(0)})$ to zeros.
▷ For $e$ in $1, \ldots, E$:
    ⋆ Repeat for each tagged sentence $s_i = (w_1^{n_i}, t_1^{n_i})$ in the training corpus until $\hat{t}_1^{n_i} = t_1^{n_i}$:
        ◇ Run the Viterbi algorithm to find the best scoring sequence $\hat{t}_1^{n_i}$ which satisfies the following equality, as defined above:
              $$\hat{t}_1^{n_i} = \underset{\tau_1^{n_i} \in \mathcal{T}^{n_i}}{\arg\max} \sum_f \alpha_f^{(e-1),i} \Phi_f(w_1^{n_i}, \tau_1^{n_i}) \qquad (4.8)$$
        ◇ If $\hat{t}_1^{n_i} \neq t_1^{n_i}$, update every weight $\alpha_f$ in the following fashion:
              $$\alpha_f^{(e),i} = \alpha_f^{(e-1),i} + \Phi_f(w_1^{n_i}, t_1^{n_i}) - \Phi_f(w_1^{n_i}, \hat{t}_1^{n_i}) \qquad (4.9)$$
▷ For each $f = 1, \ldots, |\mathcal{F}|$ compute
      $$\alpha_f = \frac{1}{E \cdot |\mathcal{S}|} \sum_{e=1}^{E} \sum_{i=1}^{|\mathcal{S}|} \alpha_f^{(e),i} \qquad (4.10)$$
▷ Return the weight vector $\vec{\alpha} = (\alpha_1, \ldots, \alpha_{|\mathcal{F}|})$.
```

Figure 4.3: Averaged perceptron algorithm used in Collins (2002)

time $\hat{t}_1^n$ differs from the actual tag sequence t_1^n, weights corresponding to the features holding true for $\hat{t}_1^n$ are decreased; at the same time, we increase weights of the features present in t_1^n. This is done repeatedly until $\hat{t}_1^n$ is the correct tag sequence. The whole process is done in several epochs whose number is specified as an input parameter E. After the training is complete, the values of individual α_f's obtained in each iteration are averaged and these average values are returned as the output.

Collins's tagger does not use neural networks in the strict sense. We have seen that rather than computing an error vector and rotating the separating hyperplane accordingly, the weights α are simply repeatedly increased or decreased by 1 until a correct configuration is found. The principle is nonetheless very close to the perceptron algorithm, as after all also the name *averaged perceptron* suggests. This algorithm is of particular interest to us since, as we will see, one of the methods we propose in chapter 5 draws on Collins significantly.

4.7 Janicki (2004)

The tagger constructed by Janicki (2004) for Polish differs from those mentioned so far in that in assigns tags on the basis of the word form itself, irrespective of the words which come before or after it. Moreover, only the last 7 characters of each word are taken into consideration. Since the tagger is designed to handle alphabetical characters only, each position may contain one of the 35 letters of the Polish alphabet or a space (prepended in front of words shorter than 7 letters). The input layer therefore consists of seven blocks containing 36 units each. When a word is being transformed into a binary input vector, one unit in each block is set to 1 according to the letter at the corresponding position, the others are set to 0. The output layer contains one unit for each tag as usually; the tag whose unit gives the highest output is then returned as the result. The network contains no hidden layers.

To speed up the learning process, the number of neurons in the input layer was reduced to lower values, which were subsequently tested for performance. The size of 50 turned out to achieve an accuracy only slightly worse compared to the non-reduced input layer. Unfortunately, the author does not specify the criterion by which the inputs were chosen into the reduced network.

It turned out that a significant improvement can be achieved by using two separate networks, one for words shorter than six characters and the other for words containing six or more characters. The reason for this distinction is the fact that short words are more likely to be irregular or following slightly different morphological rules than the long ones. With this configuration the tagger achieved an accuracy of about 98%.

Naturally, Janicki's method has its limitations. It is based on the assumption that the part of speech is more or less fully determined by the word form itself. This is possible for languages with rich morphology like Polish in which relatively little part-of-speech ambiguity exists. While Polish makes it possible to distinguish words like *praca* 'a work' vs. *pracować* 'to work' right away, in English it is essential to take account of the neighbouring words as well, for example by checking whether it is preceded by a determiner, or rather by the infinitive particle *to*.

5 Proposed tagger

During the development of the tagger we are presenting in this text, various different methods were experimented with and evaluated in terms of their performance. As is the case with all part-of-speech tagging techniques, two factors had to be taken into consideration: accuracy, i.e. the percentage of tokens in the test data which were assigned the correct tag, and time complexity, i.e. how much time is needed to train the tagger, how long the tagging itself takes and how fast the required time grows as the training and test sets get bigger.

