View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Groningen

University of Groningen

Machine learning of phonotactics

Tjong-Kim-Sang, Erik Fajoen

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
1998

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Tjong-Kim-Sang, E. F. (1998). Machine learning of phonotactics. Groningen: s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 12-11-2019

Machine Learning of Phonotactics

Erik F. Tjong Kim Sang

The research described in this thesis has been made possible by a grant from the Dutch Research School in Logic (formerly *Nederlands Netwerk voor Taal, Logica en Informatie*).

GRONINGEN DISSERTATIONS IN LINGUISTICS 26

ISSN 0928-0030

RIJKSUNIVERSITEIT GRONINGEN

Machine Learning of Phonotactics

Proefschrift

ter verkrijging van het doctoraat in de
Letteren
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus dr. D.F.J. Bosscher,
in het openbaar te verdedigen op
maandag 19 oktober 1998
om 16.15 uur

door

Erik Fajoen Tjong-Kim-Sang

geboren op 6 december 1966
te Utrecht

Promotor: Prof. Dr. Ir. J.A. Nerbonne

Acknowledgements

It is customary in my research group to add to one's PhD thesis a section with reflections about life during the production of this book and acknowledgements to people who have been friendly to the author during this period. I will make no exception to this custom. Over the past eight years and twenty one days I have, with some interruptions, worked on a research project about the application of machine learning methods in natural language processing. Over these years I have learned a lot but I have also found out that there is a lot left to be learned. My average writing speed of nineteen words per day has resulted in the PhD thesis on this subject which I am submitting right now.

I want to thank Frans Zwarts and Jan de Vuyst for starting this research project and giving me the opportunity to work in it. I am also grateful to the organization Nederlands Netwerk voor Taal, Logica en Informatie (currently Dutch Research School in Logic), which has supplied the grant that has made my research project possible. John Nerbonne has played a major role in the fact that this the project has resulted in a PhD thesis. He has been responsible for the project supervision including the initial suggestion of the final thesis topic. John also created the possibility for me to stay longer at the department of Alfa-informatica than the time offered by my original project. Thanks!

My colleagues of the department Alfa-informatica of the University of Groningen in The Netherlands were responsible for creating a pleasant working environment during the five years I was allowed to spend with them. For that reason I want to thank Bert Bos, Dicky Gilbers, Erik Kleyn, Garry Wiersema, George Welling, Gertjan van Noord, Gosse Bouma, Harry Gaylord, Joop Houtman, Mark-Jan Nederhof, Mettina Veenstra, Peter Blok, Petra Smit, Rob Koeling, Shoji Yoshikawa and Yvonne Vogelenzang. I am specially indebted to Mettina and Bert. I was fortunate enough to spent most of my Groningen years working in the same room as them. They played a major role in my development during these years, both on a scientific and a personal level. Thanks!

After Groningen I have spent three years at the department of Linguistics of Uppsala University in Sweden. I want to thank Anna Sågvall Hein for offering me the opportunity to work in what I have regarded as a demanding but instructive environment. During my years in Uppsala I have been fortunate enough to work with the following people: Annelie Borg-Bishop, Bengt Dahlqvist, Gunilla Fredriksson, Hong Liang Qiao, Jon Brewer, Jörg Tiedemann, Klas Prytz, Lars Borin, Leif-Jöran Olsson, Malgorzata Stys, Mariana Damova, Mark Lee, Mats Dahllöf, Olga Wedbjer Rambell, Per Starbäck and Torbjörn Lager. I want to reserve a special word of gratitude to my former students in Uppsala which as a group have been a great example of motivation towards their studies and general friendliness both inside and outside the classroom. Thanks!

No thesis can be finalized without being read by and approved by a thesis committee. I want to thank Anton Nijholt, Ger de Haan and Nicolay Petkov for being part of my thesis committee and for reading and commenting the thesis.

In the past eight years I have met many nice people in different circumstances both in Groningen, Uppsala and at other locations. Their kindness has had a positive influence on my general well-behavior which has contributed to the completion of this thesis. I want to thank all of them.

Groningen, August 21, 1998,

Erik Tjong Kim Sang

Contents

Acknowledgements **5**

Contents **7**

1 Introduction **11**

1 Theoretical background 12
1.1 Problem description 12
1.2 Data representation 13
1.3 Positive and negative learning examples 14
1.4 Innate knowledge 14
2 Experiment setup 15
2.1 Goals 16
2.2 The training and test data 17
2.3 Data complexity 18
2.4 The linguistic initialization model 20
2.5 Elementary statistics 22
3 Related work 23
3.1 The work by Ellison 23
3.2 The work by Daelemans et al. 24
3.3 Other work 25

2 Statistical Learning **27**

1 Markov models 27
1.1 General description of Markov models 27
1.2 The forward procedure 29
1.3 The Viterbi algorithm 31
2 Hidden Markov Models 34
2.1 General description of Hidden Markov Models 34
2.2 The extended forward procedure 35
2.3 The extended Viterbi algorithm 37
2.4 Learning in a Hidden Markov Model 39
2.5 Using Hidden Markov Models in practice 44

3 Initial Experiments . . . 45
3.1 A test experiment . . . 45
3.2 Orthographic data with random initialization . . . 46
3.3 Orthographic data with linguistic initialization . . . 48
3.4 Discussion . . . 50
4 Experiments with bigram HMMs . . . 52
4.1 General bigram HMM experiment set-up . . . 53
4.2 Orthographic data with random initialization . . . 53
4.3 Orthographic data with linguistic initialization . . . 55
4.4 Phonetic data with random initialization . . . 58
4.5 Phonetic data with linguistic initialization . . . 60
5 Concluding remarks . . . 63

3 Connectionist Learning **65**
1 Feed-forward networks . . . 65
1.1 General description of feed-forward networks . . . 66
1.2 Learning in a feed-forward network . . . 68
1.3 Representing non-numeric data in a neural network . . . 71
2 The Simple Recurrent Network (SRN) . . . 72
2.1 General description of SRNs . . . 73
2.2 Learning in SRNs . . . 75
2.3 Using SRNs for language experiments . . . 76
3 Experiments with SRNs . . . 77
3.1 General experiment set-up . . . 78
3.2 Finding network parameters with restricted data . . . 80
3.3 Orthographic data with random initialization . . . 83
3.4 Orthographic data with linguistic initialization . . . 85
4 Discovering the problem . . . 87
4.1 The influence of the number of valid successors of a string . . . 87
4.2 Can we scale up the Cleeremans et al. experiment? . . . 88
4.3 A possible solution: IT-SRNs . . . 90
4.4 Experiments with IT-SRNs . . . 91
5 Concluding remarks . . . 93

4 Rule-based Learning **95**
1 Introduction to Rule-based Learning . . . 95
1.1 Positive versus negative examples . . . 96
1.2 The expected output of the learning method . . . 97
1.3 Available symbolic learning methods . . . 97
2 Inductive Logic Programming . . . 99
2.1 Introduction to Inductive Logic Programming . . . 99
2.2 The background knowledge and the hypotheses . . . 102
2.3 Deriving hypotheses . . . 105
2.4 The hypothesis models and grammar theory . . . 107

3 Experiments with Inductive Logic Programming . . . 110
3.1 General experiment setup . . . 110
3.2 Handling orthographic and phonetic data . . . 112
3.3 Adding extra linguistic constraints . . . 113
3.4 Discussion . . . 116
4 Alternative rule-based models . . . 118
4.1 Extending the model . . . 118
4.2 Deriving extended hypotheses . . . 120
4.3 Experiments with the extended model . . . 122
4.4 Compressing the models . . . 124
5 Concluding Remarks . . . 128

5 Concluding remarks **131**
1 Experiment results . . . 131
2 Recent related work . . . 134
3 Future work . . . 136

Bibliography **139**

Samenvatting **145**

Chapter 1

Introduction

This thesis contains a study of the application of machine learning methods to natural language. We will use the learning methods Hidden Markov Models, Simple Recurrent Networks and Inductive Logic Programming for automatically building models for the structure of monosyllabic words. These three learning algorithms have been chosen as representatives for three main machine learning paradigms: statistical learning, connectionist learning and rule-based learning. The language data to which they will be applied has been limited to monosyllabic words in order to keep down the complexity of the learning problem. We will work with Dutch language data but we expect that the results of this study would have been the same if it had been done with another related language.

The study will focus on three questions. First, we want to know which of the three learning methods generates the best model for monosyllabic words. Second, we are interested in finding out what the influence of data representation is on the performance of the learning algorithms and the models they produce. Third, we would like to see if the learning processes are able to create better models when they are equipped with basic initial knowledge, so-called innate knowledge.

This book contains five chapters. The first chapter will describe the problem. In the second chapter we will introduce the statistical learning method Hidden Markov Models and present the results of the experiments we have performed with this method. In the third and fourth chapters we will do the same for the connectionist method Simple Recurrent Networks and the rule-based method Inductive Logic Programming respectively. The final chapter contains a comparison of the results of all experiments and some concluding remarks.

1 Theoretical background

In this section we will give a description of the learning problem we want to tackle: building phonotactic models for monosyllabic words. We will also introduce some theoretical issues related to this problem: the importance of a good representation of data, the influence of negative training examples and the influence of innate knowledge.

1.1 Problem description

Why is *pand* a possible English word while *padn* is impossible in English? Why is *mloda* a possible Polish word but not a possible Dutch word? For giving the answers to these questions one has to know the syllable structures which are allowed in English, Polish and Dutch. Native speakers of English can tell you that *pand* is a possible English word and that *padn* is not. Judging the words does not require that the native speakers have seen them before. They use their knowledge of the structure of English words to make their decision. How did they obtain this knowledge?

In the example we have presented we showed that the possibility that a word exists depends on the structure of the language the word appears in. Certain languages, like Polish, allow *ml* onsets of words but others, like English and Dutch, do not. The structure of words in a language is called the phonotactic structure of a language. Different languages may have different phonotactic structures.

There are two possibilities for entering a language dependent phonotactic structure into a computer program. The first is by making humans examine the language and make them create a list of rules defining the phonotactic structure. This requires a lot of labor which has to be done for all languages. The second possibility is making the program *learn* the phonotactic structure of a language by providing it with language data. People manage to learn phonotactic rules which restrict the phoneme sequences in their language so it might be possible to construct an algorithm that can do the same. If we are able to develop a model for the phonotactic structure learning process we can use the model to analyze the phonotactic structure of many languages.

Both artificial intelligence and psychology offer a wide variety of learning methods: rote learning, induction, learning by making analogies, explanation based learning, statistical learning, genetic learning and connectionist learning. We are not committed to one of these learning methods but we are interested in finding the one that performs best on the problem we are trying to tackle: acquiring phonotactic structure. For our experiments we have chosen three machine learning paradigms: statistical learning, connectionist learning and rule-based learning. We will use learning methods from these three paradigms and compare their performance on our learning problem.

A possible application of these phonotactic models lies in the field of Optical Character Recognition (OCR). OCR software frequently has to make a choice between two or more possible interpretations of a written or printed word, for example between *ball*

and *baII*. The most easy way of solving this problem is by looking up the words in a dictionary and choosing the one which appears in the dictionary. This approach fails when neither one of the words is present in the dictionary. In that case the software should be able to determine the probability that the words exist in the language which is being read. A phonotactic model for the language can be used for this.

This study on phonotactics is also important for our research group because it is our first application of machine learning techniques to natural language processing. The problem chosen is deliberately simple in order to make possible a good understanding of the machine learning techniques. The results of this study will be the basis of future research in even more challenging applications of machine learning to natural language processing.

1.2 Data representation

Every artificial intelligence text book emphasizes the importance of knowledge representation. The way one represents the input data of a problem solving process can make the difference between the process finding a good result or finding no result. A nice example of the usefulness of knowledge representation is the Mutilated Checkerboard Problem presented in (Rich et al. 1991). Here a checkerboard from which two opposite corner squares have been removed, needs to be covered completely with dominoes which occupy two squares each. This problem is unsolvable. This fact can be proven by trying out all possible domino configurations but that will require a lot of work.

The solution of the Mutilated Checkerboard Problem can be found more quickly by changing the representation of the problem and representing the board as a collection of black and white squares. An inspection of the board reveals that it contains 30 white squares and 32 black squares. Each domino must cover exactly one white square and exactly one black square so this problem is unsolvable. One could ask if this more suitable problem representation could have been foreseen. Unfortunately the best way for representing data is dependent on the problem that one wants to solve. There is no algorithmic method for deciding what data representation is the best for what type of problem.

The input data for our learning problem can be represented in several ways. We will take a look at two representation methods. The first one is called the orthographic representation. Here words are represented by the way they are written down, for example: *the sun is shining*. The second way of representing the words is the phonetic way. If we use the phonetic representation then the sentence *the sun is shining* will be represented as [ðəsʌnɪʃainɪŋ]. We do not know which of the two representations will enable the learning processes to generate the best word models. Acceptance decisions of words by humans may be based on the way the words are written but they may also be based on the pronounceability of the words. We are interested in finding out which representation method is most suitable for our learning problem. Therefore we will perform two variants of our learning experiments: one with data in the orthographic

representation and one with the same data in the phonetic representation.

1.3 Positive and negative learning examples

A learning algorithm can receive two types of learning input: positive examples and negative examples. A positive example is an example of something that is correct in the language that has to be acquired and a negative example is an example of something that is incorrect. Together with the examples the algorithm will receive classifications (correct/incorrect).

Gold's landmark paper (Gold 1967) has shown that it is not possible to build a perfect model for a general language that contains an infinite number of strings by only looking at positive examples of the language. The reason for this is that for any set of positive example strings there will be an infinite number of models that can produce these examples. Without negative examples it is not possible to decide which of these models is the correct one. The research result of Gold has consequences for natural language learning. Natural languages are infinite because they contain an infinite number of sentences. This means that according to language learning theory it is not possible to build a perfect model for a natural language by only looking at correct sentences of that language.

With Gold's research results in mind one would predict that children use negative language examples for acquiring natural language. However, research in child language acquisition has found no evidence of children using negative examples while learning their first language (Wexler et al. 1980). Even when children are corrected they will pay little attention to the corrections. Here we have a problem: according to computational learning theory, children need negative examples for learning if they want to be able to learn a natural language. Children do not seem to make use of negative examples and yet they manage to acquire good models for natural languages.

We will approach the acquisition of models for monosyllabic words from the research results in child language acquisition. We will supply our learning methods with positive information only. However mathematical language learning theory predicts that negative examples are required for obtaining good language models. We will assume that negative information can be supplied implicitly. In the next section we will deal with a possible solution for the absence of negative examples in our learning experiments.

1.4 Innate knowledge

There have been a number of attempts to explain the gap between what learning theory states about the necessity of negative examples and what child language acquisition reports about the absence of these negative examples. One proposed explanation assumes that the learners use available semantic information for putting constraints on natural language utterances (Finch 1993). Another explanation suggests that learners acquire reasonably good language models rather than perfect models (probably

approximately correct (PAC) learning (Adriaans 1992)). A third explanation restricts the languages that human learners can acquire to a small subset of the languages which are possible theoretically (Finch 1993).

All three explanations have some cognitive plausibility and could be applicable to our learning problem. The usage of extra semantic information in computational learning experiments has a practical problem: this information is unavailable (Finch 1993). The ideas behind probably approximately correct (PAC) learning are interesting and we will use some of them in our experiments. As in PAC learning we will accept models that perform as well as possible rather than restricting ourselves to perfect models. However unlike some PAC learning algorithms we will rely on the fact that all of our learning examples are correct and we will not use the ORACLE concept mentioned in (Adriaans 1992) because that would imply using negative examples.

We will perform learning experiments in which the set of languages that can be acquired will be restricted. This can be done in practice by enabling the learning algorithm to choose from a small set of models instead of all possible models. This simplifies the task of the learning algorithm. Restricting the set of languages is an approach which is also suggested in human language acquisition theory (Wexler et al. 1980) (Chomsky 1965). Humans are not regarded as being capable of learning all mathematically constructible languages. They can only acquire languages of a smaller set: the natural languages. The restriction to this smaller set is imposed by innate cognitive constraints. Human language learning can be modeled by a system which sets parameters in a general language device in order to change it to a language-specific device.

While it might be necessary to assume extra initial knowledge for the acquisition of a complete language model, one could also try to generate some reasonably good language models without using initial knowledge. The thesis of Steven Finch (Finch 1993) gives some examples of extracting lexical information from positive data without assuming innate language knowledge. We do not know whether such an approach would be successful for our learning problem. We are interested in what gain artificial language learning systems can get from equipping them with initial linguistic knowledge. Therefore we will perform two versions of our experiments: one version without initial knowledge and another in which the learning algorithm starts from basic phonotactic knowledge. The linguistic model that we will use as initial knowledge will be explained in section 2.4.

2 Experiment setup

This section contains the practical issues concerning approaching our phonotactic learning problem. We will start with describing the goals of the experiments we want to perform. After that we will take a look at the format and the complexity of our training and test data. We will continue with examining the linguistic model we will use in our experiments which start with basic phonotactic knowledge. The section will be concluded with a paragraph containing the statistical theory which will be used for

interpreting the experiment results.

2.1 Goals

We will perform experiments with monosyllabic phonotactic data and attempt to derive a phonotactic model from positive examples of the data. The model should be able to decide whether strings are members of the language from which the training data has been taken or not. It can be considered as a black box which takes strings as input and returns *yes* if the string is a possible member of the training language and *no* if it is not. The model might assign more than two evaluation scores to the strings. If that is the case then we will assume that the scores can be ordered and that the comparison of their values with a threshold value will determine whether they should be counted as *yes* or *no*.

Our training algorithms will not receive the complete set of monosyllabic data during the training phase. The consequence of this is that memorizing the training data is insufficient for obtaining a good model. The phonotactic models will have to be able to give a reasonable evaluation of unseen words which might be correct despite the fact that they were not present in the training data. In other words: the models have to be able to generalize.

In order to test the generalization capabilities of the models we will test them with unseen positive data. We will require that after training the models accept all training data so we can skip testing their performance on this data. We will also test the models with incorrect data to make sure that they do not accept a lot of incorrect strings. The two tests will result in two scores: the probability of accepting a string of the unseen positive data and the probability of rejecting a string of the negative data.

While performing the phonotactic model acquisition experiments we will look for the answers to the following questions:

1. What learning algorithm produces the best phonotactic model?

2. What data format results in the best phonotactic model?

3. Does starting from initial knowledge produce better phonotactic models?

We will perform the same experiment with algorithms representing three different machine learning paradigms: Hidden Markov Models (statistical learning), Simple Recurrent Networks (connectionist learning) and Inductive Logic Programming (rule-based learning). All experiments will be done with the same training and test data and under the same conditions to make a comparison of the results fair. It is not possible for us to test every possible learning algorithm so the final comparison might not point to the best algorithm. However it will give an indication to which machine learning paradigm performs best on this problem.

We will compare two different data formats: the orthographic format and phonetic format (see section 1.2). For each experiment we will perform two variants: one with

training and test data in orthographic format and one with the same data in phonetic format. We are interested in finding out which data format will enable the learning algorithms to produce the best phonotactic models.

We would also like to find out whether learning algorithms that are equipped with initial phonotactic knowledge will generate better phonotactic models than algorithms without this knowledge. Therefore we will perform two variants of each learning experiment: one in which the learning algorithms start without any knowledge and one in which they have some initial phonotactic knowledge. The initial model will be derived from a phonological model which will be described in section 2.4.

Thus we will perform twelve variants of a phonotactic acquisition experiment with three learning techniques, two data formats and two initialization types. Care will be taken to perform the experiments under the same conditions so that a final comparison between the results will be fair.

2.2 The training and test data

The learning algorithms will receive a training data set as input and they will use this set for building models for the structure of Dutch monosyllabic words. The models will be able to compute acceptance values for arbitrary strings. They can either accept a string as a possible monosyllabic Dutch word or reject it. A good phonotactic model will accept almost all correct unseen words (positive test data) and reject almost all impossible words (negative test data, also called random data).