5.1 Method 1: recurrent MLP network

The first method draws on the ideas of Schmid (1994). The core part of the tagger is a set of multilayer perceptron networks, the topology of which is shown in figure 4.1. Each network functions as a binary classifier choosing one out of two candidate tags. The output layer contains two neurons corresponding to the two tags in question. Upon activation, the tag whose neuron gives higher output is chosen as the result and assigned to the corresponding word.

The input layer consists of three blocks of neurons which represent the preceding word, the following word (we will refer to these blocks as the *left context block* and *right context block*, respectively) and finally the word to be tagged (referred to as the *input word block* in the following text). The left and right context blocks both contain T neurons where T is the size of the tagset. The input word block consists of two neurons which represent the two tags between which we want to disambiguate.

The tagging of a sentence $w_1 \dots w_n$ goes from the end to beginning as follows. For each word w_i, its left and right neighbour w_{i-1} and w_{i+1} (respectively) are obtained. Supposing that w_i is ambiguous between tags t_k and t_l, we pick the corresponding network and feed

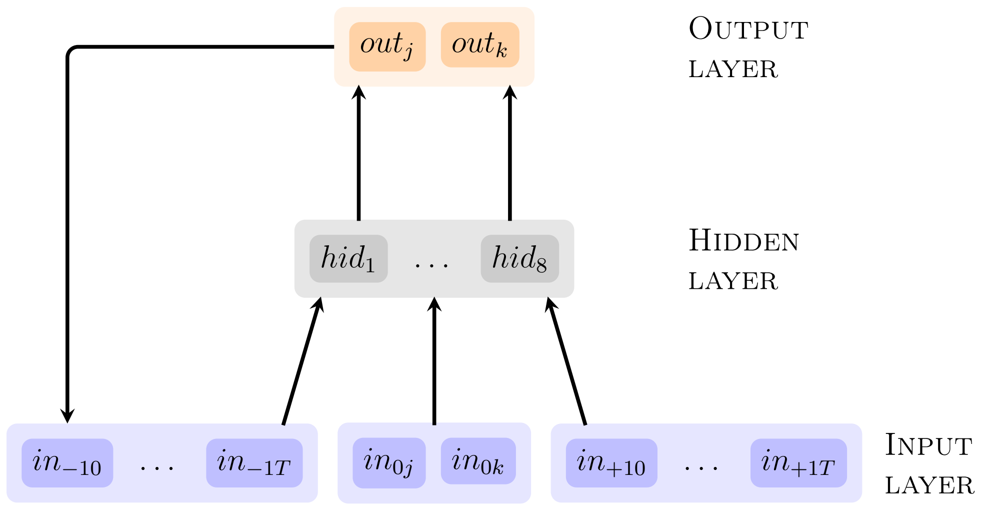

Figure 5.1: Topology of the MLP network used in method 1. The tagger uses a whole system of such networks, each functioning as a binary classifier deciding between two tags. This particular network makes a choice between the tags t_j and t_k. Note that the recurrent link connecting the output layer with the left context block is only relevant in the active phase; it is not utilized during the training.

the estimated probabilities $P(t_k|w_i), P(t_k|w_j)$ into the neurons of the input word block. Similarly, the probabilities $P(t_m|w_{i-1}), m = 1, \ldots, T$ are passed to the input of the neurons in the left context block. The right context block is a recurrent unit which is fed with output values obtained in the previous run, i.e. for the word to the right of the current one. If the following word was for example ambiguous between t_k and t_l and the corresponding network produced output values y_k and y_l, the neurons representing these tags in the right context block will be passed these values to the input, all the other inputs being set to 0.

Each network also includes a single hidden layer, which contains eight neurons. Among the numbers which were tested—from a few up to several hundred—this turned out to provide best classification accuracy. The hidden neurons use the hyperbolic tangent as their activation function, since it yields better results compared to the logistic sigmoid.

Naturally, not all words are ambiguous between exactly two part-of-speech categories. Some are virtually unambiguous (e.g. *the*, *where*), some can be, on the other hand, assigned to a higher number of different tags (e.g. *round*, which can in different contexts function as a noun, adjective, verb, adverb or preposition). Moreover, if a word to be tagged does not appear in the training corpus, we typically want the guesser (based on suffixes in our case) to generate a high recall/low precision set of candidate tags, which can best be accomplished if more tags are included rather than less. For these reasons, we need to be able to convert the multi-class classification problem into such a form that the networks described above can be used for solving it.