The positive data sets have been derived from the CELEX cd-rom (Baayen et al. 1993). From the Dutch Phonology Wordforms directory (dutch/dpw/dpw.cd) we have extracted 6218 monosyllabic word representation pairs.[1] The first element of each pair was the orthographic representation of the word (field Head) and the second the phonetic representation of the word (field PhonolCPA). We have removed 10 words of the list because they were not mentioned in the standard Van Dale dictionary for Dutch (Geerts et al. 1992) (*flute*, *flutes*, *frite*, *Joosts*, *move*, *moves*, *rocks*, *straight*, *switch* and *switcht*). Another three words have been removed because they had been incorrectly classified as monosyllabic words (*racend*, *fakend* and *shakend*). We obtained 6205 unique pairs. The list contained 6177 unique orthographic strings and 5684 unique phonetic strings.

After this we randomly chose 600 pairs from the list. We made sure that these pairs contained neither duplicate orthographic strings nor duplicate phonetic strings. The 600 orthographic strings and the corresponding 600 phonetic strings will be used as test data. The remaining 5577 orthographic words and 5084 phonetic words will be used as training data. The orthographic data contained the twenty six characters of the alphabet plus the quote character ' in order to allow for words as *ski's*. The phonetic data contained 41 different characters.

[1]The monosyllabic words have been selected by removing all lines with hyphenation marks and all lines with empty phonetic representations. After that the fields Head and PhonolCPA were extracted, the upper case characters were converted to lower case and the duplicate pairs were removed.

We have used the character frequencies of the orthographic data for generating 700 random orthographic strings. The generation process has assumed that characters occurred independently of each other. We have transcribed the 700 random strings and thus we have obtained a list of 700 random phonetic strings. From the lists we have removed 60 strings that resembled Dutch strings, 2 strings that had a phonetic representation that occurred earlier in the list and 38 strings from the end of the list. We obtained a list of 600 unique implausible orthographic strings and a list of 600 corresponding unique phonetic strings. We will use these lists as negative test data files for our experiments.

The final operation we performed on the data sets was appending to each string an end-of-word character.[2] This was necessary because in the statistical and the connectionist learning methods the words will be presented to the learning algorithms in a sequence and the algorithms will need some way for determining where the current string ends and a new string begins.

2.3 Data complexity

The performance of the learning algorithm is dependent on how difficult the data is. Acquiring a phonotactic model for a language that consists of the strings *a*, *aa* and *aaa* is much easier than acquiring a model for monosyllabic Dutch words. There are different measures which formalize this intuitive notion of data complexity. In this section we will look at two of these. We will also apply these complexity measures to our data.

The first complexity measure we will examine is called ENTROPY.[3] This is a number which indicates how uncertain the result is of drawing elements from a set. For example we have a set containing two *x*'s, one *y* and one *z* and the probability of drawing an *x* is 50% and the probability of drawing a *y* or a *z* is 25%. The entropy of this experiment is 1.5 (the computation will be performed below). A small entropy means that it is easy to predict the results of the draws and a large entropy means that the prediction is difficult.

The results of draws from a set can be represented with the concept STOCHASTIC VARIABLE. Our example is equivalent to a stochastic variable with the values *x*, *y* and *z* in which the probabilities of the values are 50%, 25% and 25% respectively. We give this stochastic variable the name *W* and call the three values c_1, c_2 and c_3. Then we can compute the entropy H(W) of this variable by using the probabilities P(c_i) of each value and the formula (Charniak 1993):

$$H(W) = - \sum_{c_i} (P(c_i) * log_2(P(c_i)))$$

So H(W) is the negation of the sum of the products of the probability P(c_i) and its $\log_2$ value for all values c_i. If we apply this formula to our example with the probabilities

[2] In chapter 2 we will give a motivation for using a start-of-word character as well.

[3] The complexity measure PERPLEXITY is related to entropy. It will not be not discussed here.

P(x)=0.5, P(y)=0.25 and P(z)=0.25 then we obtain the following computation: H(W) = -(0.5*$\log_2$(0.5) + 0.25*$\log_2$(0.25) + 0.25*$\log_2$(0.25)) = -(-0.5+-0.5+-0.5) = 1.5.

In order to be able to compute the entropy of our data we will regard the words in the positive data as sequences of draws from a set of characters. The probability of each character can be estimated by computing the frequency of the character in the positive data and dividing it by the total number of characters in the data. After having acquired the character probabilities we can compute the entropy of the data. Our positive orthographic data with 6177 words and 41005 characters of which 29 are unique has an entropy of 4.157. Our positive phonetic data with 5684 words and 34430 characters of which 43 are unique has an entropy of 4.294.[4]

A language model that estimates the validity of a word by looking at isolated characters is called a UNIGRAM MODEL. Such a model uses probabilities of characters without looking at their contexts. We will also use models which compute the probability of a character given information about previous characters. The probabilities in these models will say something about the occurrence of pairs of characters and therefore they are called BIGRAM MODELS.

Words in natural languages are not arbitrary sequences of characters. The context of a character has an influence on its probability. For example, in our positive orthographic data the probability that an arbitrary character is a *u* is 3.8% but the probability that a character following a *q* is a *u* is 92%. The task of predicting a character in word is easier when one knows the preceding character. Thus the earlier defined concept of entropy is not very useful for bigram models.

In order to describe the data of a bigram model we need to use a kind of conditional entropy which we will call BIGRAM ENTROPY. The value is computed in a similar way to standard entropy. However instead of character probabilities this computation uses conditional character probabilities: the probability of a certain character c_j given that a specific character c_i precedes it. We have used an adapted version of the formula presented in (Van Alphen 1992) page 96:

$$\text{H(W)} = -\sum_{c_i} P(c_i) \sum_{c_j} (P(c_j|c_i) * log_2(P(c_j|c_i))$$

We have applied this formula to our orthographic and phonetic data and obtained the bigram entropy values of 3.298 and 3.370 respectively.[5] From these values we can conclude that the data is less complex when it is considered as a sequence of character bigrams rather than a sequence of isolated characters. In other words, it is easier to predict a character of a word when one knows the preceding character.

Entropy and bigram entropy give an indication about the complexity of the data. It is unclear how useful these concepts are for predicting the degree of learnability of the data. The general expectation is that data with a large entropy is more difficult to learn

---

[4]In both orthographic and phonetic data each word contains a start-of-word and an end-of-word character. Without these characters the orthographic data has an entropy of 4.255 while the phonetic data has an entropy of 4.551.

[5]The bigram entropy values without the end-of-word characters were 3.463 for the orthographic data and 3.617 for the phonetic data.

than data with a small entropy. However there are counter-examples. The entropy of a language A consisting of strings that contain an arbitrary number of *a*'s is 0 while a language B with strings that contain any combination of *a*'s and *b*'s has an entropy of 1. If we restrict A to strings with a length *n* where *n* must be a prime number then A would probably be more difficult to learn than B. Yet the entropy of A is still 0 which predicts that A is easier to learn than B.

An alternative measure which can be used for determining the complexity of our data is the CHOMSKY HIERARCHY. This is a hierarchy of classes of grammar which can be used for modeling artificial and natural languages. The hierarchy distinguishes one grammar class for finite languages and four grammar classes for infinite languages. The complexity of a language is determined by the place in the hierarchy of the least complex grammar that is capable of producing it.

Our phonotactic data is finite. The longest strings in our data contain nine characters without begin-of-word and en-of-word character. Monosyllabic words with a few more characters might be possible but the existence of an English or Dutch monosyllabic word of twenty or more characters should be ruled out. Both our orthographic and our phonetic data should be placed in the lowest spot in the Chomsky hierarchy: the group of finite languages. This indicates that a model for the data can be acquired by looking at positive data only (Gold 1967).[6]

A problem of using the Chomsky hierarchy for determining the complexity of data is that the differences within each language/grammar class are large. The empty language and the language consisting of prime numbers with less than a million digits are both finite languages. Yet the first is much easier to learn than the second. The Chomsky hierarchy and the related language learning classes put forward by (Gold 1967) give only a rough indication of the learnability of languages (see also (Adriaans 1992) section 2.3.4).

We conclude that the available techniques for determining data complexity do not have an exact answer on the question of how difficult the learning process will be.

2.4 The linguistic initialization model

We will perform two versions of our learning experiments: one that starts with initial phonotactic knowledge and one that start without any initial knowledge. As an initialization model we have chosen the syllable model which is presented in (Gilbers 1992) (see Figure 1.1). This model is a mixture of syllable models by (Cairns and Feinstein 1982) and (Van Zonneveld 1988). Hence it will be called the Cairns and Feinstein model.

The Cairns and Feinstein model is a hierarchical syllable model consisting of a tree with seven leaves. Each leaf can either be empty or contain one phoneme. The appendix leaf may contain two phonemes. Each leaf is restricted to a class of phonemes: in models for Dutch and many other languages the peak may only contain vowels and

[6]Derivation of a perfect model is only guaranteed if one is able to evaluate the complete data set. However we will not provide the complete data set to our learning algorithms.

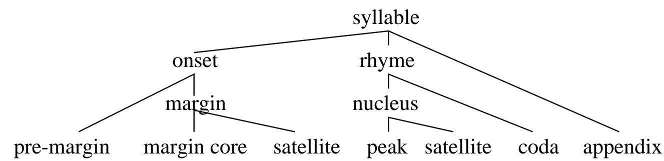


Figure 1.1: The syllable model of Cairns and Feinstein

the other leaves may only contain consonants. The exact phonemes that are allowed in a leaf are language dependent. In the syllable model there are vertical lines between nodes and daughter nodes which are main constituents. A slanting line between two nodes indicates that the daughter node is dependent on the sister node that is a main constituent. A dependent constituent can only be filled if its main constituent is filled. For example, the margin satellite can only contain a phoneme if the margin core contains a phoneme.

This syllable model can be used to explain consonant deletion in child language. For example, the word *stop* fits in the model like *s*:pre-margin, *t*:margin core, *o*:peak and *p*:coda. An alternative structure with *t* in the margin core satellite is prohibited by a constraint for Dutch which states that satellite positions only may contain sonorant consonants (like *l*, *m*, *n* and *r*). The model predicts that a child which has difficulty producing consonant clusters will delete the dependent part in the onset cluster and produce *top*. The alternative *sop* would violate the constraint that dependent constituents, in this case pre-margin, cannot be filled if the corresponding main constituent, here margin, is not filled. Another example is the word *glad* which fits in the model like *g*:margin core, *l*:margin satellite, *a*:peak and *d*:coda (the *g* is prohibited in the pre-margin in Dutch). In this case the model will predict that the child that has problems with producing consonant clusters will delete the *l* rather than the *g*. Both predictions are correct.

In our experiments with initial knowledge we will supply the learning algorithms with the syllable structure presented in Figure 1.1. Two extra constraints will be provided to the algorithms: the fact that the peak may only contain vowels while the other leaves are restricted to consonants. Furthermore the division of phonemes in vowels and consonants will be made available for the learning algorithms. Their task will be to restrict the phonemes in each leaf to those phonemes that are possible in the language described by the learning examples. By doing this they will convert the general Cairns and Feinstein model to a language-specific syllable model.

2.5 Elementary statistics

In this thesis we will need to compare the performances of different learning algorithms on the acquisition of phonotactic models. The performance of the models is defined by two scores: the percentage of accepted correct strings and the percentage of rejected incorrect strings. Both scores have an average and a standard deviation. The question is how we can compare these scores and determine whether one is significantly better than the other.

Two numbers with an average and a standard deviation can be compared with the t-test (Freedman et al. 1991). By using this test we can determine whether the difference between the numbers is caused by a real difference or by a chance error. The test computes a number *t* by dividing the difference of the averages of two numbers we want to compare by the standard error of this difference. The standard error of the difference is equal to the square root of the sum of the squares of the standard errors of the two input numbers. Here are all necessary formulas for comparing two numbers which have averages AVG_x and AVG_y and standard deviations SD_x and SD_y based on measuring them *n* times:

$\text{SE}_x = \text{SD}_x/\sqrt{n-1}$ (standard error of *x*)
$\text{SE}_y = \text{SD}_y/\sqrt{n-1}$ (standard error of *y*)
$\text{SE}_d = \sqrt{SE_x^2 + SE_y^2}$ (standard error of *x*-*y*)
$\text{t} = \frac{AVG_x - AVG_y}{SE_d}$ (t-score)

The result of these computations will be a number *t(n-1)*. This number can be looked up in a *t*-table to find out what the probability is that the difference was caused by a chance error. If the probability is smaller than 5.0% then we will assume that there is a real significant difference between the two numbers.

Example: We have performed two sets of 5 experiments with Hidden Markov Models that process orthographic data: one set was randomly initialized and the other one was initialized with a general version of the Cairns and Feinstein model. In the first set of experiments the HMM needed 108, 87, 81, 48 and 65 training rounds to become stable. In the second experiment set the learning algorithm needed 43, 17, 63, 22 and 47 training rounds. The models with phonotactic initialization seem to train faster and we want to know if the difference is real. Therefore we compute the averages (77.8 and 38.4), the standard deviations (20.3 and 16.9), the standard errors (10.1 and 8.4) the standard error of the difference (13.3) and *t(4)* (3.0). We look up this *t(4)* value in a *t*-table and find out that the probability *p* that the difference between the two number lists was caused by a chance error is between 1.0% and 2.5% ($<$5.0%). We will adopt the notation used in (Van den Bosch 1997) and express this result as *t(4)*=3.0, $p<0.025$. So the difference cannot be explained by a chance error: HMMs with phonotactic initialization train faster than randomly initialized HMMs for this problem.

3 Related work

In this section we will examine literature on the problem of machine learning of phonotactics and related areas. We will start with the work of T. Mark Ellison who has published a thesis and some papers about applying machine learning to phonology. After that we will take a look at the work of Walter Daelemans and his group on using machine learning for building models for hyphenation, syllabification, stress placement and grapheme-to-phoneme conversion. We will conclude with an overview of related work by other authors.

3.1 The work by Ellison

In the thesis (Ellison 1992) Mark Ellison investigates the possibility of using machine learning algorithms for learning phonological features of natural languages. He imposes five constraints on the algorithms:

1. they should work in isolation, that is without help of a human teacher,
2. they should be independent of the way the (phonetic) data has been encoded,
3. they should be language independent,
4. they should generate models which are accessible and
5. they should generate models which are linguistically meaningful.

The goal of Ellison in his thesis is to show that it is possible to create successful machine learning applications which satisfy all five constraints for phonological problems. He evaluates a number of existing learning applications in related domains and concludes that none of them fulfills all five requirements. Many learning systems violate the first constraint because they are supplied with extra knowledge like a vowel-consonant distinction or monosyllabic data. Other systems like statistical and connectionist learning algorithms fail to generate models which are accessible and linguistically meaningful.

Ellison investigates machines learning techniques on three different phonological tasks with positive input data. The first task is the derivation of vowel-consonant distinction models. For this derivation Ellison used an inductive learning technique combined with a model evaluation measure which favored a simple model over more complex models. The learning algorithm represented phonological models as sets of parameters and searched the parameter space for the best model by using the searching technique simulated annealing. Ellison's learning algorithm was applied to data from thirty languages. In all but four languages it divided the phonemes in two groups: the vowels and the consonants. The problems in the four languages were caused by the fact that the program erratically had divided either the consonants or the vowels in two groups based on their positions in the words.

Ellison applied the same learning algorithm to the second and the third learning task. The second task consisted of deriving the sonority hierarchy of natural languages where sonority means the loudness of phonemes. As in the first task the learning algorithm was applied to data from thirty languages. It generated on average three sonority classes per languages and put the vowels in different classes than the consonants in all except one language. The program performed well in separating consonants from vowels but performed less well in building vowel hierarchies and consonant hierarchies.

In the third learning task the algorithm had to derive harmony models. These models contain context constraints on phoneme sequences. The learning algorithm was applied to data from five languages. It discovered correct constraints on vowel sequences for all five languages. Ellison wanted to study vowel harmony and therefore he supplied the learning algorithm with vowel sequences. He argued that this does not mean that the learning algorithm fails his first isolation constraint because it should be considered as part of a larger program in which the vowel-consonant distinction was discovered by the untutored first learning program.

The learning algorithm used by Ellison in his three experiments satisfies his five learning constraints. Its is interesting to compare the experiments we want to perform with these constraints. Only one of our algorithms will satisfy the fourth and the fifth constraint about generating accessible and meaningful results: the rule-based learning method. Neither the statistical nor the connectionist method will generate accessible and meaningful phonotactic models. All our algorithms will be language-independent and use an arbitrary data representation. However they will not satisfy the first untutored constraint because they will process monosyllabic data and will be supplied with linguistic knowledge in the initialized experiments.

3.2 The work by Daelemans et al.

The group of Walter Daelemans has done a lot of work on applying machine learning in natural language processing areas such as hyphenation, syllabification, placement of stress, grapheme-to-phoneme conversion and morphological segmentation. In this work different machine learning algorithms have been applied to several linguistic problems. The algorithms used are inductive memory-based techniques and connectionist techniques. The learning methods that performed best were memory-based techniques which simply stored learning examples rather than building an abstract representation for them.

(Daelemans et al. 1993) describes the application of three learning methods for acquiring the stress pattern in Dutch. Their data contained 4868 monomorphemes for Dutch words. In their first experiment they have applied backpropagation (BP), Analogical Modeling (ANA) and Instance-Based Learning (IBL) for predicting the stress patterns of the words. IBL and BP performed approximately equally well on unseen data while ANA performed slightly less well. IBL and ANA were used in a second experiment with two versions of the data set (one phonetic version). Their

learning methods performed reasonable even for stress patterns which are difficult to predict for state-of-the-art theory. The authors conclude that computational learning methods such as ANA, BP and IBL are an alternative to Principles and Parameters based learning theories.

(Daelemans et al. 1995) presents a study in which machine learning techniques are used for building a models for determining the diminutive suffix of Dutch nouns. The learning method used is the decision tree learning method C4.5 (Quinlan 1993). The model generated by C4.5 performed well: it obtained an error rate of 1.6% on this task and outperformed the theoretical model presented in (Trommelen 1983) which paid special attention to the formation of diminutives. By comparing the results of training C4.5 with different parts of the data the authors have been able to falsify Trommelen's claim that rhyme information of the last syllable of a Dutch noun is sufficient for predicting its diminutive suffix.

(Daelemans et al. 1996) discusses grapheme to phoneme conversion based on data-oriented methods. The authors have used the machine learning technique IG-Tree for building grapheme to phoneme conversion for the languages Dutch, English and French. IG-Tree performed significantly better than a connectionist state-of-the-art solution (NETtalk, Sejnowski et al. 1987) and a theoretical model (Heemskerk et al. 1993). Daelemans and Van den Bosch conclude that IG-Tree has three advantages: it does not require gathering rules, it is reusable and it is accurate.

(Van den Bosch et al. 1996) describes the application of four inductive learning algorithms and one connectionist method on three variants of the problem of dividing words in morphemes. The algorithm that generated the best model for all three tasks was IB1-IG, an inductive algorithm that stores all learning examples and compares unseen data with the stored data while taking into account that some data features are more important than others. The algorithm uses information gain, a concept from information theory, for computing the importance of each data feature for the task. It uses the current character and six characters in its context to decide whether to insert a morpheme boundary or not.

(Van den Bosch et al. 1997) presents a motivation for the good results of lazy learning techniques in the domain of natural language learning. A categorization of language data will contain many small clusters of related elements. Since lazy learning techniques store the complete training data they will have less chance of missing small variations than learning methods which summarize data. Equipped with an information-theoretic-based weight of the data features lazy learning techniques will be even more successful in categorizing the data. The authors present an empirical study with a word pronunciation task to support their claims.