The solution for a candidate tag set of an arbitrary cardinality is following: if only one candidate tag is present, it is returned as the result. In the next iteration (i.e. when the preceding word is being processed), the value 1 is passed to the input of the corresponding neuron in the right context block, all the other ones being set to 0. In the case of two or more candidate tags, all subsets of cardinality

2 are generated. For each subset, the corresponding network is found (if it exists) and activated with the appropriate input vector. For each tag, the total number of 'victories' is counted. The tag which won most times—let us call it t_{max} for now—is returned as the result. If there is a draw between two or more tags, the default tag, i.e. the most common one in the training corpus, is used instead. In the next iteration, the input neuron in the right context block corresponding to this winner is set to 1. Each neuron of this block representing one of the other candidate tags—say, t_k—is set to the output value out_k for this tag divided by the output out_{max} for the winning tag as computed by the network deciding between t_k and t_{max}. If such a network does not exist, the neuron for t_k from the right context block is set to 0 instead.

A special attention needs to be paid to situations in which sentence boundaries lie within the context, that is when processing the first or the last token in a sentence. Since the first (last) word does not have a left (right) context, the respective context block cannot be provided with an input generated in the conventional way. A very simple solution has been adopted instead: whenever reaching beyond sentence boundaries, all the neurons in the corresponding context block are set to 0.

The training of the tagger consists of two phases. First, a statistical model is computed from the training corpus, which is subsequently used for training the networks. The statistics computed in the first phase include particularly the tag probabilities for individual words $P(t|w)$. These are sometimes estimated using the method of maximum likelihood, i.e. by dividing the number of occurences of the word w tagged as t in the training corpus by the total number of occurences of w. In our tagger, we use a Laplace estimator instead as it provided marginally better results. The probabilities $P(t|w)$ are estimated in the following manner:

$$P(t|w) = \frac{C(w,t)+1}{C(w)+B} \tag{5.1}$$

where $C(w,t)$ stands for the number of occurences of the word w tagged as t in the training corpus, $C(w)$ is the total number of occurences of w and B is (in this case) the maximum number of tags any word can take, based on the data provided.

Since the training corpus does not (and cannot) contain all words that exist in the respective language, a separate method has to be employed in estimating the tag probabilities for out-of-vocabulary items. A common method is to rely on word endings, which are in many languages good indicators of a word's part-of-speech. Taking all the words with the same ending and averaging their tag probability distributions can therefore be used as an approximation of the word's actual probability distribution.

As there is no universal rule which length of the suffix predicts the part of speech best, suffixes of length up to 10 are used for building the suffixes guesser. For each suffix s in the training corpus of length from 1 to 10, the values $C(s,t)$ and $C(s)$ are determined, representing the number of words ending in s tagged as t and the total number of words ending in s, respectively. Using the Laplace estimator again, the probability of s being tagged as t is computed for each suffix s and tag t:

$$\tilde{P}(t|w) = \frac{C(s,t)+1}{C(s)+B} \tag{5.2}$$

Since an unknown word w ending in the suffix $s = l_1 \dots l_{|s|}$ also ends in the suffixes $s' = l_2 \dots l_{|s|}$, $s'' = l_3 \dots l_{|s|}$ etc., the dilemma arises as to which suffix to use to obtain a probability distribution. This problem is typically overcome by so-called *smoothing*, a technique which combines the probability distribution of a suffix with those of all its sub-suffixes and returns a single new probability distribution. When an unknown word w appears in the data to be tagged, the longest suffix of w that is present in the pre-computed suffix dictionary is found and its smoothed probability distribution is returned, which is ready to be used in the neural networks.

The smoothed probability for a suffix $s = l_1 \dots l_{|s|}$ is calculated recursively as follows:

$$P(t|l_i \dots l_{|s|}) = \frac{\tilde{P}(t|l_i \dots l_{|s|}) + \theta \cdot P(t|l_{i+1} \dots l_{|s|})}{1 + \theta} \tag{5.3}$$

Starting from the whole suffix s, the formula incorporates probability distributions derived from ever shorter suffixes one by one. In the end, it reaches the stage when the probability $P(t|\varepsilon)$ is needed to evaluate the right-hand side of the formula. This is simply put to be equal to the probability of the tag t itself:

$$P(t|\varepsilon) = P(t) \tag{5.4}$$

It remains to explain the meaning of the parameter θ which appears in equation 5.3. Taking a value between 0 and 1, it functions as a weighting factor that determines how much individual sub-suffixes of a suffix contribute to the resulting probability distribution. If $\theta = 0$, only the longest suffix itself is taken account of. On the other hand, if $\theta = 1$ is the case, all the sub-suffixes participate equally on the result, which is a plain average of the respective probability distributions. We chose to compute the value by taking the sample variance of tag probabilities as proposed in Brants (2000), i.e.

$$\theta = \frac{1}{T-1} \sum_{i=1}^{T} [P(t_i) - \bar{P}]^2, \tag{5.5}$$

where T denotes the cardinality of the tag set and $\bar{P}$ is the mean tag probability defined as

$$\bar{P} = \frac{1}{T} \sum_{i=1}^{T} P(t_i). \tag{5.6}$$

In fact, not all the words in the training corpus are used in construction of the suffix guesser. It is a common feature observed across the world's languages that the most frequent words display different inflectional and derivational patterns than the infrequent ones. Apparently, the information that the form *was* is a verb in the past tense will be hardly of any help in the suffix guesser since there are no other past tense forms ending in *-was*, *-as* or even *-s* in English. For this reason, we decided to leave out the common words from training the guesser. Every word that appears in the training corpus more than three times is considered as common.

Now that the probabilities are computed, they can be used in network training. The networks are trained with a batch algorithm, which means that the training samples (input vectors together with their desired outputs) must be generated beforehand for each network. This is accomplished in the following way:

1. Taking one word w_i in the training corpus at a time, find the corresponding tag probability distribution as computed in the statistical model. Do the same for the words to its left and right, w_{i-1} and w_{i+1} (respectively). If either context does not exist (in the case of the first or last word in a sentence), the probability of each tag is set to 0.

2. Let $t_{i_1}, \ldots, t_{i_m}$ be the tags whose probability is greater than zero for w_i. Out of these, let t_{i_k} be the tag with which w_i is tagged in the corpus. Generate all the pairs of tags containing t_{i_k}, i.e.

$$\{t_{i_1}, t_{i_k}\}, \ldots, \{t_{i_{k-1}}, t_{i_k}\}, \{t_{i_k}, t_{i_{k+1}}\}, \ldots, \{t_{i_k}, t_{i_m}\}.$$

   For each of these pairs a neural network has to be constructed which functions as a binary classifier deciding between the two tags. Build all the networks that have not been created in one of the previous iterations. The input layer will contain $2T + 2$ neurons, two for the current word and T for each of its neighbours (T is again the total size of the tagset).

3. Generate a training sample for each network. The left and right context blocks of the input vector will always contain the whole tag probability vector for w_{i-1} and w_{i+1} (respectively), i.e.

   $$(P(t_1|w_{i-1}), \dots, P(t_T|w_{i-1}))$$

   and

   $$(P(t_1|w_{i+1}), \dots, P(t_T|w_{i+1})).$$

   The input word block will differ among the individual networks. If a network decides between, say, t_{i_j} and t_{i_k}, the two input neurons will be set to $P(t_{i_j}|w_i)$ and $P(t_{i_k}|w_i)$. The desired output will be 1 for the neuron representing t_{i_k} and 0 for the other output neuron.

Once the training samples have been generated, they are used for training the networks. Ten passes over each training set are done. The learning rate ε has been set to 0.01, the momentum to 0.1 and the rate of weight decay to 0.01. All these number have been determined experimentally, providing the best accuracy and keeping the time necessary to learn on an acceptable level.

5.2 Method 2: feature-based perceptron

The second method is inspired by Collins (2002) and his averaged perceptron algorithm. It works with a substantially higher number of context features than the method proposed in the previous section. The words are converted into vectors in a multidimensional vector space, which are subsequently classified using binary perceptron classifiers, determining their most likely part-of-speech tag.

The feature set used in this method is shown in Figure 5.2. It is a modification of Collins's feature set, itself drawing on Ratnaparkhi (1996) who used a similar feature set in his maximum entropy tagger.

Feature set

For each word `<W>` in training corpus:
▲ w_i = `<W>`
▲ w_{i-1} = `<W>`
▲ w_{i-2} = `<W>`
▲ w_{i+1} = `<W>`

For each two-word sequence `<W1> <W2>` in training corpus:
▲ w_{i-2}^{i-1} = `<W1> <W2>`
▲ w_{i+1}^{i+2} = `<W1> <W2>`

For each tag `<T>` in training corpus:
▲ t_{i-1} = `<T>`
▲ t_{i-2} = `<T>`

For each sequence of two tags `<T1> <T2>` in training corpus:
▲ t_{i-2}^{i-1} = `<T1> <T2>`

For each suffix of length ≤ 6 in training corpus:
▲ w_i has suffix `<S>`

Other features:
▲ w_i contains at least one hyphen
▲ w_i contains at least one digit
▲ w_i starts with a capital letter

Figure 5.2: The feature set used in the second method

The tagging of a sentence $w_1, \dots, w_n$ goes as follows. Starting from the first word, it is checked which context features hold true and which do not. A feature vector of 0's and 1's is generated which contains ones precisely on those positions which correspond to the present features. For example, if ϕ_3 is by definition 1 iff $w_{i-1} = \texttt{the}$, the third component of the vector will contain 1 if the word previous to the current one was indeed *the*; it will be zero otherwise.