3.3 Other work

In (Wexler et al. 1980) Kenneth Wexler and Peter W. Culicover describe a method for learning transformational grammars describing natural language syntax. Transformational grammars are capable of generating arbitrary type 0 languages (Partee et al. 1993)

and therefore they are neither learnable from positive examples only nor from both positive and negative information (Gold 1967). The learning method of Wexler and Culicover is based on putting restrictions on the transformational grammars that are to be learned. The input of the method consists of positive example pairs *(b,s)* where *b* is a phrase-marker and *s* is the corresponding surface string.

The five transformational grammar restrictions suggested by Wexler and Culicover are: the Binary Principle, the Freezing Principle, the Raising Principle, the Principle of No Bottom Context and the Principle of the Transparency of Untransformable Base Structures (see (Wexler et al. 1980) section 4.2). The authors have proven that the restrictions are necessary for making the grammars learnable. They have also discussed the linguistic plausibility of the restrictions and they have concluded that first one probably is linguistically plausible, the second one is plausible and the third one probably not. The plausibility of the fourth and the fifth restriction have not been discussed.

(Adriaans 1992) describes a rule-based learning method for learning categorial grammar. The learning method has access to an example sentence generator and an oracle which evaluates arbitrary sentences. Adriaans has put restrictions on the generator and the grammatical rules that should be learned. The generator produces sentences in an order based on their complexity and the complexity of the rules is restricted by the assumed complexity of the complete grammar.

Adriaans's algorithm contains four steps. In the first step *s* sentences are generated where *s* is dependent on the expected complexity of the grammar. After that all possible rules explaining the sentences will be extracted. In the third step these rules will be combined and simplified. Finally the validity of the rules will be tested by using them for generating sentences and supplying the sentences to the oracle. Adriaans has been able to prove that his learning system can effectively learn context-free languages when the restrictions on the generator processing and target rule complexity are satisfied.

(Gildea et al. 1996) describes an interesting learning experiment in which an induction algorithm is applied to problem with and without extra domain constraints. The problem was deriving phonological rules for phenomena as flapping, r-deletion and word-final stop devoicing. The rules were represented as deterministic transducers. The learning algorithm without the constraints failed to learn the rules. However equipped with the three constraints Faithfulness (usually an underlying segment will be the same as the corresponding surface segment), Community (segments with similar features will act the same) and Context (rules need context information) the algorithm was able to learn the target rules from positive information only. With this experiment the authors have shown that computational learning methods applied to natural language can benefit from being equipped with basic linguistic knowledge.

The three studies described in this section have in common that they attempt to tackle the learnability problems of complex domains by adding restrictions to the domain. By showing that these restrictions are linguistically plausible they have contributed to reduce the gap between mathematical learning theory and observations in child language acquisition.

Chapter 2

Statistical Learning

In this chapter we will examine the application of a statistical learning technique to the acquisition of phonotactic models. The learning technique which we will use is called Hidden Markov Models. The chapter will start with an introduction to standard Markov models. After this we will examine the more elaborate Hidden Markov Models (HMMs). The experiments we will perform with HMMs have been divided in two groups. The first group of experiments consists of test experiments with a small data set. With these experiments we will try to find out what restrictions we have to impose on the HMMs and their training data in order to be able to make them acquire phonotactic knowledge. These experiments will be discussed in the third section. The fourth section will present the results of the HMMs that were applied to our main data set. The final section of the chapter will give some concluding remarks.

1 Markov models

Before discussing the Hidden Markov Models, we will present the standard Markov models. We start with a general description of these models. After this we will introduce two basic algorithms which are used in connection with Markov models: the forward procedure and the Viterbi algorithm.

1.1 General description of Markov models

A Markov model is a model consisting of states and weighted transitions. The task of a Markov model is recognizing or producing sequences. An example of a Markov model is shown in figure 2.1. It shows some popular target locations in a tourist walk

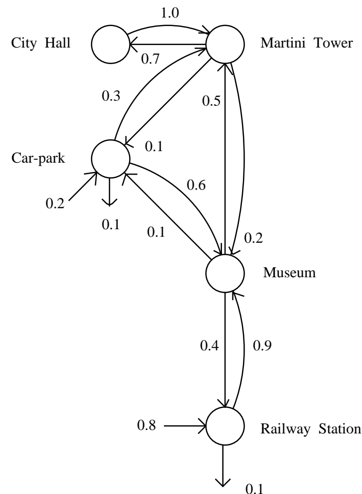


Figure 2.1: The City Walk Markov Model: A Markov model for a tourist walk through the city of Groningen. The states modeling locations are represented by filled circles and the transitions by arrows connecting them. The weights indicate the probability of moving from one state to another, e.g. when you are in the Museum the probability of moving to the Martini Tower is 0.5.

through the city of Groningen: the Museum (MU), the City Hall (CH) and the Martini Tower (MT). People start their walk from the Railway Station (RS) or from the Car park (CP).

These five locations are represented as states in the model. The weights of the links between the states indicate the probability that a visitor of a certain location will go to another location. 60% of the people present in the Car park at time *t* will be in the Museum at time *t+1* (0.6 link). All people in the City Hall will walk to the Martini Tower (1.0 link) and no people present in the Martini Tower will continue their walk by visiting the Railway Station next (no link). The starting locations of the walk are marked with links that start from outside the model. The probability that a walk starts in the Car park is 20% (0.2 link) and the probability that it starts in the Railway Station is 80% (0.8 link). A walk ends when the outward link from Car park (0.1) or Railway Station (0.1) is used.

The parameters of a Markov model can be arranged in matrices. One matrix, the A-matrix, contains the probabilities that concern the weights of internal links and an element a_{ij} of this A-matrix indicates the probability that a transition between state *i* and state *j* will be made. The Π-matrix contains the probabilities of starting in states so π_i indicates the probability that a Markov process starts in state *i*. Figure 2.2 shows

$$A = \begin{bmatrix} 0.00 & 0.90 & 0.00 & 0.00 & 0.00 \\ 0.40 & 0.00 & 0.10 & 0.50 & 0.00 \\ 0.00 & 0.60 & 0.00 & 0.30 & 0.00 \\ 0.00 & 0.20 & 0.10 & 0.00 & 0.70 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \end{bmatrix} \quad \Pi = \begin{bmatrix} 0.80 \\ 0.00 \\ 0.20 \\ 0.00 \\ 0.00 \end{bmatrix}$$

Figure 2.2: Parameter definition of the City Walk Markov Model[1]. In the A-matrix a horizontal row i contains the probabilities of leaving state i and moving to a state j. The order of the states is <Railway Station, Museum, Car park, Martini Tower, City Hall> The Π-matrix contains the probabilities of starting in state i. Examples: The probability of moving from the Museum to the Railway Station is 0.40 (coordinate 2,1 in the A-matrix) and the probability of starting in the Car park is 0.20 (third number in Π-matrix.

the matrix representation for the City Walk Markov Model.

1.2 The forward procedure

Now suppose that the people that are engaged in a walk change location once an hour. What would be the probability that someone is in the Martini Tower after a walk of two hours? If we want to compute this probability we have to find out all paths from starting locations that reach the Martini Tower in two steps. Then we have to compute the probabilities of these paths and compute their sum. This will give us the probability we are looking for.

Inspection of figure 2.1 reveals that there are two paths possible from one of the starting locations which reach the Martini Tower in two steps: Car park → Museum → Martini Tower and Railway Station → Museum → Martini Tower. The probability of starting in Car park is 0.2, moving from Car park to Museum has probability 0.6 and moving from Museum to Martini Tower 0.5. Therefore the probability of the first path is $0.2 * 0.6 * 0.5 = 0.06$. In a similar fashion we can compute the probability of the second path: $0.8 * 0.9 * 0.5 = 0.36$. The probability that someone is in the Martini Tower after a walk of two hours is equal to the sum of these two probabilities: $0.06 + 0.36 = 0.42$.

The general procedure that is used for computing the probability that a Markov model will be in a state s at a time t is called the FORWARD PROCEDURE: $\alpha_s(t)$. The definition of this procedure is:

[1]In general the A-matrix and the Π-matrix satisfy the properties $\sum_j A[i,j] = 1$ and $\sum_i \Pi[i] = 1$. The first property is not satisfied by this A-matrix because we did not include the two model-leaving 0.1 arcs in the matrix. We will elaborate on this in section 3.1.

$$\alpha_s(t) = the\ probability\ of\ being\ in\ state\ s\ at\ time\ t.$$
$$\alpha_s(0) = \pi_s \qquad (2.1)$$
$$\alpha_s(t+1) = \sum_k \alpha_{s_k}(t) * a_{s_k,s} \qquad (2.2)$$

Equation 2.1 defines that at time 0 the probability of being in a state s is equal to the probability of starting in state s. We can compute the probability of being in a state s at time point *t+1* if we know the probability of being a state at time *t*. First we have to multiply the probability of being in a state *k* with the weight of the link between state s_k and state s. After this we have to compute the sum of all these products (equation 2.2). Weights of non-existent links are defined to have value *0*.

The forward procedure can be used for computing the probability that someone is in the Martini Tower after a walk of two hours. The procedure value we are looking for is $\alpha_{MT}(2)$. We start with computing all $\alpha_s(0)$. With these values we compute all $\alpha_s(1)$ values which in turn can be used for computing $\alpha_{MT}(2)$.

$$\alpha_{RS}(0) = \pi_{RS} = 0.8$$
$$\alpha_{CP}(0) = \pi_{CP} = 0.2$$

All other $\alpha_s(0)$ are equal to 0 because the probability of starting in these states (π_s) is 0. In the following equations we will assume that the states are ordered, that is: s_1=Railway Station (RS), s_2=Museum (MU), s_3=Car park (CP), s_4=Martini Tower (MT) and s_5=City Hall (CH). Now the next step is:

$$\begin{aligned}
\alpha_{RS}(1) &= \sum_k \alpha_{s_k}(0) * a_{s_k,RS} = 0 \\
\alpha_{MU}(1) &= \sum_k \alpha_{s_k}(0) * a_{s_k,MU} = \\
&= \alpha_{RS}(0) * a_{RS,MU} + \alpha_{CP}(0) * a_{CP,MU} = \\
&= 0.8 * 0.9 + 0.2 * 0.6 = 0.84 \\
\alpha_{CP}(1) &= \sum_k \alpha_{s_k}(0) * a_{s_k,CP} = 0 \\
\alpha_{MT}(1) &= \sum_k \alpha_{s_k}(0) * a_{s_k,MT} = \\
&= \alpha_{CP}(0) * a_{CP,MT} = \\
&= 0.2 * 0.3 = 0.06 \\
\alpha_{CH}(1) &= \sum_k \alpha_{s_k}(0) * a_{s_k,CH} = 0
\end{aligned}$$

$$\begin{aligned} \alpha_{leave}(1) &= \sum_k \alpha_{s_k}(0) * a_{s_k,leave} = \\ &= \alpha_{RS}(0) * a_{RS,leave} + \alpha_{CP}(0) * a_{CP,leave} = \\ &= 0.8 * 0.1 + 0.2 * 0.1 = 0.1 \end{aligned}$$

There are no links from starting locations to the states Railway Station, Car park and City Hall. These locations cannot be reached in one step so their $\alpha_s(1)$ value is equal to *0*. The Martini Tower state can be reached from the Car park state in one step and the Museum state can be reached from both the Railway Station state and the Car park state. They have an $\alpha_s(1)$ value which is larger than *0*. The $leave$ state stands for leaving the city.

With the values of the forward procedure for time 1 we can now compute $\alpha_{MT}(2)$:

$$\begin{aligned} \alpha_{MT}(2) &= \sum_k \alpha_{s_k}(1) * a_{s_k,MT} = \\ &= \alpha_{MU}(1) * a_{MU,MT} = \\ &= 0.84 * 0.5 = 0.42. \end{aligned}$$

The Martini Tower can only be reached in two steps from one of the starting locations if the first step takes the tourist to the Museum. The probability $\alpha_{MT}(2)$ is equal to the one we have computed earlier.

By using the forward procedure we can speed up the computation of the probability of being in a state *s* at time *t*. For example, suppose that we have to perform such a computation for a ten-state Markov Model in which all states are connected with each other and in which each state can be a starting state. Suppose that we want to know a probability at time 10. If we compute this by summing the probabilities of all possible paths to the state we will get into trouble. We can start in ten different states (time 0) and at each new time point we can move to ten different states. This means that there are 10^{10} different paths to a state at time 10. Since every path from time 0 to time 10 requires 10 multiplications, we will need $10*10^{10}=10^{11}$. multiplications in order to compute the probability in this fashion.

With the forward procedure we can simplify this computation. At each time step we can compute each $a_s(t)$ with 10 multiplications (see equation 2.2). There are 10 $a_s(t)$, so at each time step we will need 10^2 multiplications. For a probability at time 10 we will need 9 of those sets of multiplications and 10 additional multiplications for the final time slice. In total we need 910 multiplications with the forward procedure compared with the 10^{11} multiplications with the naive method.

1.3 The Viterbi algorithm

Now suppose that the city council wanted to put signs along a popular walk through Groningen. The budget of the council is limited, and they only have enough money

for a walk passing four locations. They want as many people as possible to see the signs so they want to know what four-step path starting from a starting location is the most likely one.

In total there are 48 4-step paths starting in one of the two initial positions. If we compute the probability of each of these paths, we will find out that the path Railway Station → Museum → Martini Tower → City Hall → Martini Tower is the most probable 4-step path (probability 0.252). We saw in the previous section that the number of paths can become quite large. In fact, the number of possible paths increases exponentially as path length increases. For longer paths computing the most likely one in this manner will take a lot of effort. Fortunately a good computational method exists for computing the most probable path: the VITERBI ALGORITHM. Its definition is:

$$\begin{aligned}
\delta_s(t) &= the\ probability\ of\ the\ most\ likely\ path\ arriving \\
&\quad in\ state\ s\ at\ time\ t. \\
\delta_s(0) &= \pi_s && (2.3) \\
\delta_s(t+1) &= max_k(\delta_{s_k}(t) * a_{s_k,s}) && (2.4)
\end{aligned}$$

The probability of the most likely path to a state at time 0 is equal to the probability of starting in that state (equation 2.3). Probabilities at other time points can be computed by using the probabilities of the previous time points (equation 2.4). Partial paths that cannot contribute to the most likely path are pruned by the *max* function. The Viterbi algorithm can be used to find out the most likely four-step path in the City Walk Markov Model. In this example it will have to compute 5*5=25 probabilities instead of the 48 probability computations that were necessary with the previous method. We start with computing $\delta_s(0)$:

$$\begin{aligned}
\delta_{RS}(0) &= \pi_{RS} = 0.8 \\
\delta_{CP}(0) &= \pi_{CP} = 0.2
\end{aligned}$$

The other $\delta_s(0)$ are equal to 0 and have been left out. We continue by computing $\delta_s(1)$:

$$\begin{aligned}
\delta_{MU}(1) &= max_k(\delta_{s_k}(0) * a_{s_k,MU}) = \\
&= max(\delta_{RS}(0) * a_{RS,MU}, \\
&\qquad \delta_{CP}(0) * a_{CP,MU}) = \\
&= max(0.8 * 0.9, 0.2 * 0.6) = 0.72 \\
\delta_{MT}(1) &= max_k(\delta_{s_k}(0) * a_{s_k,MT}) = \\
&= \delta_{CP}(0) * a_{CP,Martini\ Tower} = \\
&= 0.2 * 0.3 = 0.06
\end{aligned}$$

Again, the other $\delta_s(1)$ are equal to zero and have been left out. We will not compute $\delta_{leave}(t)$ because the probability of leaving the Markov model has no influence on the probabilities at later time points; in this model people that leave the city are not coming back. Here are the computations for $\delta_s(2)$

$$\begin{aligned}
\delta_{RS}(2) &= max_k(\delta_{s_k}(1) * a_{s_k,RS}) = \\
&= \delta_{MU}(1) * a_{MU,RS} = \\
&= 0.72 * 0.4 = 0.288 \\
\delta_{MU}(2) &= max_k(\delta_{s_k}(1) * a_{s_k,MU}) = \\
&= \delta_{MT}(1) * a_{MT,MU} = \\
&= 0.06 * 0.2 = 0.012 \\
\delta_{CP}(2) &= max_k(\delta_{s_k}(1) * a_{s_k,CP}) = \\
&= max(\delta_{MU}(1) * a_{MU,CP}, \\
&\qquad \delta_{MT}(1) * a_{MT,CP}) = \\
&= max(0.72 * 0.1, 0.06 * 0.1) = 0.072 \\
\delta_{MT}(2) &= max_k(\delta_{s_k}(1) * a_{s_k,MT}) = \\
&= \delta_{MU}(1) * a_{MU,MT} = \\
&= 0.72 * 0.5 = 0.36 \\
\delta_{CH}(2) &= max_k(\delta_{s_k}(1) * a_{s_k,CH}) = \\
&= \delta_{MT}(1) * a_{MT,CH} \\
&= 0.06 * 0.7 = 0.042
\end{aligned}$$

The computations of $\delta_s(3)$ and $\delta_s(4)$ can be performed in a similar fashion. We will not list the complete computations here but confine with the results:

$$\begin{aligned}
\delta_{RS}(3) &= 0.0048 \\
\delta_{MU}(3) &= 0.2592 \\
\delta_{CP}(3) &= 0.036 \\
\delta_{MT}(3) &= 0.042 \\
\delta_{CH}(3) &= 0.252 \\
\delta_{RS}(4) &= 0.10368 \\
\delta_{MU}(4) &= 0.0216 \\
\delta_{CP}(4) &= 0.02592 \\
\delta_{MT}(4) &= 0.252 \\
\delta_{CH}(4) &= 0.0294
\end{aligned}$$

Location	Likes (L)	Dislikes (D)
Railway Station	0.7	0.3
Museum	0.5	0.5
Car park	0.1	0.9
Martini Tower	0.9	0.1
City Hall	0.8	0.2

Figure 2.3: The probabilities of liking locations in the City Walk through Groningen. Probabilities are only dependent on the currently visisted object. We assume that previously visited locations do not influence the opinion of the tourists. Example: the probability that someone that visited the Martini Tower disliked it is 0.1 and the probability that the person liked it is 0.9.

As we can see $\delta_{MT}(4)$ has the largest value (0.25200) so the path starting at the Railway Station and ending at the Martini Tower is the most likely path containing 4 steps. This δ-value only gives us the final point of this path. If we want to know the other locations present in the path, we will have to trace back the equations: $\delta_{MT}(4)$ most probable ancestor was $\delta_{CH}(3)$, its most probable ancestor was $\delta_{MT}(2)$ which on its turn was preceded by $\delta_{MU}(1)$ and $\delta_{RS}(0)$. So the most probable four-step path is Railway Station $\rightarrow$ Museum $\rightarrow$ Martini Tower $\rightarrow$ City Hall $\rightarrow$ Martini Tower and its probability is 0.252.

2 Hidden Markov Models

In the previous section we have examined Markov models. In this section we will present an extended version of these models: Hidden Markov Models. We will give a general description of these models and present the adapted versions of the forward procedure and the Viterbi algorithm which are used in Hidden Markov Models. After this we will describe how they can learn and how they can be used in practice.

2.1 General description of Hidden Markov Models

A researcher of the University of Groningen dedicates himself to finding out if people that take part in a city walk through Groningen like the locations they visit. To find the answer to this question he makes people fill in forms in which they are asked if they like the locations they have visited. The results of these forms are summerized in the table in figure 2.3.

The probability that someone likes the Railway Station (L) is 70% while the probability that the person does not like it (D) is 30%. All other probabilities are listed in the table. Now every walk through Groningen can be represented by a sequence con-

$$B = \begin{bmatrix} 0.70 & 0.30 \\ 0.50 & 0.50 \\ 0.10 & 0.90 \\ 0.90 & 0.10 \\ 0.80 & 0.20 \end{bmatrix}$$

Figure 2.4: An Extension of the parameter definition of the City Walk Markov Model: the B-matrix. In this matrix an entry b_{s_i,t_j} represents the probability that in state s_i token t_j can be produced. Example: the probability of producting L (like, t_1) in the Car park state (s_3) is 0.10 and the probability of producing D (dislike, t_2) in the same state is 0.90. The A-matrix and the Π-matrix remain unchanged.

taining the tourist's opinion about the different locations. For example, if the previous most-probable walk Railway Station → Museum → Martini Tower → City Hall → Martini Tower was made by someone who likes the Martini Tower and the Museum but does not like the Railway Station, the Car park and the City Hall, we would get the sequence DLLDL.

The DL-sequences are an interesting by-product of the City Walk model. They are not in a unique correspondence with state sequences. For example a walk consisting of Car park → Martini Tower → Museum → Railway Station → Museum taken by the same tourist can also be represented with the sequence DLLDL. Furthermore, the same walk of a tourist that likes all locations can be represented by LLLLL. So different walks can be represented with the same DL-sequences and the same walk can generate different DL-sequences when made by different tourists.

It is possible to extend Markov Models to make them simulate this behavior. In order to do that we define that in a state of the model different tokens can be produced. The probabilities that tokens are produced will be stored in a new parameter matrix of the model: the B-matrix (figure 2.4). In this matrix an entry b_{s_i,t_j} represents the probability that in state s_i token t_j can be produced. The other model parameters incorporated in the A-matrix and the Π-matrix remain the same as in figure 2.2.

Now we have obtained a HIDDEN MARKOV MODEL (HMM). The model is called *hidden* because from the token sequences generated by the model it is in general impossible to find out which states were passed through while generating the sequence. In the specific example of the City Walk Markov Model different state sequences could lead to the same DL-sequence. Therefore it was impossible to find out the state sequence used if we only know a token sequence.

2.2 The extended forward procedure

We have presented the forward procedure and the Viterbi algorithm for Markov models by asking two questions. We will do the same for the related functions for HMMs.

Our first Markov model question can be rephrased to: what is the probability of being at the Martini Tower after a walk of two hours in which the tourist did not like the starting location but in which he did like the second and the third location? In Hidden Markov Model terms, what is the probability of being at state Martini Tower after two steps while having produced token sequence DLL? We cannot use the forward procedure for Markov Models for computing this probability because it does not take into account the token production probabilities. A function that uses these probabilities is the EXTENDED FORWARD PROCEDURE:

$$
\begin{aligned}
\alpha_s(t, x_0..x_t) &= the\ probability\ of\ being\ in\ state\ s\ at\ time\ t \\
&\quad after\ producing\ sequence\ x_0..x_t \\
\alpha_s(0, x_0) &= \pi_s * b_{s,x_0} & (2.5) \\
\alpha_s(t+1, x_0..x_{t+1}) &= \sum_k \alpha_{s_k}(t, x_0..x_t) * a_{s_k,s} * b_{s,x_{t+1}} & (2.6)
\end{aligned}
$$

The probability of producing a token in a state at time $t = 0$ is equal to the probability of starting in that state multiplied with the probability of producing the token in the state (equation 2.5). The probability of producing a token x_{t+1} in state s at time $t + 1$ is equal to the sum of all values of the forward procedure for time t and state s_k multiplied with the probability of moving from state s_k to s and the probability of producing x_{t+1} in s (equation 2.6). Note that the extended forward procedure uses the probabilities of all tokens up to the current one. So the probability computed by this function is not only dependent on the current token but also on all previous tokens in the sequence.

We can use the extended forward procedure for answering the question we mentioned at the start of this section:

$$
\begin{aligned}
\alpha_{RS}(0, D) &= \pi_{RS} * b_{RS,D} = \\
&= 0.8 * 0.3 = 0.24 \\
\alpha_{CP}(0, D) &= \pi_{CP} * b_{CP,D} = \\
&= 0.2 * 0.9 = 0.18 \\
\alpha_{MU}(1, DL) &= \sum_k \alpha_{s_k}(0, D) * a_{s_k,MU} * b_{MU,L} = \\
&= \alpha_{RS}(0, D) * a_{RS,MU} * 0.5 + \alpha_{CP}(0, D) * a_{CP,MU} * 0.5 = \\
&= 0.24 * 0.9 * 0.5 + 0.18 * 0.6 * 0.5 = 0.162 \\
\alpha_{MT}(1, DL) &= \sum_k \alpha_{s_k}(0, D) * a_{s_k,MT} * b_{MT,L} = \\
&= \alpha_{CP}(0, D) * a_{CP,MT} * 0.9 = \\
&= 0.18 * 0.3 * 0.9 = 0.0486
\end{aligned}
$$

All $\alpha_s(0, x_0)$ and $\alpha_s(1, x_0..x_1)$ which are equal to 0 have been left out. Now we can compute the probability we are looking for:

$$\begin{aligned}
\alpha_{MT}(2, DLL) &= \sum_k \alpha_{s_k}(1, DL) * a_{s_k,MT} * b_{MT,L} = \\
&= \alpha_{MU}(1, DL) * a_{MU,MT} * 0.9 = \\
&= 0.162 * 0.5 * 0.9 = 0.0729
\end{aligned}$$

The probability which is the result of this computation is smaller than the $\alpha_{MT}(2)$ which was computed for the Markov Models (0.42) because of the multiplications with the token production probabilities (b_{s_i,t_j} <1).

2.3 The extended Viterbi algorithm

We can also rephrase our second Markov Model question for Hidden Markov Models: what is the most probable two step walk in which the tourist liked all locations? Or in Hidden Markov Models terms what is the most probable path corresponding with the sequence LLL? We cannot use the Viterbi algorithm for Markov Models because it does not include token production probabilities. We have to adapt this function to obtain the EXTENDED VITERBI ALGORITHM:

$$\begin{aligned}
\delta_s(t, x_0..x_t) &= the\ probability\ of\ the\ most\ likely\ path \\
&\quad ending\ in\ state\ s\ and\ producing\ token \\
&\quad sequence\ x_0..x_t \\
\delta_s(0, x_0) &= \pi_s * b_{s,x_0} && (2.7) \\
\delta_s(t+1, x_0..x_{t+1}) &= max_k(\delta_{s_k}(t, x_0..x_t) * a_{s_k,s} * b_{s,x_{t+1}}) && (2.8)
\end{aligned}$$

The probability of the most likely path that ends in state s and produces a sequence containing one token x_0 is equal to the probability of starting in s multiplied with the probability of producing x_0 in s (equation 2.7). The probability of the most likely path ending in s for a longer sequence is the maximal value that can be obtained by multiplying the probability of the most likely path for the prefix of the sequence with the probability of moving from the final state s_k of the prefix to s and the probability of producing the current token in s. The computation this extended Viterbi algorithm performs is sequence-specific just as the computation of the extended forward procedure.

This extended Viterbi algorithm can be used for finding out the most probable state path corresponding with the sequence LLL. First we compute the values of the δ-function for time $t = 0$:

$$\begin{aligned}
\delta_{RS}(0,L) &= \pi_{RS} * b_{RS,L} = \\
&= 0.8 * 0.7 = 0.56 \\
\delta_{CP}(0,L) &= \pi_{CP} * b_{CP,L} = \\
&= 0.2 * 0.1 = 0.02
\end{aligned}$$

We can use values for computing the values of the functions for $t = 1$:

$$\begin{aligned}
\delta_{MU}(1,LL) &= max_k(\delta_{s_k}(0,L) * a_{s_k,MU} * b_{MU,L}) = \\
&= max(\delta_{RS}(0,L) * a_{RS,MU} * 0.5, \delta_{CP}(0,L) * a_{CP,MU} * 0.5) = \\
&= max(0.56 * 0.9 * 0.5, 0.02 * 0.6 * 0.5) = 0.252 \\
\delta_{MT}(1,LL) &= max_k(\delta_{s_k}(0,L) * a_{s_k,MT} * b_{MT,L}) = \\
&= \delta_{CP}(0,L) * a_{CP,MT} * 0.9 = \\
&= 0.02 * 0.3 * 0.9 = 0.0054
\end{aligned}$$

The $\delta_s(0,L)$ and $\delta_s(1,LL)$ that are equal to zero and have been left out. With these results we can compute all $\delta_s(2,LLL)$:

$$\begin{aligned}
\delta_{RS}(2,LLL) &= max_k(\delta_{s_k}(1,LL) * a_{s_k,RS} * b_{RS,L}) = \\
&= \delta_{MU}(1,LL) * a_{MU,RS} * 0.7 = \\
&= 0.252 * 0.4 * 0.7 = 0.07056 \\
\delta_{MU}(2,LLL) &= max_k(\delta_{s_k}(1,LL) * a_{s_k,MU} * b_{MU,L}) = \\
&= \delta_{MT}(1,LL) * a_{MT,MU} * 0.5 = \\
&= 0.0054 * 0.2 * 0.5 = 0.00054 \\
\delta_{CP}(2,LLL) &= max_k(\delta_{s_k}(1,LL) * a_{s_k,CP}) * b_{CP,L} = \\
&= max(\delta_{MU}(1,LL) * a_{MU,CP} * 0.1, \\
&\qquad \delta_{MT}(1,LL) * a_{MT,CP} * 0.1) = \\
&= max(0.252 * 0.1 * 0.1, 0.0054 * 0.1 * 0.1) = 0.00252 \\
\delta_{MT}(2,LLL) &= max_k(\delta_{s_k}(1,LL) * a_{s_k,MT} * b_{MT,L}) = \\
&= \delta_{MU}(1,LL) * a_{MU,MT} * 0.9 = \\
&= 0.252 * 0.5 * 0.9 = 0.1134 \\
\delta_{CH}(2,LLL) &= max_k(\delta_{s_k}(1,LL) * a_{s_k,CH} * b_{CH,L}) = \\
&= \delta_{MT}(1,LL) * a_{MT,CH} * 0.8 = \\
&= 0.0054 * 0.7 * 0.8 = 0.003024
\end{aligned}$$

From these computation we can conclude that the most probable path that produces LLL ends in the Martini Tower. We can find out the previous locations by checking what state was used in the maximal part of the $\delta_{MT}(2, LLL)$ computation and this turns out to be the Museum ($\delta_{MU}(1, LL)$). The location before that can be found by checking what state was used in the maximal part of the $\delta_{MU}(1, LL)$ computation. This turns out to be the Railway Station ($\delta_{RS}(0, L)$) so the most probable path producing LLL is Railway Station $\rightarrow$ Museum $\rightarrow$ Martini Tower.

2.4 Learning in a Hidden Markov Model

Now suppose that in some distant country a group of engineers decides to rebuild the main tourist attractions of Groningen. The engineers also want to enable the visitors of New Groningen to experience the famous City Walk through Groningen. Unfortunately, the engineers do not know what the main buildings look like and which buildings were connected with each other. The only feature about the City Walk they were able to collect is a list of DL-sequences that were produced by participants in the City Walk through Groningen. The engineers decide to set up some wooden barracks with roads connecting them to each other and make tourists walk through this village. The tourists all take with them a form in which they mention what locations they visited and whether they liked the location. If the tourists thus produce a DL-sequence that is in the list the engineers are trying to reproduce, the engineers will do nothing. If, however, the DL-sequence is not in the list the engineers start improving or damaging the buildings and the roads. This process continues until the tourists only produce sequences that are in the list. The engineers have then succeeded in reproducing the City Walk through Groningen and they have succeeded in reconstructing the underlying Hidden Markov Model as far as the production of DL-sequences is concerned.

The problem of finding a Hidden Markov Model which produces a specific set of token sequences is a common task. Our goal is to obtain a Hidden Markov Model that produces sequences of characters. The model should assign high probabilities to sequences that are words in some language and low probabilities to sequences that cannot appear as words in the language. Note that we are not aiming at reproducing the exact underlying model for the language. We will try to find a model that *behaves* like the underlying language model. Like the engineers of New Groningen we only know the token sequences produced by the model we are trying to rebuild. We will be satisfied if we succeed in creating a model that is able to reproduce our data.

The problem is that there is no direct method for computing the parameters (the matrices A, Π and B) of a Hidden Markov Model that is able to produce a specific set of sequences with a large probability. Fortunately, there are methods for estimating the values of the parameters of such a Hidden Markov Model. The most well-known method for estimating the parameters of a Hidden Markov Model from a set of sequences is called the BAUM-WELCH ALGORITHM or the forward-backward algorithm which has been described in (Rabiner et al. 1986) and (Van Alphen 1992) among others. This algorithm consists of three steps:

1. Initialize the Hidden Markov Model with random parameter values.
2. Make the Hidden Markov Model compute the probability of every sequence in the set. During this computation we count how often transitions between states are used and how often tokens are being produced in each state. We use the resulting numbers for computing a new set of parameters.
3. We use the new parameter values for reinitializing the Hidden Markov Model. The new Hidden Markov Model will assign a higher probability to the set of training strings. Now we repeat step 2 and 3 until the behavior of the Hidden Markov Model stabilizes.

There are different methods for deciding when a HMM has become stable. We will discuss these in a later section. It is possible to prove that the Baum-Welch algorithm terminates so we can be sure that the algorithm will always be able to produce a stable HMM in a finite amount of time. The proof is complex and we will not list it here. Interested readers are referred to section 5.4 of (Huang et al. 1990)

The most complex step of the Baum-Welch algorithm is step 2. We will formalize this step by using the extended forward procedure (see equations 2.5 and 2.6) and three other algorithms we will introduce in this section. Our goal is to find new values for the HMM parameters a_{ij}, $b_{s_i,m}$ and π_{s_i}. The definitions of these parameters are:

$$a_{s_i,s_j} = \frac{probability\ of\ making\ a\ transition\ from\ s_i\ to\ s_j}{probability\ of\ being\ in\ state\ s_i} \tag{2.9}$$

$$b_{s_i,m} = \frac{probability\ of\ being\ in\ s_i\ while\ producing\ token\ m}{probability\ of\ being\ in\ state\ s_i} \tag{2.10}$$

$$\pi_{s_i} = probability\ of\ being\ in\ state\ s_i\ at\ time\ 0 \tag{2.11}$$

In equations 2.9 and 2.10 it is necessary to divide the numerator probabilities by the probability of being in state s_i to make sure that for each s_i all a_{s_i,s_j} add up to 1 and all $b_{s_i,m}$ add up to 1. In order to be able to compute new values for the A-matrix, we should be able to compute the probability that in a production of a sequence a specific transition between two states will be made. We can view the production of a sequence as consisting of three steps: the production of the current token, the production of the prefix of this token and the production of the suffix of the token.[2] We have an algorithm that models the production of the prefix of a token: the forward procedure. The first algorithm we will introduce here is the BACKWARD ALGORITHM: an algorithm that models the production of the suffix of a token (*e* is the empty sequence and *T* is the time at which the final element of the sequence is produced):

[2] In this chapter we do not use prefix and suffix as the linguistic terms. For us the prefix of a sequence is the subsequence from the start to the current token (non-inclusive) and the suffix a sequence is the subsequence from the current token (non-inclusive) to the end of the sequence.

$$\begin{aligned}
\beta_{s_i}(t, x_{t+1}..x_T) &= probability\ that\ a\ sequence\ x_{t+1}..x_T\ is\ produced \\
&\quad while\ the\ state\ at\ time\ \ t\ is\ s_i. \\
\beta_{s_i}(T, e) &= 1 \qquad (2.12) \\
\beta_{s_i}(t, x_{t+1}..x_T) &= \sum_j \beta_{s_j}(t+1, x_{t+2}..x_T) * a_{s_i,s_j} * b_{s_j,x_{t+1}} \qquad (2.13)
\end{aligned}$$

The backward algorithm is the counterpart of the extended forward algorithm. It computes the probability that a suffix of a sequence is produced while starting in a specific state (s_i). The probability that the empty string is produced in state s_i at time T is defined to be equal to one (equation 2.12). If we know the probabilities of a sequence which starts at time $t+1$ in any of the states s_j then we can compute the probability of the same sequence preceded by some token at time t and state s_i by multiplying the known value with the probability of moving from s_i to s_j and the probability of producing x_{t+1} in s_j and adding all these products together. Note that in this definition the production of a token is imagined as happening directly after the transition to the state that produces the token. That is why equation 2.13 contains $b_{s_j,x_{t+1}}$ and not b_{s_i,x_t} and why in 2.12 no token production has been taken into account.

We can combine the forward and the backward algorithm for computing the probability that the HMM assigns to a sequence:

$$\begin{aligned}
P_h(x_0..x_T) &= probability\ assigned\ to\ sequence\ x_0..x_T\ by\ an\ HMM \\
P_h(x_0..x_T) &= \sum_i \alpha_{s_i}(t, x_0..x_t) * \beta_{s_i}(t, x_{t+1}..x_T) \qquad (2.14)
\end{aligned}$$

The term $\alpha_{s_i}(t, x_0..x_t) * \beta_{s_i}(t, x_{t+1}..x_T)$ computes the probability of being in state s_i at time *t* while producing sequence $x_0..x_T$. Equation 2.14 computes the sum for all i which gives us the probability of producing sequence $x_0..x_T$ while being in an arbitrary state s_i at time t. This is equal to the probability of producing the sequence $x_0..x_T$. We can use $P_h(x_0..x_T)$ for computing the probability of being in state s_i at time *t*:

$$\begin{aligned}
\gamma_{s_i}(t) &= the\ probability\ of\ being\ in\ state\ s_i\ at\ time\ t \\
\gamma_{s_i}(t) &= \frac{\alpha_{s_i}(t, x_0..x_t) * \beta_{s_i}(t, x_{t+1}..x_T)}{P_h(x_0..x_T)} \qquad (2.15)
\end{aligned}$$

We need to divide $\alpha_{s_i}(t, x_0..x_t) * \beta_{s_i}(t, x_{t+1}..x_T)$ by the probability of producing sequence $x_0..x_T$ in order to make sure that for each t the $\gamma_{s_i}(t)$ probabilities sum up to 1. With this $\gamma_{s_i}(t)$ and the equations 2.10 and 2.11 we are now able to compute new values for π_{s_i} and $b_{s_i,m}$:

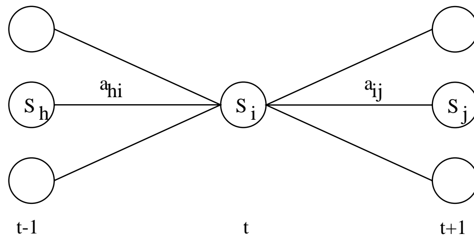


Figure 2.5: The computation of $\gamma_{s_i}(t)$, the probability of being in state s_i at time *t* while producing the sequence $x_0..x_T$. Compute the probability that the prefix of the sequence ends in s_i ($\alpha_{s_i}(t, x_0..x_t)$), multiply it with the probability that the suffix of the sequence starts in s_i ($\beta_{s_i}(t, x_{t+1}..x_T)$) and divide the result by the probability of the sequence ($P_h(x_0..x_T)$).

$$\begin{aligned}
\pi_{s_i} &= probability\ of\ being\ in\ state\ s_i\ at\ time\ 0\ = \\
&= \gamma_{s_i}(0) \qquad (2.16) \\
b_{s_i,m} &= \frac{probability\ of\ being\ in\ s_i\ while\ producing\ m}{probability\ of\ being\ in\ state\ s_i} = \\
&= \frac{\sum_t \{\gamma_{s_i}(t) \mid x_t = m\}}{\sum_t \gamma_{s_i}(t)} \qquad (2.17)
\end{aligned}$$

The numerator of equation 2.17 computes the sum of all $\gamma_{s_i}(t)$ for which $x_t = m$ holds. The summations over *t* in equations 2.17 are necessary to take into account all $\gamma_{s_i}(t)$ ($b_{s_i,m}$ is independent of time). Apart from abling us to compute these two HMM model parameters, $\gamma_{s_i}(t)$ can also be used for computing the denominator for equation 2.9. Now we have to develop a function for computing the numerator of that equation. We start by expanding equation 2.14:

$$\begin{aligned}
P_h(x_0..x_T) &= \sum_i \alpha_{s_i}(t, x_0..x_t) * \beta_{s_i}(t, x_{t+1}..x_T) = \\
&= \sum_i \alpha_{s_i}(t, x_0..x_t) * \sum_j \beta_{s_j}(t+1, x_{t+2}..x_T) * a_{s_i,s_j} * b_{s_j,x_{t+1}} = \\
&= \sum_i \sum_j \alpha_{s_i}(t, x_0..x_t) * a_{s_i,s_j} * b_{s_j,x_{t+1}} * \beta_{s_j}(t+1, x_{t+2}..x_T)
\end{aligned} \qquad (2.18)$$

The first line is equal to equation 2.14 and in the second line we have applied the definition of $\beta_{s_i}(t, x_{t+1}..x_T)$ (equation 2.13). By reordering the elements of that line

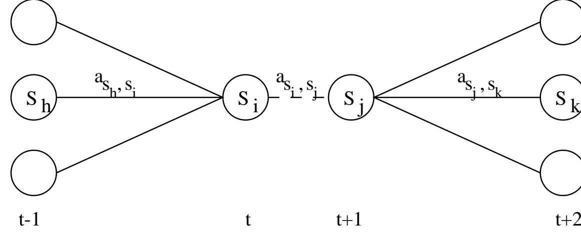


Figure 2.6: The computation of $\xi_{s_i,s_j}(t, x_{t+1})$, the probability of making a transition from state s_i to s_j: Compute the probability that the prefix of the sequence ends in s_i ($\alpha_{s_i}(t, x_0..x_t)$), multiply it with the HMM transition probability a_{s_i,s_j}, the HMM token production probability $b_{s_j,x_{t+1}}$ and the probability that the suffix of the sequence starts in s_j ($\beta_{s_j}(t+1, x_{t+2}..x_T)$) and divide the result by the probability of the sequence ($P_h(x_0..x_T)$).