Once the feature vector has been computed, it is passed as an input to a set of neurons. Each neuron represents one part-of-speech tag, for which it serves as a yes/no classifier, determining whether the word should be tagged with the respective part-of-speech or not. To speed up the tagging process, only those neurons are activated which correspond to the word's possible tags. If the word was not present in the training corpus and it begins with a capital letter, this is changed to lowercase and tag candidates are checked for once again. If this still does not help, all the neurons are activated one after another.

In the optimal case, exactly one neuron will give the output 1, all the other producing 0; the winner's tag would then be returned as the result. Nevertheless, nothing prevents more neurons from returning 1's; also, the situation may arise when all the neurons yield 0's. For this reason, it is the inner potentials, i.e. weighted sums of the inputs, that are compared. The tag whose neuron's inner potential was the highest upon activation is then used for tagging the word in question.

It is worth noting that some features make reference to the previous (but not following) tags, see Figure 5.2. The tags assigned to preceding words by the mechanism described above are therefore immediately used for finding the values of these features in words that follow. No probabilistic computation takes place, each word is assigned to precisely one fixed tag at the moment when it is being processed. If a word has been misclassified, it can naturally have an impact on the correctness of the tags of words which follow. On the other hand, the system makes use of a number of other features which do not rely on the previous tags, so a single mistake does not necessarily mean producing nonsense in the whole rest of the sentence.

Before the system can be used for tagging, the neurons have to be trained first. The idea is as straightforward as it gets: for each word w_i tagged as t_i in the training corpus, check which features hold true and generate the appropriate feature vector. This will be used as an input vector of each of the T neurons. The desired output value will differ among individual neurons: it will be 1 for the neuron which makes a decision about the tag t_i, 0 for every other neuron.

The number of features increases rapidly as the training corpus grows bigger; it is roughly comparable to the corpus size in terms of tokens. When learning from a 1,000,000-word corpus, a neuron with several milion inputs has to be trained for each part-of-speech tag. Clearly, this is a time demanding task which cannot be accomplished easily using the traditional perceptron algorithm without any modifications. The training algorithm described in what follows has a significantly shorter running time compared to the perceptron, yet it needs to be said that for bigger corpora the required time still gets unacceptably long.

The training algorithm used is the fast perceptron trainer of the authors Gkanogiannis and Kalamboukis (2009), already described in section 3.3. During the initialization of the weight vector, the samples are split into two classes according to the desired output (0 or 1) and their centers $\mathbf{C}_0$ and $\mathbf{C}_1$ are computed. The initial weight vector $(w_1, \ldots, w_F)$ is obtained by subtracting $\mathbf{C}_1 - \mathbf{C}_0$. Afterwards, the value w_0 of the bias has to be determined. Using the Scut method as proposed in the original article does not seem adequate here due to its extreme time complexity—for 1,000,000 training samples it would mean 10^{12} activations of a neuron with several million inputs. Instead, we chose such a bias value that the separating hyperplane pass through the midpoint of the line segment $\mathbf{C}_0\mathbf{C}_1$, i.e. $\mathbf{C_0} + \frac{\mathbf{C}_1-\mathbf{C}_0}{2}$.

In the original article, the weights are adapted in every iteration in the following way: using the current weight vector, compute the neuron's output for each sample. Compare the output with the desired output for the sample; it these two are not equal, label the sample as either false negative (desired output 1, actual 0) or false positive (desired output 0, actual 1). Find the respective centers $\mathbf{FN}, \mathbf{FP}$ for the sets of false negative and false positive samples and modify the weights by adding an ε-multiple of $\mathbf{e} = \mathbf{FP} - \mathbf{FN}$ to the current weight vector such that both $\mathbf{FN}$ and $\mathbf{FP}$ lie on the new separating hyperplane.

The time needed to perform a single weight modification step largely depends on the total number of misclassified samples: the more of them, the longer it takes to compute the centers $\mathbf{FN}$ and $\mathbf{FP}$. To speed up the training, we made a minor modification to the algorithm: instead of taking all the misclassified samples, only 100 of them are randomly chosen and used in finding the new weight vector.