($\alpha_{s_i}(t, x_0..x_t)$ is independent of j) we have derived the equation at the third line. We have expanded the β-term in order to get a term within the sums in which both the computation of the probability of making the transition from s_i to s_j (a_{s_i,s_j}) and the production of token x_{t+1} ($b_{s_j,x_{t+1}}$) are visible. We can use the final part of equation 2.18 for computing the probability of making a transition from one state to another (see also figure 2.6):

$$\begin{aligned}
\xi_{s_i,s_j}(t, x_{t+1}) &= probability\ of\ making\ the\ transition\ from\ state \\
&\quad s_i\ to\ s_j\ at\ time\ t\ and\ producing\ x_{t+1} \\
\xi_{s_i,s_j}(t, x_{t+1}) &= \frac{\alpha_{s_i}(t, x_0..x_t) * a_{s_i,s_j} * b_{s_j,x_{t+1}} * \beta_{s_j}(t+1, x_{t+2}..x_T)}{P_h(x_0..x_T)}
\end{aligned} \tag{2.19}$$

Now we need to divide $\alpha_{s_i}(t, x_0..x_t) * a_{s_i,s_j} * b_{s_j,x_{t+1}} * \beta_{s_j}(t+1, x_{t+2}..x_T)$ by the probability of producing sequence $x_0..x_T$ in order to make sure that for every t the $\xi_{s_i,s_j}(t, x_{t+1})$ probabilities sum up to 1. With this $\xi_{s_i,s_j}(t, x_{t+1})$ function we are now able to compute the a_{s_i,s_j} parameters of the HMM:

$$\begin{aligned}
a_{s_i,s_j} &= \frac{probability\ of\ going\ from\ s_i\ to\ s_j\ while\ producing\ x_{t+1}}{probability\ of\ being\ in\ state\ s_i} = \\
&= \frac{\sum_t \xi_{s_i,s_j}(t, x_{t+1})}{\sum_t \gamma_{s_i}(t)}
\end{aligned} \tag{2.20}$$

Again the summations over *t* are necessary to take into account all values of the functions for different *t* (a_{s_i,s_j} is independent of time). Now we have obtained formulas

for computing new values for HMM parameters. Our training process will start with random values for the HMM parameters a_{s_i,s_j}, $b_{s_i,m}$ and π_{s_i}. We make the HMM process the training data and by using the equations 2.16, 2.17 and 2.20 we will be able to obtain better values for the HMM parameters. We continue applying this process until the probabilities that the HMM assigns to the training data become stable. At that point we hope to have obtained an HMM which is a good model for the training data.

2.5 Using Hidden Markov Models in practice

In this section we have introduced the mathematical background of Hidden Markov Models (HMMs). In the next two sections we will apply HMMs on training data that consists of monosyllabic Dutch words. Instead of DL-sequences the HMMs will process arbitrary sequences of characters. The HMMs will assign scores to these sequences. This scores will be equal to:

$$
\begin{aligned}
P_h(x_0..x_T) &= \sum_i \alpha_{s_i}(t, x_0..x_T) * \beta_{s_i}(T, e) \\
&= \sum_i \alpha_{s_i}(t, x_0..x_T) \qquad (2.21)
\end{aligned}
$$

This is the sum for all s_i of the extended forward procedure applied at the complete string of which the production ended in state s_i. This equation was derived from equation 2.14. $\beta_{s_i}(T)$ is equal to 1 for all s_i (equation 2.12). The HMM will be trained by presenting the training data to it and applying the Baum-Welch algorithm until the HMM becomes stable. Here we have defined a stable HMM as an HMM that assigns scores to training strings that do not differ more than 1% of the scores assigned by the HMM before the final training round. In each training round the complete training data set will be processed.

When the HMM has become stable we will test it by applying it to the positive and the negative test data sets that we have described in chapter 1. The HMM should accept as many strings from the positive test data as possible and reject as many negative data as possible. We need to define a threshold score for deciding if a string is acceptable or of it is not. If a string receives a score that is higher than the threshold score it will be accepted and if it is lower than this threshold the string will be rejected.

The problem is that different HMMs will assign different scores to strings. Therefore it is impossible to determine a universal threshold value. Each HMM will require its own threshold value. Since we want all strings in the training data set to be accepted we will define the THRESHOLD SCORE as the smallest score that is assigned to an element of the training data set.

3 Initial Experiments

We have performed three initial experiments to find out how we need to configure the Hidden Markov Models (HMMs) in order to enable them to learn the phonotactic structure of monosyllabic Dutch words. In these experiments we have used a small data set: 3507 monosyllabic Dutch words which were extracted from the Dutch spelling guide (Spellingscommissie 1954). The HMMs were trained with 3207 words; 300 words were used as positive test data. The test data also contained an additional set of 300 randomly generated words that were constructed by taking into account the character frequencies and word length frequencies of the strings in the training data and the positive test data. No effort was taken to remove strings from the random data set that occurred in the training data or the positive test data. The data contained character representations of words rather than phonetic transcriptions. Three experiments have been performed with HMMs which contained seven states:

1. An experiment with standard data sets and random initial HMM parameter values.
2. An experiment with modified data sets and random initial HMM parameter values.
3. An experiment with modified data sets and initial HMM parameter values which had been derived from a phonological model.

We have chosen a seven-state HMM rather than an HMM with any other number of states because the linguistic model we have used for initializing the HMMs (the Cairns and Feinstein model, see section 2.4 of chapter 1) also contains 7 states. Using HMMs with the same number of states made the initialization process easier.

The next sections describe the results of these experiments.

3.1 A test experiment

In our first experiment we initialized the A, B and Π matrices with random values. Then we used the Baum Welch algorithm to train the HMM. We stopped training when the scores assigned by the HMM to the words in the training set did not change more than 1% compared with the values after the previous training round. The parameters of the HMM after training can be found in figure 2.7.

In HMMs the probabilities of the outgoing transitions of each state have to sum up to 1. In this HMM this is not the fact. The reason for this is that this HMM does not handle word boundaries explicitly. For example, the outgoing transitions probabilities of the first state (top row A-matrix) sum up to 98%. This means that the probability of leaving the model after visiting state s_1 is 2%.

Because of this implicit handling of the word boundaries the scores assigned to words were worthless. The HMM will never assign prefixes of a sequence a score that is smaller than the sequence itself because the score of $sequence + x$ is computed

$$A = \begin{bmatrix} 0.04 & 0.94 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.51 & 0.31 & 0.00 & 0.06 & 0.02 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.15 & 0.25 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.20 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.53 & 0.00 \\ 0.37 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix} \quad \Pi = \begin{bmatrix} 0.93 \\ 0.01 \\ 0.00 \\ 0.00 \\ 0.06 \\ 0.00 \\ 0.00 \end{bmatrix}$$

Figure 2.7: Parameter definition of the test HMM after training. The B Matrix is omitted. Apart from the backward connection from state 6 to state 1 (0.37) the HMM contains forward links only. The probabilities in the rows of the A-matrix do not add up to 1 because word boundaries were handled implicitly.

by multiplying the score of $sequence$ with the probability of x (a number with 1 as maximal value). For example, *rot* will receive a larger score than *rots* and *rots* itself will receive a larger score than *rotst*. After merging the test results with the random results we discovered that, apart from two exceptions, all words of length n+1 received a smaller score than words of length n. This means that the HMM will consider impossible four-character strings more probable than valid five-character words. This clearly is wrong. Of course, in natural language it is not always the fact that a complex suffix is less improbable than a simple suffix. For example, in Dutch a syllable ending in the character *c* is more improbable than a syllable ending in *ch*. Some way has to be found to model this fact.

To handle this problem we have added an end-of-word character to all words in our training and test data. This extra character can only be the output of an eighth state in the HMM. After processing a word the HMMs have to be in this last state. This state is a so-called null state ((Van Alphen 1992)): no transitions are possible from this state. The eighth state has not been made visible in the pictures in this chapter. All transition probabilities from the other states to the last state have been put in a special vector Ω (analogous to the Π vector, compare figure 2.7 with 2.8).

3.2 Orthographic data with random initialization

Our next experiment was similar to the previous apart from the fact that we have used modified data sets (with end-of-word characters). Again the A, B, Ω and Π matrix of a seven-state HMM were initialized with random values. The Baum Welch algorithm (see section 2.4) was applied repeatedly until all training pattern scores stayed within a 1% distance of the previous scores which required 51 training rounds. The result of this was an HMM (see figure 2.9) without backward transitions (see figure 2.9, the initial HMM contained some backward transitions). The character output of the states was interesting. Here is a list of characters which are most likely to be the output of

$$A = \begin{bmatrix} 0.19 & 0.64 & 0.16 & 0.01 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.33 & 0.67 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.68 & 0.25 & 0.06 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.01 & 0.81 & 0.13 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.27 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.28 \end{bmatrix} \quad \Pi = \begin{bmatrix} 0.17 \\ 0.68 \\ 0.12 \\ 0.03 \\ 0.00 \\ 0.00 \\ 0.00 \end{bmatrix} \quad \Omega = \begin{bmatrix} 0.00 \\ 0.00 \\ 0.00 \\ 0.01 \\ 0.05 \\ 0.73 \\ 0.72 \end{bmatrix}$$

Figure 2.8: Parameter definition of the randomly initialized HMM after training. The HMM does not contain backward links. The elements of each row in the A-matrix together with the corresponding element in the Ω-matrix add up to one.

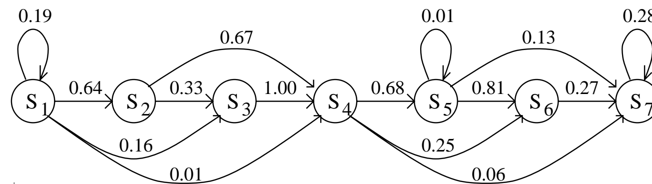


Figure 2.9: The randomly initialized HMM for monosyllabic Dutch words after training. This is a graphical representation of the A-matrix of figure 2.8. The HMM after training contains no backward links while before training links in any direction were possible.

the states in order of decreasing probability (ignoring characters which had less than 5% chance of being produced in the states):

- s_1: s c
- s_2: k t h p b v g d m z
- s_3: r l w n
- s_4: o a e i u
- s_5: e a n o u r i l j
- s_6: k r p l n f g m d s t
- s_7: s t

State s_4 has changed into a vowel production state. Only the five vowels *a*,*e*,*i*,*o* and *u* are produced by this state with a probability larger than 5%. In Dutch the *y* can be used

both as a vowel and as a consonant. In fact, *y* is the only other token that is produced by s_4 with a score that is larger than 10^{-8}. The probability of producing a *j*, which is used in Dutch writing as a suffix for the *i* to create a frequent diphthong, is largest in the after-vowel state s_5. The initial values in the A and B matrices of the HMM were random so the HMM did a good job in discovering the difference between vowels and consonants and discovering the special position of the *j* in Dutch.

The HMM assigned the following scores for sequences in the data sets:

	training data	positive test data	negative test data
Maximum:	$4.361*10^{-04}$	$4.379*10^{-04}$	$1.183*10^{-04}$
Average:	$7.472*10^{-05}$	$8.329*10^{-05}$	$3.429*10^{-06}$
Median:	$1.298*10^{-05}$	$1.286*10^{-05}$	$0.000*10^{+00}$
Minimum:	$1.927*10^{-11}$	$7.910*10^{-21}$	$0.000*10^{+00}$

The difference between the positive test data and the negative test data is most obvious in the difference between the medians. The average score of a data set is not a good comparison value because a small number of highly probable sequences will have a large influence on this average. When we consider the median value, the score of the negative data is a lot smaller than the score of the positive data. Therefore we can say that the HMM recognized the difference between the negative data set and the positive data set.

In individual cases it is more difficult to falsify data. For example, the sequence *pajn* (score $1.971*10^{-05}$), which clearly is not a Dutch syllable, would be ranked 126th in the list of 300 test data. It receives a larger score than the perfect Dutch syllable *worp* (score $1.868*10^{-05}$).

By using our threshold definition we obtain a threshold value of $1.927*10^{-11}$, the minimum score that the HMM has assigned to an element of the training data. With this threshold value the HMM accepts 298 words of the positive test data. It rejects 2 words: *stoischt* ($2.49*10^{-12}$) and *tsjech* ($7.910*10^{-21}$). This number of rejected words is acceptable. However, the HMM also accepts 57 words of the negative test data. Among these accepted words are words which are impossible in Dutch like *jlaj* ($6.744*10^{-7}$) and *ufhf* ($3.099*10^{-11}$). This HMM does not work as we would like it to do.

3.3 Orthographic data with linguistic initialization

In the third experiment the initial parameter values of the HMM were derived from the syllable model defined in (Cairns and Feinstein 1982) (see section 2.4 of chapter 1). All state transitions and character productions which are possible in the Cairns and Feinstein model received a random value. The others, for example the probability of moving from s_7 to s_1 and the probability of producing a consonant in the vowel state,

$$A = \begin{bmatrix} 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.05 & 0.24 & 0.71 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.32 & 0.17 & 0.48 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.14 & 0.19 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{bmatrix} \quad \Pi = \begin{bmatrix} 0.16 \\ 0.80 \\ 0.00 \\ 0.04 \\ 0.00 \\ 0.00 \\ 0.00 \end{bmatrix} \quad \Omega = \begin{bmatrix} 0.00 \\ 0.00 \\ 0.00 \\ 0.03 \\ 0.00 \\ 0.67 \\ 1.00 \end{bmatrix}$$

Figure 2.10: Parameter definition of the linguistically initialized HMM after training. The training process has only changed the values of the non-zero entries in these matrices. The zero entries in the matrices represent impossible links that were initialized on zero. The training process could not change these values.

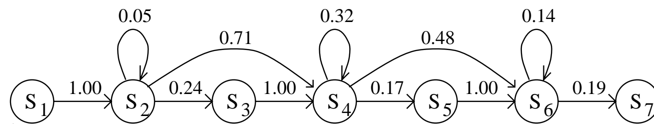


Figure 2.11: The linguistically initialized HMM for monosyllabic Dutch words after training. This is the graphical version of the A-matrix in figure 2.10. The training process only changed the weights of the links present in this picture. The result of removing specific links from the model in advance is that the HMM after training is more simple than the one in figure 2.9.

were set to zero. It was impossible for the HMM to change these zero-values. Its task was to find out the best values for the non-zero HMM parameters.

Again the Baum Welch algorithm was applied until the scores of the training patterns stayed within a 1% distance of the previous scores. This time only 15 training rounds were necessary. The parameters of the HMM after training can be found in figure 2.10. The characters which are most likely to be the output of the states are in order of decreasing probability (ignoring characters which have a probability of less than 5%):

- s_1: s
- s_2: k t h p b l v g c d
- s_3: r l w
- s_4: e a o i u

- s_5: n r l j m
- s_6: s t k p l d n f g r
- s_7: t s

Again, all five vowels in Dutch (*a*,*e*,*i*,*o*,*u*) are assigned to state 3. This was already the fact in the initial HMM and training did not change this fact.

The HMM assigned the following scores for sequences in the data sets:

	training data	positive test data	negative test data
Maximum:	$9.644{*}10^{-04}$	$6.794{*}10^{-04}$	$1.497{*}10^{-04}$
Average:	$7.588{*}10^{-05}$	$8.634{*}10^{-05}$	$3.653{*}10^{-06}$
Median:	$6.945{*}10^{-06}$	$7.654{*}10^{-06}$	$0.000{*}10^{+00}$
Minimum:	$7.418{*}10^{-13}$	$6.781{*}10^{-13}$	$0.000{*}10^{+00}$

A comparison between positive test data and negative test data leads to the same result as in the previous experiment: the average scores show no difference while the median scores show a clear difference between the two data sets. Comparing individual cases remains a problem. The acceptance threshold value is $7.4{*}10^{-13}$. With this threshold the HMM rejects one word of the positive test data set ($tsjech$). However, the HMM accepts 112 words of the negative test data among which words like *lve* ($9.202{*}10^{-07}$) and *fbep* ($4.111{*}10^{-07}$). This HMM accepts too many strings.

3.4 Discussion

Ideally it should be possible to choose some threshold score for an HMM and decide that every sequence with a score above this threshold is a possible sequence in the language while a sequence with a score below the threshold is not. In order to be able to do this correctly, all impossible words should receive a lower score than the 'most improbable' word in the language. However, we have not been able to find such a perfect threshold in the previous experiments. The most improbable word of the positive test data, *tsjech*, has received the score $6.781{*}10^{-13}$ of the linguistically initialized HMM. This score is lower than the scores of some words of the negative data set which are impossible in Dutch, for example *zrag* ($1.747{*}10^{-05}$), *pesf* ($7.323{*}10^{-06}$) and *jlaj* ($5.496{*}10^{-07}$). Figure 2.12 shows a comparison between words in the positive test data and words in the negative test data. The randomly generated impossible word *ddne* is about as probable as the Dutch word *snoodst*, the impossible *gvna* is about as probable as *placht* and there are more couples like that.

There are two explanations for this behavior. The first is that HMMs in general will assign a lower score to longer words than to shorter words. The Dutch word *tsjech* contains six characters while the non-words mentioned only contain four. However, this HMM feature can only be a part of the explanation. When we try finding impossible six-character words in the negative test data we find *letfdh* (score $8.218{*}10^{-11}$), *fobhlh*

281	0.00000002702359	schrap	yuz
282	0.00000002672019	knapst	yuz
283	0.00000002419223	grootst	yuz
284	0.00000002289886	zeeuws	daxn
285	0.00000002171036	joodst	daxn
286	0.00000001651132	fijnst	tzoeh
287	0.00000001589515	schaak	tzoeh
288	0.00000001508380	blondst	tzoeh
289	0.00000000943477	snoodst	ddne
290	0.00000000889425	bruutst	yjw
291	0.00000000858229	schold	ywua
292	0.00000000758883	placht	gvna
293	0.00000000730357	schoor	sewz
294	0.00000000710498	smacht	sewz
295	0.00000000636563	speech	uuuar
296	0.00000000050732	schoolst	zgoyt
297	0.00000000003132	echtst	odhnf
298	0.00000000002609	stoischt	odhnf
299	0.00000000001088	knechts	qnpu
300	0.00000000000068	tsjech	fobhlh

Figure 2.12: The 20 most improbable words (according to linguistically initialized HMM) in the positive test data together with negative test data that is equally probable. The scores assigned to negative strings are too high. In this respect the performance of the HMM can be improved if it can recognize that certain characters pairs do not occur in monosyllabic Dutch words: *yu*, *xn*, *dd*, *yj*, *yw*, *gv*, *wz*, *zg*, *dn*, *qn* and *lh*.

($1.03*10^{-12}$) and *edfgdg* ($6.5*10^{-13}$). These are all impossible six-character words in Dutch and only the third one receives a lower score than the valid six-character word *tsjech*.

Another explanation for the behavior of the HMM can be found when we look at the most probable processing sequence of the Dutch word *pijn* (pain) by the linguistically initialized HMM. The HMM will start in s_2 (p), move to s_4 (i), move to s_5 (j) and finish in s_6 (n). Now let's keep this state sequence and replace the characters which are produced by other characters. For example, we can replace the *i* produced in s_4 by an *a*. The *a* is more likely to be produced in s_4 than the *i* (see the state character schema in section 3.3, on each row characters are ordered from most frequent to least frequent) so we have obtained a word *pajn* which has a more probable main state sequence than the word *pijn*. Unfortunately, *pajn* is impossible in Dutch.

The problem here is clear. In monosyllabic Dutch words a *j* can follow an *i* but it cannot follow an *a*. However, the probability that the HMM will produce a *j* in s_5 is independent of the character produced in s_4. The HMM does not have memory of

previous parts of the sequence. Therefore it will still assign incorrect scores to some sequences.

A straightforward solution to this problem is to change the tokens the HMMs process from one character tokens to two or three character tokens. So we change the HMMs from a unigram model (no context characters) to a bigram model (one character context) or a trigram model (two character context). A test with a bigram HMM resulted in the desired behavior: because no *aj* bigrams occur in the training data a randomly initialized HMM will reject *pajn* (score 0.0) after training. This data format modification solves the problem of *pajn* but it might cause a new problem when we are going to work with HMMs that start from some initial phonotactic knowledge. If we want to use bigrams or trigrams, we will have to find out how to initialize a model that processes these structures. The initialization model we want to use, the Cairns and Feinstein model (see section 2.4 of chapter 1), does not contain an explicit context environment.

Another problem is that in HMMs a sequence can never be more probable than its prefix (the sequence without the final character). HMMs compute the score of a sequence consisting of a prefix plus one extra character by multiplying the score of the prefix with the probability that the character follows the prefix. Neither of these values will be larger than one. Because of this way of computation longer sequences will receive a smaller score than shorter sequences. We want to compute the probability that a sequence is present in the vocabulary of a language. This probability does not depend on the length of the sequence only. Therefore the scores the HMM compute should be changed to sequence-length-independent scores.

After having examined the scores the bigram HMM had assigned to the training data, we observed that the average scores of sequences of length n were about 10 times as large as scores of sequences of length n+1. Therefore we have decided to multiply all HMM scores with a factor $10^{sequence\ length}$ in order to decrease the influence of length on sequence score.

4 Experiments with bigram HMMs

In this section we will describe four series of experiments that we have carried out with bigram HMMs. We will start with a general description of the set-up of the experiments. After that we will present the results of the four experiment series. The series are divided in two groups: in one group we have used orthographic data and in the other group we have used phonetic data. In each group we have performed two series of experiments: one with HMMs that were initialized randomly and one with HMMs that we initialized by using the phonological model of Cairns and Feinstein that was described in chapter 1. Each experiment will be described in a separate section.

4.1 General bigram HMM experiment set-up

In order to create a bigram HMM, we modified the way the HMM interpreted strings. The unigram HMM interprets one character as one token: *splash=s-p-l-a-s-h*. Our bigram HMM divides strings into two-character tokens: *splash=sp-pl-la-as-sh*. By using this simple input interpretation, we were able to use the same theoretical learning model as in the previous experiments.

The interpretation created a problem. We consider an HMM production of a bigram as the production of the second character of the bigram in the context of the first. This means that the production of a six-character word like *splash*⋆ (⋆ is the word-end character) contains six steps (the five bigrams mentioned above plus the bigram *h*⋆). These six steps produce the characters *plash*⋆. The first character of the word will never be produced because during word production there is no bigram which contains the first character at the second position.

The omission of the production of the first character of the words generated erratic behavior from the HMMs. The one-character words in the training set received a zero-score from the HMMs which caused them to collapse. We solved this problem by expanding our data representation by adding a word-start character to all words. The production of a word then involved the production of a word-start character/first character bigram (for *splash* this is ∧*s*, ∧ is the word-start character) which means that now the first character of the word will be produced.

Apart from changing the representation of the data, we made another change in these bigram experiments. We observed that the random initialization of the HMM parameters influences the HMM performance. Because of the initial values of the model, the learning performance can differ. To minimize this influence we performed have five experiments with different initial values in each series. The average performance of the five experiments has been used as the result of the experiment series.

In these experiments we have used the large data set described in section 2.2 of chapter 1: 5577 words in orthographic representation or 5084 words in phonetic representation as training data, a 600 words positive test data set and a negative test data set containing 600 strings.

4.2 Orthographic data with random initialization

The first series of five experiments involved training randomly initialized HMMs on orthographic data. We have used the orthographic data described in section 2.2 of chapter 1: a training data set of 5577 monosyllabic Dutch words, a positive test data set of 600 monosyllabic Dutch words that did not occur in the training data set and a set of 600 negative test strings that did not occur in the previous two data sets. The results of these experiments can be found in figure 2.13.

On average the HMM needed 77.8 rounds to become stable. The stability criterion used was the same as in the earlier experiments: the HMM was considered stable when after a training round the evaluation scores of the words in the training sequence remained within a 1% distance from the scores that were assigned to them before this

nbr.	rounds	threshold	positive accepts	negative rejects
1.	108	$3.599*10^{-13}$	591	564
2.	87	$1.090*10^{-12}$	594	535
3.	81	$5.909*10^{-14}$	594	544
4.	48	$1.311*10^{-13}$	594	523
5.	65	$5.637*10^{-13}$	594	565
avg.	77.8±20.3		593.4±1.2 (98.9%)	546.2±16.4 (91.0%)

Figure 2.13: The results of five experiments with randomly initialized bigram HMMs that processed orthographic data. After an average of 78 training rounds the HMMs accepted on average 593 words of the positive test data set (98.9%) and rejected 546 strings of the negative test data set (91.0%). Nineteen negative strings were accepted by all five HMMs and five positive words were rejected by all five HMMs.

round. The resulting models accepted on average 593.4 of the 600 positive test words (98.9%) and rejected 546.2 of the 600 negative strings (91.0%). Six positive test words were rejected by all models: *ij's*, *q's*, *fjord*, *f's*, *schwung* and *t's*. The models assigned a low score to *ij's* because it contains a trigram (*ij'*) that does not occur in the training data. The other five words contain a bigram that was not present in the training data. The HMMs set the probability of occurrence of this bigram to zero. Therefore the scores of these words also became zero.

Nineteen of the 600 negative strings were accepted by all five models: *deess*, *enc*, *horet*, *ieer*, *maung*, *metet*, *oarp*, *ooe*, *oui*, *ousc*, *sassk*, *sspt*, *teaq*, *tskip*, *tspt*, *uai*, *uast*, *waese* and *woic*. These strings are not acceptable as monosyllabic Dutch words. Some of them consists of two syllables (*horet* and *metet*) and others do not even contain a vowel (*sspt* and *tspt*). Most of these misclassifications of the models can be explained by the small context the models have been using. For example, *ieer* consists of three very common bigrams *ie*, *ee* and *er* and the models use this fact to assign a high score to the word. However, the combination of these three bigrams in a Dutch monosyllabic word is not possible. The models could have been prevented from making this mistake if they had been using a larger context: the trigram *iee* does not occur in the training data. The two-syllable strings in this set can be explained by the occurrence of some accepted words from foreign languages in the training data like *shaket* and *faket*. Acceptance of the consonant words was caused by the presence of the two consonant interjections *pst* and *sst* in the training data set.

The errors for the positive test set are reasonable but we are less satisfied with the errors the HMMs make for the negative test data. The tendency of the models to accept too many unacceptable strings can be contributed to the small one character context that they are using. Expanding the context of the models would mean using trigrams instead of bigrams. However, then we would run into computational problems. The trigram models will simply need more computational resources for training than we presently have available. Therefore we will try to improve the performance of these

$$A = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \Pi = \begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \Omega = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} vowels & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ consonants & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 \\ y & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

Figure 2.14: Initial configuration for bigram HMMs for orthographic data that start learning from linguistic knowledge. The value 0.0 indicates an impossible link or an impossible character output and the value 1.0 indicates possible links or character output.

models in a different fashion. We will supply the HMMs with some initial linguistic knowledge and thus attempt to put constraints on the models that will be produced by training. Our hope is that the constrained models will be more strict when evaluating negative strings.

4.3 Orthographic data with linguistic initialization

In the second series of experiments we have used the bigram HMMs with an initial configuration that was derived from the model from Cairns and Feinstein. In the initial configuration we ignored the first characters of bigrams and we treated the output of a *XY* bigram as the output of a *Y* unigram. The initial configuration contained two types of values: value 0.0 and values larger than 0.0. The first value type indicated links or bigram outputs which are impossible according to the phonological model from Cairns and Feinstein. It was impossible for the HMM to change this value during training. The other value type was used for parameters that represented possible links or possible bigram outputs. These parameters were initialized with a random value.

An outline of the initial HMM configuration can be found in figure 2.14 and figure 2.15. This initial configuration was based on the Cairns and Feinstein model (see section 2.4 of chapter 1). The original version of the Cairns and Feinstein model is unable to explain the structure of all strings in orthographic training data. An HMM that would use this model as initialization model would assign zero-scores to part of our training data. This would make it unfit as an orthographic model since strings with zero-scores should be rejected and we require that our models accept all training data.

We have made three extensions to the standard Cairns and Feinstein model in order to make it usable as an initial orthographic model. First, the initial configuration

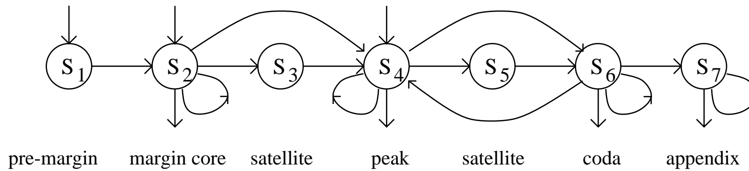


Figure 2.15: Initial bigram HMM for orthographic data. This is a graphical representation of the HMM parameters presented in figure 2.14. The visible features that were added to the original Cairns and Feinstein model are the self-links at state 2 (margin core), state 4 (peak) and state 6 (coda), the outward link from state 2 (margin core) and the backward link from state 6 (coda) to state 4 (peak). These links are necessary to enable the model to handle consonant clusters (*ch*, vowel clusters (diphtongs) and the non-vowel words in the training data set like *b*, *c*, *pst* etcetera.

for the A-matrix contains three extra links from a state to itself namely for state 2 (margin core), state 4 (peak) and state 6 (coda). These links are necessary because in orthographic data some consonants (for example *ch*) and some diphtongs (for example *au*) are represented by two tokens while the production of specific tokens is restricted to specific states by the B-matrix, for example vowels can only be produced by state 4 (peak). When sounds are represented by a cluster of tokens it is necessary to use a state that should produce such a cluster a number of times in succession. In order to be able to do that the state should contain a link to itself. The link from state 7 (appendix) to itself is necessary because the appendix can contain more than one *s* or *t*, for example: *tam*, *tams* and *tamst* in which the *s*'s and *t*'s should appear in the appendix (this is already a feature of the standard Cairns and Feinstein model).

The second extension is the backward link: state 6 → state 4 = coda → peak. The HMMs were not able to process the training data set without this backward link. Leaving out the backward link would make the HMMs assign zero-scores to accepted loan words as *ace*, *creme* and *file*. These words contain two isolated vowel groups. However, we will enable only one state (peak) to produce vowels. To be able to process words with two isolated vowel groups the HMMs will have to use this state twice and therefore a backward link to state 4 (peak) is necessary. The danger of having such a link in the HMMs is that they could use it for assigning high scores to multiple syllable words.

The third extension of the Cairns and Feinstein model in this initial model is the added possibility to finish a string after having processed the margin core (state 2). In the initial model this is represented by a link from state two to the hidden eight state. This link becomes visible in the second element of Ω-matrix in figure 2.14. The link is necessary to enable the HMMs to process interjections like *pst* and *sst* and the consonants of alphabet (*b*, *c*, *d*, etc.) that are also present in our training data set as words.

nbr.	rounds	threshold	positive accepts	negative rejects
1.	43	$3.313*10^{-15}$	595	509
2.	17	$9.947*10^{-17}$	595	511
3.	63	$3.337*10^{-15}$	595	508
4.	22	$1.036*10^{-16}$	595	513
5.	47	$1.439*10^{-14}$	595	514
avg.	38.4±16.9		595.0±0.0 (99.2%)	511.0±2.3 (85.2%)

Figure 2.16: The results of five experiments with linguistically initialized bigram HMMs that processed orthographic data. After an average of 38 training rounds the HMMs accepted 595 words of the positive test data set (99.2%) and rejected on average 511 strings of the negative test data set (85.2%). Eighty-two negative strings were accepted by all five HMMs and five test words were rejected by all five HMMs.

The initial B-matrix contains three groups of tokens. The first group consists of the vowels which can only be produced by state 4 (peak). The second group consists of the consonants. These can be produced by any state except state 4 (peak). We have regarded the single quote character (', among others present in *d'r*, *j's* and *vla's*) as a vowel. Finally there is the *y* which can be used both as a vowel and a consonant in Dutch. This character can be produced by any state.

Like in the previous section we performed five experiments with different initial parameter values. The results can be found in figure 2.16. The HMMs needed on average only 38.4 rounds to become stable. Thus they trained faster than the HMMs that were initialized randomly ($t(4)=3.0$, $p<0.025$, see section 2.5 of chapter 1). The linguistic initialization procedure resulted in small increase of the positive test words that were accepted: on average 595.0 compared with 593.4 for the randomly initialized HMMs ($t(4)=2.7$, $p<0.05$). Contrary to our goal the HMMs with linguistic initialization rejected fewer incorrect strings from the negative test data set than the HMMs that were initialized randomly: 511.0 compared with 546.2 ($t(4)=4.3$, $p<0.01$). We have to conclude that the phonological model we used for initializing the HMMs is not suitable for our orthographic data.

We have inspected one of the HMMs that resulted after training from a random initialization. This model suggested that we should make three changes to our initial model. First, we should allow vowel production in two states instead of one state. The model suggested to use state 6 as an extra vowel state. This state would be allowed to produce both consonants and vowels. The extra vowel state is necessary for being able to process the foreign words with two vowel clusters. As a result of extending the production capabilities of state 6 we can remove the backward link between state 6 and state 4. This is the second change we make to the model. Finally, the trained HMM processed the quote character as a consonant, not as a vowel. We will make this change in the initial model as well. There were other differences between the trained

$$A = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \Pi = \begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \Omega = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} vowels & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ consonants & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 \\ y & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

Figure 2.17: Modified initial configuration for bigram HMMs for orthographic data that start learning from linguistic knowledge. In order to mimic the behavior of the trained models with random initialization, we have removed the backward link from state 6 to state 4 and allowed state 6 to produce vowels. Furthermore, we have treated the quote character ' as a consonant instead of a vowel.

randomly initialized model and our linguistic initial model but these three were the most important ones. The initial model can be found in figure 2.17 and 2.18.

The HMMs trained with this initial configuration performed better with respect to the negative strings than the previous linguistically initialized HMMs (figure 2.19). They rejected 567 of the 600 negative strings (94.5%) compared with on average 511.0 rejected strings by the previous HMMs ($t(4)=49.1$, $p<0.005$) The training time needed was about as long as the previous HMMs (52.2 rounds compared with 38.4 rounds, $t(4)=1.1$ $p>0.1$) while they accepted fewer strings of the positive test data (593.4 compared with 595, $t(4)=6.5$, $p<0.005$). If we compare these HMMs with the randomly initialized HMMs we find out that they need about the same training time (52.2 rounds compared with 77.8 rounds, $t(4)=1.8$, $p>0.05$), accept the same number of positive test words (593.4 compared with 593.4, $t(4)=1.8$, $p>0.25$) and reject more strings of the negative test data set (567 compared with 546.2, $t(4)=2.5$, $p<0.05$).

We can conclude that for orthographic data the performance of the HMMs can be improved by starting training from a good initial HMM configuration. Constructing the initial HMM from a phonological model without making any data-specific adjustments did not provide us with good results. The difference between a good phonological model and a good orthographic model is too large.

4.4 Phonetic data with random initialization

The third series of five experiments involved training randomly initialized HMMs to process phonetic data. We have used the phonetic data described in section 2.2: a

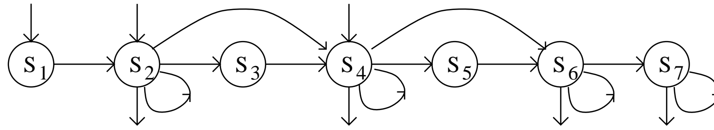


Figure 2.18: Modified initial bigram HMM for orthographic data. This is a graphical representation of the HMM parameters presented in figure 2.17. The backward link from state 6 to state 4 has been removed from the model. The other features that were added to the original Cairns and Feinstein model, the self-links at state 2 (margin core), state 4 (peak) and state 6 (coda), and the outward link from state 2 (margin core), remained in the model.

nbr.	rounds	threshold	positive accepts	negative rejects
1.	55	$2.841*10^{-14}$	593	567
2.	52	$5.974*10^{-14}$	594	567
3.	21	$6.092*10^{-14}$	594	567
4.	82	$2.778*10^{-14}$	593	567
5.	51	$2.847*10^{-14}$	593	567
avg.	52.2±19.3		593.4±0.5 (98.9%)	567±0.0 (94.5%)

Figure 2.19: The results of five experiments with bigram HMMs that processed orthographic data and used the modified linguistic initialization. After an average of 52 training rounds the HMMs accepted 593 words of the positive test data set (98.9%) and rejected 567 strings of the negative test data set (94.5%). 33 negative strings were accepted by all five HMMs and 6 test words were rejected by all five HMMs.

training data set of 5084 monosyllabic Dutch words, a positive test data set of 600 monosyllabic Dutch words that did not occur in the training data set and a negative test set of 600 strings. The results of these experiments can be found in figure 2.20.

These five HMMs performed equally well as the five randomly initialized HMMs that were trained on the orthographic data. They needed on average 68.6 training rounds to become stable (similar the 77.8 for orthographic data, t(4)=0.5, p>0.25) after which they accepted on average 594.6 words (99.1%) of the positive test data set (similar to the 593.4 for orthographic data, t(4)=1.7, p>0.05) and rejected 565.6 strings (94.3%) of the negative test data set (better than the 546.2 for orthographic data, t(4)=2.4, p<0.05). Five words of the positive test data set were rejected by all five HMMs: fjord [fjɔrt], fuut [fyt], schwung [ʃwuŋ], schmink [ʃmiŋk] and schminkt [ʃmiŋkt]. These words contain bigrams that are not present in the training data: [fj], [fy], [uŋ] and [iŋ]. The HMMs assign the score 0 to these bigrams and therefore the scores of these words also become 0.

nbr.	rounds	threshold	positive accepts	negative rejects
1.	48	$3.401*10^{-10}$	595	566
2.	58	$6.274*10^{-11}$	595	568
3.	131	$2.151*10^{-10}$	593	563
4.	41	$4.240*10^{-10}$	595	565
5.	65	$4.425*10^{-10}$	595	566
avg.	68.6±32.3		594.6±0.8 (99.1%)	565.6±1.6 (94.3%)

Figure 2.20: The results of five experiments with randomly initialized bigram HMMs that processed phonetic data. After an average of 69 training rounds the HMMs accepted on average 595 words of the positive test data set (99.1%) and rejected 566 strings of the negative test data set (94.3%). Twenty-eight negative strings were accepted by all five HMMs and five positive test words were rejected by all five HMMs. Twenty-four of the universally accepted negative strings had an acceptable phonetic representation. When we take this into account, the average performance of the HMMs on the negative test data set becomes 98.3%.