The weights are iteratively modified until at least one of the stopping criteria is met. Naturally, the training stops when the neuron classifies all the training samples correctly; moreover, it there is only one misclassified sample, the training is finished as well. The second stopping criterion limits the number of iterations to prevent an overly long training without any progress: if 30 iterations passed without beating the best accuracy reached so far, the training is terminated and the weights that gave the best accuracy are kept.

5.3 Baseline method

To get a better idea as to how well the proposed methods perform, a third method was used for comparison. It is a so-called backoff tagger which iself incorporates three n-gram taggers—trigram, bigram and unigram—together with a primitive suffix guesser. For each word w_i in given sentence w_1^n, the probability $P(t_i|w_i, t_{i-1}^{i-2})$ is determined for each t_i based on a training corpus provided beforehand; the tag with highest probability is then assigned to w_i. If the probabilities are zero for all candidate t_i's, the tag is computed using the bigram probability $P(t_i|w_i, t_{i-1})$. In the case that this step fails too, the unigram probability $P(t_i|w_i)$ is used. If this is still not enough due to the absence of w_i from the training data, the three-letter suffix s_i^3 is taken from w_i and the tag t_i with maximal $P(t_i|s_i^3)$ is assigned. Finally, it it was not possible to determine t_i by any of the previous methods, the most frequent tag from the training corpus is chosen for t_i.

5.4 Technical details of implementation

The tagger was implemented in the programming language Python, making use of certain functionality provided by the Python library `nltk` (Bird, 2006). The core of the system is formed by three modules named `tag1`, `tag2` and `tag3` which correspond to the individual tagging methods described above. Each module contains the class `Tagger` implementing the interface `nltk.TaggerI`. This essentially means it contains a method `tag()` which takes a list of tokens representing a bare sentence and returns a list of tuples `(word, tag)`. One particular benefit of this choice is the availability of the method `evaluate()` which, given a list of tagged sentences, readily returns the tagger's accuracy as a number between 0 and 1.

To build and train the multilayer perceptron networks in method 1, we used the Python library `pybrain` (Schaul et al., 2010) which has been designed for this purpose. It makes it possible and very intuitive to build various neural networks with a desired number of layers and neurons with a chosen activation function. In addition to this, the training of the network can also be performed in a simple way, allowing one to choose the values of parameters like the learning rate ε, momentum and weight decay rate. Since `pybrain` is typically not a built-in part of Python installations, it has to be installed beforehand for the tagger to work properly. The same applies to the package `nltk`.

The backoff tagger from method 3 was implemented using the classes `TrigramTagger`, `BigramTagger`, `UnigramTagger`, `AffixTagger` and `DefaultTagger`, all available from the package `nltk`.

The program contains an interface for training the tagger and evaluation of the individual methods for the Brown corpus. The module is named `test` and is used in the following way:

```
test.py [-l] [-s] [-o] -m<METHOD> <train_size> <test_size>
```

There are three parameters that have to be passed to the script as command-line arguments. The parameter `-m<METHOD>` selects that is to be used for training and tagging, the options being `-m1`, `-m2` or `-m3`. `<train_size>` specifies the number of sentences from the beginning of the corpus that will be used for training. Similarly, `<test_size>` determines the number of sentences from the *end* of the corpus used for evaluation.

There are three more optional command-line arguments: `-s` allows to export the tagger after training into a file, from which it can be subsequently loaded by invoking the option `-l`. Finally, calling the script with `-o` generates an output file to which the test sentences tagged by the chosen method are written in a human-readable form.

So, for example, if we want to train the tagger using method 2 (feature-based perceptron) with the first 1462 sentences from the Brown corpus, evaluate it on the last 10128 sentences and save the tagger into a file, it is accomplished through the following call:

```
$ python test.py -s -m2 1462 10128
```

6 Results and discussion

6.1 Results

To test the performance of the individual methods, we used the Brown corpus, a manually tagged English corpus of 1,161,192 words which is freely available from the package `nltk`. From the total of 57,340 sentences in the corpus, the last 10,128 sentences containing 161,194 words were used for testing. From the rest several training sets of different sizes were built to determine how the tagging accuracy changes depending on the amount of data provided. Training sets of five sizes were generated: 435 sentences (10,018 words), 1462 sentences (31,623 words), 4,600 sentences (100,012 words), 14,900 sentences (316,235 words) and finally the whole rest of the corpus containing 47,212 sentences (999,998 words). For each experiment, the accuracy was computed as the ratio of correctly tagged tokens to the total number of tokens. The accuracies achieved by the three methods for each training set are shown in tables 6.1 through 6.3. In addition, a graph plotting the results is provided in figure 6.1.