Twenty-eight negative strings were accepted by all five HMMs: astt [ɑst], brhat [brɑt], cci [ki:], ckeds [skɛts], cto [sto], deess [de:s], ejh [ɛj], ejss [ɛjs], fovhst [fɔfst], hurwd [hœrwt], kkraeb [kre:p], klolc [klɔlk], kuktzt [kœktst], nalc [nɑlk], oarp [o:rp], ousc [ɑusk], piuttd [pju:t], roqks [rɔks], sassk [sɑsk], teaq [ti:k], terh [tɛr], tskip [tskıp] ttik [tık], ttra [tra:], ttui [t∧y], twosd [twɔst], udsb [œtsp] and uzs [y:s]. The orthographic representations of these words are not acceptable as monosyllabic Dutch words but 24 of the 28 of the phonetic representations are acceptable. The four strings that do not have an acceptable phonetic representation are: hurwd [hœrwt], klolc [klɔlk], tskip [tskıp] and udsb [œtsp]. The conversion of the orthographic representations to the phonetic representations has been done by a native Dutch speaker and this might have resulted in 'quasi-Dutch' transcriptions. An inspection of the negative data set resulted in two more acceptable transcriptions: tzips [tsIps] and ttsue [tsy:]. If we omit the 26 acceptable negative strings from the data, the model has rejected on average 564.2[3] of 574 negative strings (98.3%) which is a good score.

4.5 Phonetic data with linguistic initialization

The results of the randomly initialized HMMs that processed phonetic data were acceptable. Still we are interested what the HMMs will do when they are provided with basic initial phonotactic knowledge. In this experiment series we will apply linguis-

[3] When the strings tzips [tsIps] and ttsue [tsy:] are removed from the data the average rejection scores will decrease with 1.0 because all HMMs rejected ttsue [tsy:] and with another 0.4 because two HMMs rejected tzips [tsIps].

$$A = \begin{bmatrix} 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad \Pi = \begin{bmatrix} 1.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \quad \Omega = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} vowels & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ consonants & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

Figure 2.21: Initial configuration for bigram HMMs for phonetic data that start from basic phonotactic knowledge. The value 0.0 indicates an impossible link or an impossible character production and the value 1.0 indicates possible links or possible character productions. The non-zero parameters were initialized with random values before training.

tically initialized HMMs to phonotactic knowledge. The initialization model we will use is the model of Cairns and Feinstein we have introduced in section 2.4 of chapter 1. Again we had to adapt the standard Cairns and Feinstein model in order to enable it to handle all words of the training data set. This set contains two interjections (*pst* and *sst*) and one abbreviation (*s*) that cannot be explained with the Cairns and Feinstein model. In order to enable the models to handle these words we have added a link from state 2 (margin core) to hidden eight state (see Ω-matrix in figure 2.21 and figure 2.22). This link will enable the model to accept strings that do not contain a vowel. Furthermore, we had to link state 1 (pre-margin) to itself in order to enable the models to process the consonant clusters in *pst* and *sst*. All impossible links were removed from the HMMs and they were prevented from restoring them. The initial model can be found in figures 2.21 and 2.22.

Like in the experiments with the orthographic data the program was supplied with information about the difference between vowels and consonants. Vowels were allowed as the output of state 4 (peak) only and consonants were allowed as the output of any other state (see B-matrix in figure 2.21). The task of the HMMs was to find out the best values of the state transitions in the model (A-matrix, Π-matrix and Ω-matrix) and discover what consonants are allowed in which state (B-matrix). The values of impossible links (A-matrix, Π-matrix and Ω-matrix) and probabilities of impossible productions of characters were initialized with zero. All other model parameters were initialized with a random value. We performed five experiments with this set-up. The results can be found in figure 2.23.

The HMMs needed on average 28.2 rounds to become stable. Thus they need less training rounds than the models without linguistic initialization (68.6 rounds, t(4)=2.3,

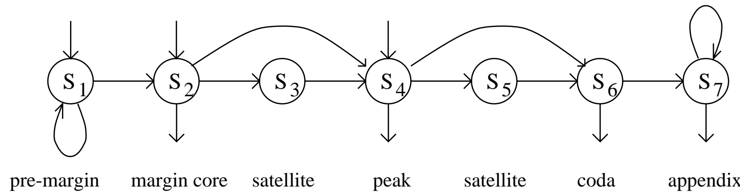


Figure 2.22: Initial bigram HMM for phonetic data. This is a graphical representation of the HMM parameters presented in figure 2.18. The two visible differences between this model and the original phonological model of Cairns and Feinstein are the extra link leaving state 2 (margin core) and the self-link at state 1 (pre-margin). These links are necessary to enable the model to handle the non-vowel interjections *pst* and *sst*, the abbreviation *s* and the consonant clusters in these strings.

nbr.	rounds	threshold	positive accepts	negative rejects
1.	33	$1.575*10^{-10}$	594	570
2.	48	$1.614*10^{-10}$	595	570
3.	17	$1.213*10^{-10}$	595	570
4.	10	$3.562*10^{-11}$	595	569
5.	33	$3.562*10^{-11}$	594	570
avg.	28.2±13.4		594.6±0.5 (99.1%)	569.8±0.4 (95.0%)

Figure 2.23: The results of five experiments with linguistically initialized bigram HMMs that processed phonetic data. After an average of 28 training rounds the HMMs accepted on average 595 words of the positive test data set (99.1%) and rejected 570 strings of the negative test data set (95.0%). When we removed strings with an acceptable phonetic transcription from the negative test data the HMMs obtained a rejection score of 99.1% on this test set.

$p<0.05$). The models accepted on average 594.6 words of the positive test data and this is exactly as many as the HMMs with random initialization (594.6 words, $t(4)=0.0$, $p>0.25$). They rejected more strings of the negative test data set (569.8) than the models of the previous section (565.6 strings, $t(4)=5.0$, $p<0.005$).

Twenty-nine negative strings were accepted by all five HMMs and five test words were rejected by all five HMMs. The five uniformly rejected test words were the same as in the experiments with randomly initialized HMMs. Again the words received the score zero because they contained bigrams that do not occur in the training data. Of the 29 accepted strings from the negative test data set 24 were in the set of 26 reasonable phonetic transcriptions (see previous section). The other five universally accepted strings were not acceptable: ephtsb [εptsp], hurwd [hœrwt], klolc [klɔlk],

sfhi [sfi:] and udsb [œtsp]. When we remove the 26 reasonable transcriptions from the data the HMM has rejected on average 568.6 of 574 negative strings (99.1%). This is an improvement compared with the average score of 98.3% achieved in the experiments with random initialization ($t(4)=4.9$, $p<0.005$).

5 Concluding remarks

It is possible to use Hidden Markov Models (HMMs) for building phonotactic models from a list of monosyllabic words. The resulting HMMs show recognition of language specific features such as vowel-consonant distinction. Unigram HMMs perform well in recognizing the difference between a set of positive test data and a set of negative test data. However, recognizing the difference between individual cases is a problem: the unigram HMMs often assign higher scores to incorrect data than to correct language data. The main problems are that standard unigram HMMs do not pay attention to the context of a character and that they exaggerate the influence of sequence length on sequence score. Providing the unigram HMMs with initial phonotactic knowledge shortens the training phase but it does not increase their performance after training.

Bigram HMMs with score correction for sequence length perform better. They misclassify few words and recognize a clear difference between positive test data and negative test data accepting on average 99.1% of a set of unseen correct words in phonetic representation (98.9% for orthographic data) while rejecting on average 98.3% of a set of impossible negative phonetic test sequences (91.0% for orthographic data). Providing initial linguistic knowledge to the HMMs caused a significant and large increase of the training speed for the phonetic data but only small increases in performance. The number of training rounds needed went down from an average of 68.6 to 28.2 for phonetic data ($t(4)=2.3$, $p<0.05$, see section 2.5 of chapter 1). The acceptance rate for the positive phonetic test data was the same for randomly and linguistically initialized HMMs (99.1%) but the rejection rate of the negative test data showed a small increase from 98.3% for the randomly initialized HMMs to 99.1% for the linguistically initialized HMMs ($t(4)=6.2$, $p<0.005$). HMMs with initial linguistic knowledge that processed orthographic data needed approximately the same number of training rounds as HMMs that were initialized randomly and performed worse. The phonological model that we used for initializing the HMMs was not suitable for orthographic data.

We can examine two of the three research questions mentioned in chapter 1. The phonetic data format seems to be better suitable for our problem. HMMs that processed orthographic data accepted as many positive test words as those that processed phonetic data but the latter rejected significantly more negative strings. Starting from basic phonotactic knowledge enables the HMMs to produce better models but the difference was only noticeable in the rejection rates of the negative data. Both HMMs that processed orthographic data as those that processed phonetic data accepted as many positive test data with and without initial knowledge but the rejection rates for negative data were significantly larger for the initialized HMMs.

The models that we have built in this chapter suffer from one of the problems that were mentioned in (Fudge et al. 1998) namely the presence of loan words in the data. These words have complicated the models. However, it is not easy to remove the loan words from the data set without making assumptions about the structure of the words. Therefore, we have chosen to leave these words in the training and test data. Fudge and Shockey have also recognized the problem of accepting incorrect strings when no context information is used. We have discussed this problem in section 3.4 with the example string *pajn* and solved the problem by using bigram HMMs instead of unigram HMMs. Our orthographic bigram models reject two of the three problematic words mentioned in (Fudge et al. 1998): *smlasr* and *sdring*. These strings should also be impossible in Dutch. The third word, *ssming*, is accepted by the models because the unusual onset *ss* occurs in the interjection *sst* which is present in the training data.

Fudge and Shockey also discuss the difference between accidental and systematic gaps in language patterns. Our approach to this is to regard any string that is not accepted by a model as a systematic gap and regard all strings that are accepted by a model but that are not present in the language as an accidental gap. Thus the difference between accidental and systematic gaps has become a model-dependent difference.

The models derived in this chapter satisfy two of the five properties Mark Ellison outlined in his thesis (Ellison 1992). They are cipher-independent (independent of the symbols chosen for the phonemes) and language-independent (they make no initial assumptions specific for a certain language), but their internal structure is neither accessible nor linguistically meaningful. The HMMs also fail to satisfy Ellison's first property (operation in isolation) because they receive preprocessed language input: monosyllabic words. The removal of the monosyllabicity constraint we put on our training data is an interesting topic for future work.

Chapter 3

Connectionist Learning

Connectionist approaches have become increasingly popular in the field of natural language processing. These approaches use built-in learning mechanisms. Therefore they should be of interest to research in language acquisition and language learning as well. In this chapter we will examine the promises connectionist approaches, also called neural network approaches, have for natural language learning. We will concentrate on the language learning task which is central in this thesis: the acquisition of phonotactic structure. In the first section we will give an introduction to a class of neural networks: feed-forward networks. In the second section we will introduce the network we have chosen to experiment with: the Simple Recurrent Network (SRN) developed by Jeffrey Elman. The third section covers the acquisition experiments we have performed with this network. This set of experiments will not give us optimal results. We will use the fourth section for laying bare the problem that is the cause of these suboptimal results and presenting a method for restructuring the input data that could enable the networks to obtain better results. In the final section we will present a summary of this chapter with some concluding remarks.[1]

1 Feed-forward networks

In this section we will give an introduction to a specific type of neural network: the feed-forward network. We will start with a general description of this network type. After that we will show how feed-forward networks learn. We will conclude with showing how non-numeric data can be encoded in a feed-forward network.

[1] Some parts of this chapter have earlier been published in (Tjong Kim Sang 1995).

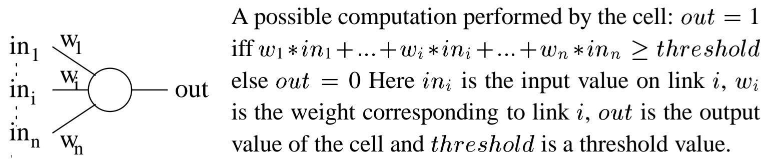


Figure 3.1: An example of a cell in a neural network and the computation it performs.

1.1 General description of feed-forward networks

Artificial neural networks have been inspired by biological neural networks. They consist of cells which are connected to each other by weighted links. The weights of the links determine the function the network performs. Artificial neural networks usually contain a learning function which adapts the value of the weights to optimize the network's performance on a task.

Connectionist or neural networks have often been compared with rule-based models in cognitive science literature. Researchers have posed that neural networks are more suitable for modeling cognitive tasks such as language processing because they are robust and because they are biologically plausible. Robustness, the ability to deal with noisy data, is an interesting property. It will enable the networks to handle training data with errors. The errors will be infrequent and thus have little influence on the final model. Neural networks share the robustness feature with statistical models. However, robustness is not a feature that is often contributed to rule-based models.

There are many different types of neural networks available. A quick glance in a neural network introductory book like (Wasserman 1989) will reveal that there are about a dozen main types of neural networks and the main types have many variants. Some of these networks are biologically or psychologically plausible and others are just parallel cell models with smart learning algorithms. The cells used in the second network type have a far more simple structure than human brain cells (Zeidenberg 1990). We are interested in solving our problem, the acquisition of phonotactic structure, as well as possible. We feel that using biologically or psychologically inspired artificial neural networks is an interesting approach to solve the problem However, we refrain from committing ourselves to using only biologically or psychologically inspired artificial neural networks.

We have described a neural network as a collection of cells which are connected to each other with weighted links. The cells use these links to send signals to each other. The links are one-way links: signals can only travel through a link in one direction. Furthermore, signals are numbers between zero and one. The cells perform a simple function: they multiply incoming signals with the weight of the link the signals are on and compute the sum of all multiplications. When this result is larger than or equal to a threshold value, the cell will put a signal 1.0 at its output link otherwise it will put a signal 0.0 on its output link. The function that the cell performs to compute the output signal from the input signals is called the ACTIVATION FUNCTION.

in_1	in_2	$XOR(in_1, in_2)$
0	0	0
0	1	1
1	0	1
1	1	0

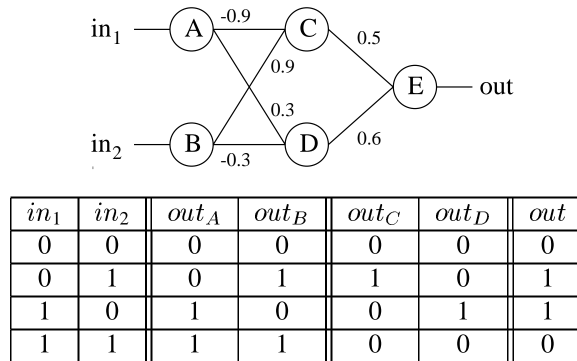
Figure 3.2: The XOR-function. It will produce a 1 if and only if exactly one of the two binary input parameters is equal to 1. Otherwise the function will produce 0.

When we connect these cells with each other and give cells input and output connections with the outside world, we have created a neural network. After presenting the network a set of input signals, it will produce a set of output signals. Sets of signals are also called patterns So it is capable of converting an input pattern to an output pattern. As an example we will design a network that computes the XOR-function (see figure 1.1).

We can try to build a network that performs this function by starting from a random collection of cells with a random number of links between them. However, we will to divide the network in groups of cells so that we can use the learning algorithm which will be described in the next section. We will call these cell groups layers. We will number the layers and impose a hierarchy on them. Cells in the first layer receive input from outside and send their output to cells in the second layer. Cells in the second layer receive input from the first layer and send their output to cells in the third layer, and so on. Cells in the final layer send output to the outside world. So the data will first enter the first layer, then it will move to the second layer then to the third and so on until it reaches the final layer via which it is sent to the outside world. Because the data flows from the back to the front of the network this network is called a feed-forward network.

Figure 3.3 shows a three-layer network which is able to compute the XOR-function, a binary function with two input parameters and one output parameter which is only equal to one if exactly one of the input parameters is equal to one. The cells in the network all have a threshold value of 0.1. The two input cells do not change their input: they simply pass it on to the second layer. The table presents an insight into the information flow in the network. For example, the [0,1] pattern (in_1=0 and $in_2 = 1$) will be passed to the second layer and result in a [1,0] output signal of the second layer. This will result in a [1] signal as output of the network.

This network prefectly computes the XOR-function because it contains the correct weights to do this. But now the question is: how do we get a network to compute an arbitrary function? In other words: how do we find weights that enable a network to compute a specific function? We will deal with this question in the next section.



in_1	in_2	out_A	out_B	out_C	out_D	out
0	0	0	0	0	0	0
0	1	0	1	1	0	1
1	0	1	0	0	1	1
1	1	1	1	0	0	0

Figure 3.3: A network that computes the XOR-function. Signals travel through the network from left to right. These signals have been represented by binary numbers in the table. The numbers next to the links in the network are the weights of the links. All cells have a threshold value of 0.1.

1.2 Learning in a feed-forward network

We want a feed-forward network to perform a task which is computing a specific function. This can be done by presenting the network the input and the corresponding output of the function and making it learn a weight configuration that can be used for simulating the function. This kind of learning is called SUPERVISED LEARNING: the learner is provided with both input and output for a particular task. In unsupervised learning the learner will only receive the task input (Zeidenberg 1990).

One of the learning algorithms which can be used for deriving a network weight configuration is the backpropagation algorithm described in (Rumelhart et al. 1986). In this algorithm the output of the network for the training patterns is compared with the required output. The difference between these two, the error, is used for improving the current weights. First the error of the network output is used for computing the weights of the links to the last layer, then for the weights to the next to last layer and so on. The error is propagated from the output layer of the network back to the input layer, hence the name of the algorithm 'backpropagation'. This learning algorithm is not biologically plausible. However it is still the most frequently used connectionist learning algorithm because it will produce a reasonable model for most data sets (Rumelhart et al. 1986).

Let us look at the backpropagation algorithm in more detail (Rumelhart et al. 1986). First we define 6 variables:

$target_{c,l}$	the target output value of cell c of layer l
$out_{c,l}$	the actual output value of cell c of layer l
$\delta_{c,l}$	the error in the output of cell c of layer l ($target_{c,l}$-$out_{c,l}$)
$w_{c_1c_2,l}$	the weight of the link between cell c_1 in layer l-1 and cell c_2 in layer l
$\Delta w_{c_1c_2,l}$	the size of the change of $w_{c_1c_2,l}$ as computed by the algorithm
η	the learning rate, a parameter of the algorithm

Here layer 1 will be the input layer, layer 2 will be the first hidden layer, layer 3 will be the second hidden layer if present and so on. When there are n hidden layers, layer n+2 will be the output layer. We will use one hidden layer and therefore in our networks layer 3 will be the output layer. The first step of the backpropagation algorithm consists of computing the $\delta_{c,l}$ values for the cells in the output layers:

$$\delta_{c,l} = (target_{c,l} - out_{c,l}) * (1 - out_{c,l}) * out_{c,l} \tag{3.1}$$

The simplest formula for computing a $\delta_{c,l}$ would have been $\delta_{c,l} = (target_{c,l} - out_{c,l})$ in which case the error would have been equal to the difference between the target output value and the actual output value. However, for the correctness proof of the backpropagation algorithm it is convenient to use the complex error equation that has been displayed. In this complex equation the error is the product of the simple error function and the derivative of the activation function with respect to the input.[2]

When we know the error values, we can use them for computing the weight change values for the output layer $\Delta w_{c_1c_2,l}$ and these can in turn be used for computing the new values of the weights of the links between the hidden layer cells and the output layer cells:

$$\Delta w_{c_1c_2,l} = \eta * \delta_{c_2,l} * out_{c_1,(l-1)} \tag{3.2}$$

$$new\ w_{c_1c_2,l} = old\ w_{c_1c_2,l} + \Delta w_{c_1c_2,l} \tag{3.3}$$

The change of a weight of a link $\Delta w_{c_1c_2,l}$ is proportional to the network parameter, learning rate η, the error value of the cell at the end of the link $\delta_{c_2,l}$ and the value of the signal on the link $out_{c_1,(l-1)}$. The equation contains an input signal value because multiple input signals determine the output value of a cell and the weights corresponding with the signals that contribute most to this output value should receive the largest increase or decrease. The new value of the weight will be the old value increased with the change.