Looking on the results, several observations can be made from them. Firstly, both methods 1 and 2 performed better than method 3 (the only exception being the smallest training set), which is an indication that the proposed algorithms actually work. This is definitely good news,

# of sents	# of words	accuracy	training time
435	10018	77.75%	50s
1462	31623	86.24%	3m 9s
4600	100012	90.81%	12m 40s
14900	316235	93.27%	57m 40s
47212	999998	94.93%	5h 46m 19s

Table 6.1: Results for method 1 (recurrent MLP network)

# of sents	# of words	accuracy	training time
435	10018	82.41%	6m 31s
1462	31623	89.59%	46m 24s
4600	100012	92.96%	4h 15m 55s
14900	316235	?	?
47212	999998	?	?

Table 6.2: Results for method 2 (feature-based perceptron)

# of sents	# of words	accuracy	training time
435	10018	78.10%	1s
1462	31623	85.40%	5s
4600	100012	88.83%	13s
14900	316235	91.08%	44s
47212	999998	92.98%	2m 24s

Table 6.3: Results for method 3 (backoff tagger)

confirming that the proposed methods make sense. Secondly, the recurrent MLP network approach has shorter training times compared to the feature-based perceptron; in fact, the time needed to train the neurons in the latter method with the two largest training sets was so long that we were not able to determine the respective accuracies. Thirdly, for the three smallest training sets, the feature-based perceptron did constantly better compared to the recurrent MLP network at the cost of substantially longer training times. The absolutely best accuracy achieved by any of the methods was 94.93%.

6.2 Discussion

We have seen in the previous section that the feature-based perceptron consistently provided the best results in terms of accuracy, as far as we were able to tell. However, the extremely long training times pre-

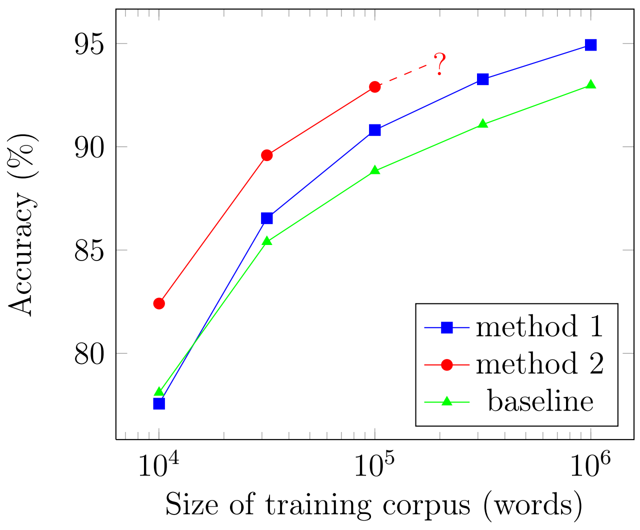

Figure 6.1: Accuracies achieved by the individual methods for each training set

vented us from measuring the performace for training sets containing hundreds of thousands or millions of tokens. This was in spite of the optimization techniques that are made use of during the training to accelerate the learning (see section 5.2). To determine the accuracy for larger corpora, the training could be performed on a supercomputer or alternatively—and better—further optimization could be made to the learning algorithm. It is likely that many features (perhaps a majority of them) in our feature set do not have a significant impact on the probabilities of individual tags. If we were able to detect these 'junk' features for each tag and keep only those whose presence is strongly correlated with it—whether in a negative or positive way—the number of inputs could decrease drastically, making the time necessary for training significantly shorter.

It is fair to say here that the best accuracy we were able to reach, 94.93%, is by far not the best accuracy that could be achieved by any existing tagger. For example, when the largest training set of 999,998 words was processed by HunPos (Halácsy et al., 2007), a faster and open-source reimplementation of the successful tagger TnT (Brants, 2000), the accuracy on the same test set was 96.13%; also the training time was just a fraction of the time needed by either of our methods using neural learning.

The only language for which experiments were done was English. The Brown corpus that was used for evaluation contains 40 different part-of-speech tags, a comparatively small tagset among tagset which are used in corpora. It is clear from the architecture of our tagger that larger tagsets likely increase the training time proportionally. This should be taken into consideration when tagging e.g. Slavic or Finno-Ugric languages, whose tags often incorporate numerous morphological features such as case, number, tense etc. This naturally leads to greater tagsets which can contain as much as several hundred or even more than a thousand different tags.

7 Conclusion

In this work we proposed an automated system for part-of-speech tagging which benefits from the principles of neural learning. From numerous methods experimented with in the course of the development, the two most promising ones were chosen for a more in-depth exploration and were included in the tagger. The first method, using a system of recurrent multilayer perceptron networks, achieves an overall accuracy of 94.93% when trained on the first ca. million tokens from the Brown corpus and tested on the rest. The other method, based on simple threshold neurons processing a high number of context features was computationally too complex to converge in a reasonable time for the same training set; however, for all the smaller subcorpora provided for training, it systematically achieved a significantly better accuracy compared to the first method. As far as we were able to tell, both methods performed better than a conventional trigram-bigram-unigram backoff tagger that was used as a baseline.

Time complexity is generally a major drawback of neural algorithms, though we have seen that certain tricks can be made to accelerate the learning. Even though the training times needed were rather long compared to e.g. HMM taggers, at least the first method definitely lies within the limits of acceptability. The second method could potentially be sped up by selecting the relevant context features for each tag, decreasing the amount of information needed to be processed for each word drastically.

Notwithstanding all the imperfections that our tagger can be reproached with, we believe this work gives an illustrative insight into an intriguing class of lesser-used part-of-speech tagging techniques. Despite the criticism of neural learning that can be heard from time to time, we are convinced that a tagger with an aptly chosen neural architecture can make a tough competition to the more conventional techniques used nowadays. There is beyound all doubt a room for further experiments leading to an improvement in performance, both in terms of accuracy and time complexity.

Bibliography

Bird, S. (2006). NLTK: the natural language toolkit, *Proceedings of the COLING/ACL on Interactive presentation sessions*, Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 69–72.

Brants, T. (2000). TnT: a statistical part-of-speech tagger, *Proceedings of the sixth conference on Applied natural language processing*, Association for Computational Linguistics, pp. 224–231.

Collins, M. (2002). Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms, *Proceedings of the ACL-02 conference on Empirical methods in natural language processing*, Association for Computational Linguistics, pp. 1–8.

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

Gkanogiannis, A. and Kalamboukis, T. (2009). A modified and fast perceptron learning rule and its use for tag recommendations in social bookmarking systems, *ECML PKDD Discovery Challenge 2009 (DC09)* p. 71.

Halácsy, P., Kornai, A. and Oravecz, C. (2007). HunPos: an open source trigram tagger, *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, Association for Computational Linguistics, pp. 209–212.

Janicki, A. (2004). Application of neural networks for POS tagging and intonation control in speech synthesis for Polish, *Soft Computing and Intelligent Systems (SCIS 2004)* .

Jurafsky, D. and Martin, J. H. (2009). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, 2nd edn, Pearson Education.

Ma, Q. and Isahara, H. (1997). Part-of-speech tagging of Thai corpus with the logically combined neural networks, *Proceedings of the Natural Language Processing Pacific Rim Symposium*, pp. 537–540.

Marques, N. C. and Lopes, G. P. (1996). A neural network approach to part-of-speech tagging, *Proceedings of the Second Workshop on Computational Processing of Written and Spoken Portuguese*, pp. 1–9.

Nakamura, M. and Shikano, M. (1989). A study of English word category prediction based on neutral networks, *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, IEEE, pp. 731–734.

Olde, B. A., Hoeffner, J., Chipman, P., Graesser, A. C. and Group, T. R. (1999). A connectionist model for part of speech tagging., *FLAIRS Conference*, pp. 172–176.

Poel, M., Boschman, E. and op den Akker, R. (2008). A neural network based Dutch part of speech tagger.

Pérez-Ortiz, J. A. and Forcada, M. L. (2001). Part-of-speech tagging with recurrent neural networks, *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, Vol. 3, IEEE, pp. 1588–1592.

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

Ritter, H. and Kohonen, T. (1989). Self-organizing semantic maps, *Biological cybernetics* **61**(4): 241–254.

Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T. and Schmidhuber, J. (2010). PyBrain, *The Journal of Machine Learning Research* **11**: 743–746.

Schmid, H. (1994). Part-of-speech tagging with neural networks, *Proceedings of the 15th conference on Computational linguistics*, Association for Computational Linguistics, pp. 172–176.

Šíma, J. and Neruda, R. (1996). *Teoretické otázky neuronových sítí*, Matfyzpress.
URL: *http://www2.cs.cas.cz/~sima/kniha.pdf*

Yang, Y. (2001). A study of thresholding strategies for text categorization, *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 137–145.