The learning rate η is a model parameter that backpropagation has in common with hill-climbing algorithms (Rich et al. 1991). These algorithms can in general be used for solving the problem of a walker attempting to find the highest spot in a misty mountain area. In the case of backpropagation, a set of correct weights would be a representation of the location of the highest point of the mountain. Then each $\Delta w_{c_1c_2,l}$

[2] We are using the activation function defined in (Rumelhart et al. 1986), equation (15): $out_{c,l} = (1 + e^{in_{c,l}+\theta_{c,l}})^-1$ where $in_{c,l} = \sum_{c_1} w_{c_1c,l-1} * out_{c_1,l-1}$ and $\theta_{c,l}$ is the threshold value for cell c of layer l. The derivative of this function with respect to the input $in_{c,l}$ is $(1 - out_{c,l}) * out_{c,l}$).

corresponds to one step of the walker. A large value for η increases the size of the steps that the walker makes and enables him to reach the target faster. However, larger η values also introduce the danger that the walker might step over the target location and never be able to reach it (we assume that the walker is able to make steps of many kilometers). In terms of finding the correct weight configuration: a large η value enables backpropagation to move faster to the target weight configuration but it also increases the chance of never reaching this target.

By using equations 3.1, 3.2 and 3.3 we will be able to compute new values for the weights of the links between the hidden layer and the output layer. The latter two equations will also be used for the computation of the new values for the weights of the other links. Equation 3.1, however, cannot be used for that purpose because it contains the term $target_{c,l}$. While we know the correct output value for the output layer, it is impossible to tell what the correct output value for the cells in the hidden layer needs to be. Therefore we will use a different equation for computing $\delta_{c,l}$ for non-output cells:

$$\delta_{c_1,l} = (\sum_{c_2} \delta_{c_2,(l+1)} * w_{c_1 c_2,(l+1)}) * (1 - out_{c_1,l}) * out_{c_1,l} \tag{3.4}$$

In this equation we have approximated $(target_{c_1,l} - out_{c_1,l})$ with the sum of the weights of the output links of cell c_1 ($w_{c_1 c_2,(l+1)}$) multiplied with the error computed for the cell that the link provides input for $\delta_{c_2,(l+1)}$. The δ values computed for layer $l+1$ will be used for computing the δ values for layer l. This operation can be performed for any number of network layers.

With the equations 3.1, 3.2, 3.3 and 3.4 we are able to update all the weights of the network. (Rumelhart et al. 1986) have introduced an improved version of equation 3.2, one which will increase the learning speed with minimal chance of the algorithm becoming instable[3]:

$$\Delta w_{c_1 c_2,l}(t) = \eta * \delta_{c_2,l} * out_{c_1,(l-1)}(t) + \alpha * (\Delta w_{c_1 c_2,l}(t-1)) \tag{3.5}$$

The weight change as defined in equation 3.2 is now increased with the previous value of the weight change $\Delta w_{c_1 c_2,l}(t-1)$ multiplied with the momentum parameter α which is a value between 0 and 1. The idea behind this is that the process of reaching the correct weight values usually consists of a large number of small steps in the same direction. By adding to each step a portion of the value of the previous step, successive steps in the same direction will increase the size of the steps and thus enable the algorithm to reach the target weight configuration faster.

The backpropagation algorithm will have to perform a number of weight modifications before it reaches a good weight configuration. The number of modifications that are necessary will be determined by the values of the algorithm parameters η and α. We would like the algorithm to arrive at a correct network configuration as soon

[3]A network is instable when it is unable to converge to a more or less constant model for the training data in a finite amount of training rounds.

as possible and therefore we are interested in finding optimal values for these two parameters. Unfortunately, the optimal values of η and α are dependent on the task to be learned and the network initialization. Usually finding good values for these two algorithm parameters is a matter of trial and error.

In our experiments with feed-forward networks we will use an error function for measuring the performance of the network for a certain task. The error function takes all cell output values of all output patterns, subtracts from them the desired output values and adds the squares of these subtractions together:

$$E = \sum_{pat} \sum_{cell} (desired\ output_{pattern,cell} - actual\ output_{pattern,cell})^2 \quad (3.6)$$

The result, the TOTAL SUM SQUARED ERROR, is an indication of the performance of the network. A small error value indicates that the network is performing the task well.

Now training the network is a four step process:

1. Present an input pattern to the network and make the network compute an output pattern.
2. Compare the actual output pattern of the network with the target output values and use the equations 3.1, 3.4, 3.5 and 3.3 for computing new values of the weights.
3. Repeat steps 1 and 2 for all patterns.
4. Repeat steps 1, 2 and 3 until the total sum squared error has dropped below a certain threshold or the number of times that these steps have been performed has reached some predefined maximum.

In an alternative schedule step 2 will only use the equations 3.1, 3.4 and 3.5 for computing the weight updates. These will be stored in some buffer and the actual update of the weights will be made in step 3 when all patterns have been processed.

1.3 Representing non-numeric data in a neural network

In the example in which we computed the XOR-function with a neural network, representing data was not difficult because the data was numeric and the network processed numeric data. But in language we want to process sounds and symbols and therefore we have to find out a way to represent non-numeric data in a neural network. When we consider using four characters *a*, *b*, *c* and *d* in a network, there are three basic ways for representing them in a network.

First we can represent the characters by using one signal and assigning numeric values to the characters, for example: *a* = 0.0, *b* = 0.3, *c* = 0.6 and *d* = 0.9. However, this approach will fail because imperfect data cannot be interpreted unambiguously.

If the network does not know which of two outputs is correct, it typically produces an intermediate value. A draw between *a* and *c* can result in an output value of 0.3, which actually is the representation of *b*. It is impossible to tell if a 0.3 output is the result of the network choosing for *b* or that the network cannot choose between *a* and *c*. The network output can not be analyzed because the representation of characters is inappropriate.

A second way to represent the characters is to use two binary signals: *a* = [0,0], *b* = [0,1], *c* = [1,0], *d* = [1,1]. However, this representation suffers from the same problem as the previous one: errors can occur when the network output needs to be interpreted. For example: if the network cannot choose between *b* and *c* it might produce a [0.5,0.5] output. However, this output can also be interpreted as the intermediate pattern for *a* and *d*. Furthermore, binary representations can invoke unintended similarities between the symbols. It is possible to use five bits for representing twenty six characters and use [0,0,0,0,0] (0) for *a* up to [1,1,0,0,1] (25) for *z*. It that case the representations for *e*, [0,0,1,0,0], and *m*, [0,1,1,0,0] are similar. The network will use this artificial similarity during processing. We might not want it to do that.

A third way to represent the characters in by using the localist mapping described in (Finch 1993) among others. Then every character will be linked to a particular cell in the representation and only one cell in the representation can contain a one: *a* = [1,0,0,0], *b* = [0,1,0,0], *c* = [0,0,1,0] and *d* = [0,0,0,1]. In this case we need four signals. This representation does not have the problem of the two previous ones. If the network produces an intermediate value the candidates that it suggests can unambiguously be pointed at, for example: if the network would generate a [0.5,0.5,0,0] output then we know that it had difficulty to choose between *a* and *b*. Furthermore all representation patterns are equally alike so the network cannot recognize non-intended similarities from the representations. However, the price we pay for this is that this way of representing characters requires more cells than the previous two. More cells means more links and more weights and, because computations will be performed for all weights during training, more cells means longer training times.

When a lot of symbols have to be processed, using the localist mapping can prove to be time-consuming during the training phase. In that case a balance has to be found between the two problems of large representations and non-intended similarities between patterns. A balance can be found by choosing some intermediate representation size and making the network itself find out meaningful representations of that size. (Blank et al. 1992), (Elman 1990) and (Miikkulainen et al. 1988) show three different methods for making networks build fixed length representations for symbols. The representations built by the network reflect the task the network learned to perform with the symbols. So a different network task will result in different representations.

2 The Simple Recurrent Network (SRN)

In this section we will present the network we will use in our phonotactic experiments: the Simple Recurrent Network (SRN). We will start with a general description and af-

ter that we will show how SRNs learn. The section will be concluded with a summary of language experiments performed with SRNs by others.

2.1 General description of SRNs

The three-layer feed-forward network we have presented in the previous section cannot perform all the tasks we would like to apply it to. An example of a task in which the network cannot achieve a score that is higher than 50% is predicting the next bit in a bit list like:

1 0 1 0 0 0 0 1 1 1 1 0 1 0 1 ...

Only one bit of the list is available at one time point. This bit list first appeared in (Elman 1990). At first sight the elements of the list seems to be chosen randomly and correctly predicting bits of the list seems impossible. However when we take a closer look at it, it turns out to be possible to discover three-bit patterns in the list. The first two bits of each pattern are random but the third is the XOR of the first two. In order to be able to predict this third bit correctly, the network must know the values of the previous two. However, as we said only one bit of the list is available at each moment and the network does not have memory to store the previous bit. Therefore the feed-forward network cannot predict the third bit of each three-bit pattern correctly. The network cannot do any better than guessing future bits which will result in a score of 50%.

The only way to improve the performance of the network is to give it memory. We can use a buffer to store input and only process the input when it is complete, an approach which has been chosen in (Weijters et al. 1990). To solve this problem we will need to a buffer size of one pattern. However, for general problems in which memory is required we might not know how large the buffer size needs to be. If we choose a buffer size that is too small, the network will not be able to solve the problem. Therefore we have not chosen this approach.

There are two other standard ways of adding memory to a three-layer feed-forward network. Both consist of adding a feedback loop to the network which enables it to store and use context information for the input sequence.

The first way of giving the network memory is by feeding the output of the network back to the input. This approach was developed by (Jordan 1986) and the resulting network has become known as the Jordan network. The input layer of the network is extended with the same number of cells as the output layer. These extra cells, the CONTEXT CELLS, contain a copy of the previous output layer activations. Like the input layer cells, they are connected with all cells in the hidden layer. Every output cell is connected with a context cell. Furthermore, each context cell is connected with itself and with every other context cell (see figure 3.4). The weights of the backward and the inter-context cell links are equal to one and these weights cannot be changed. Signals only flow through these links when the input of the network is updated.

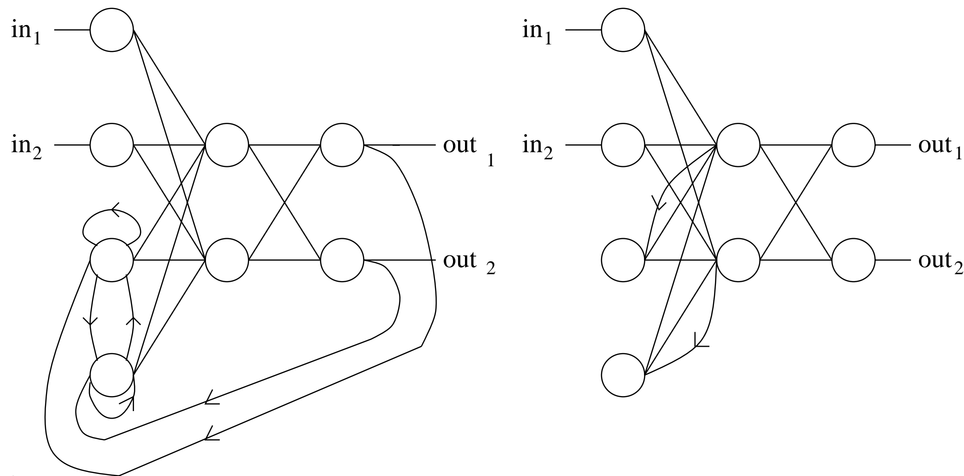


Figure 3.4: A Jordan network (left) and a Simple Recurrent Network (right). Except when arrows indicate otherwise, all connections are forward connections. The two context cells at the bottom left of each network make the difference between these two networks and the networks we have discussed in the previous section. These context cells implement the memory of the networks.

The second way of adding memory to the network is by feeding the output of the hidden layer back to the input layer. (Elman 1990) first used this approach and he named the network Simple Recurrent Network (SRN). Now the input layer is extended with as many context cells as the hidden layer. The other aspects of the network are the same as in the Jordan network apart from the internal context layer links which are present in the Jordan network but do not exist in the SRN (see figure 3.4).

Both networks are capable of performing the XOR-sequence task correctly. An example of the output of an SRN after training can be found in figure 3.5. The average error of the network for the third, the sixth and the ninth bits in the test list is 0.40 or lower but for other bits it is about 0.50. While this network does not perform perfectly, it seems to have recognized the structure of the bit lists.

For other tasks performance of these two network types can be different: sometimes the Jordan network performs better and sometimes the SRN. The Jordan network trains faster in a supervised task because the correct input values of the context cells are known during training (context cell input = previous network output in Jordan networks). The correct input values of the context cells in an SRN are unknown during training (context cell input = hidden layer output in SRNs). However, the hidden layer activation values may contain an interesting representation of features of the symbols and the task in which they are used (Elman 1990). Feeding back these activations can aid the network in performing well. This last feature is the prime reason for us for

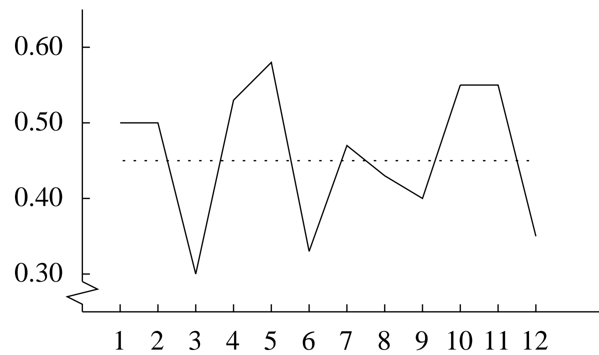


Figure 3.5: Average error in the SRN output for the XOR task for 12 consecutive inputs averaged over 10 trials. The network had to predict the next bit of a bit list of which each third bit was the XOR of the previous two. The overall average error is 0.45 (dotted line) but the error for the bits 3, 6, 9 and 12 was smaller (0.35 on average) which indicates that the network has recognized the structure of the bit list.

choosing for the SRN for our experiments. With this choice we join previous connectionist language research like (Elman 1990) and (Cleeremans 1993) which both use SRNs to model sequential language processing. Jordan networks are not used as often; an example of a Jordan networks application is the work of (Todd 1989) in music generation.

2.2 Learning in SRNs

An SRN can be trained by using the same equations and training schedule as was used for the feed-forward networks in section 1.2. The only differences are that some weights will not be changed during training (the weights of the backward links) and a part of the input data will be obtained from the hidden layer output for the previous pattern (the context cell input values).

Again training the network is a four step process:

1. Present an input pattern to the network and make the network compute an output pattern. The context cell input values are equal to the output values of the hidden layer for the previous pattern. If no previous pattern exists then the context cell input values will be equal to 0.

2. Compare the actual output pattern of the network with the target output values and use the equations 3.1, 3.4, 3.5 and 3.3 of the backpropagation algorithm for computing new values of the weights. The backward links should not be changed.

3. Repeat steps 1 and 2 for all patterns.
4. Repeat steps 1, 2 and 3 until the total sum squared error has dropped below a certain threshold or the number of times that these steps have been performed has reached some predefined maximum.

2.3 Using SRNs for language experiments

SRNs have been used successfully in different experiments. (Elman 1990) has used an SRN for analyzing simple sentences. From a small grammar he generated 10,000 two- and three-word sentences. The sentences were concatenated in a long sequence and an SRN was trained to predict the next word in the sequence. The network only had available the current word and a previous hidden layer activation which can be seen as a representation for the previous words in the sequence. The SRN cannot perform this task perfectly because the order of the words in the sequence is non-deterministic and the network is too small to memorize the complete sequence.

However, Elman was not interested in the output of the network. He was interested in the activation patterns which were formed in the hidden layer. Elman discovered that the average activation at the hidden layer for a word presented at the input reflected the way the word was used in the sentences. A cluster analysis of these average hidden representations divided the words in two groups: the verbs and the nouns. Within these groups some subgroups could be found: animals, humans, food, breakable objects and different verb types (Elman 1990). So the network developed internal representations for words which reflect their syntactic and semantic properties.

In another experiment Axel Cleeremans, David Servan-Schreiber and James McClelland (Cleeremans 1993) (Cleeremans et al. 1989) (Servan-Schreiber et al. 1991). have trained a network to recognize strings which were generated using a small grammar that was originally used in (Reber 1976) (see figure 3.6). Cleeremans et al. trained an SRN to predict the next character in a sequence of 60,000 strings which were randomly generated by the grammar. Again this task was non-deterministic and the size of the network was too small to memorize the complete sequence. So the network could not perform this task without making errors. For Cleeremans et al. it was sufficient that the network indicated in its output what characters are a valid successor of the sequence. They represented characters using the localist mapping and their aim was to make the network output at least 0.3 in the cells that correspond to valid successors. So when the output of the network is something like shown in figure 3.7, then valid successors are *S* and *X* because these have received an output value higher than 0.3. The network accepted a string if it considerated all characters of the string as valid characters in their context.

Cleeremans et al. have tested their network with 20,000 strings generated by the Reber grammar. For all characters in these strings the network output in the corresponding cells was 0.3 or higher. So the network accepted 100% of the grammatical strings. After this Cleeremans et al. fed their network 130,000 random strings built from the same characters. This time the network accepted 0.2% of the strings. It

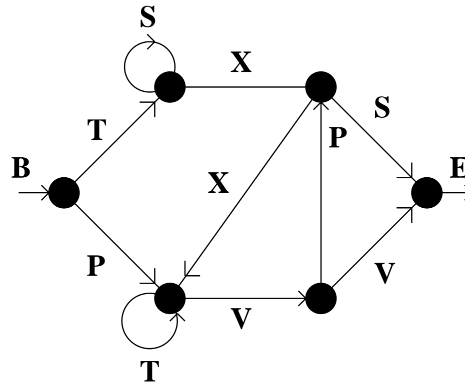


Figure 3.6: Finite state network representing the Reber grammar. Valid strings produced by this grammar will always start with *B* and end with *E*. Between these characters there will a string containing *T*'s, *S*'s, *X*'s, *V*'s and *P*'s. The finite state network indicates which characters are possible successors of a substring. For example: the substring *BT* can be followed by either *S* or *X*.

B	T	S	X	V	P	E
0.0	0.0	0.4	0.5	0.0	0.1	0.0

Figure 3.7: Possible network output of an SRN trained on the Reber grammar after presenting substring *BT* to the network. A number below a character indicates the output value of the network output cell corresponding with that character. The SRN attempts to predict the successor of the string. It considers *S* and *X* as valid successors of *BT* because the output values corresponding to these two characters are larger than 0.3.

turned out that all accepted strings were grammatical strings. All other strings were invalid according to the grammar. Thus the network separated perfectly the grammatical strings from the non-grammatical strings.

The results obtained by Cleeremans et al. are excellent. Their problem was related to ours so we hope to obtain results that are similar to theirs when we apply SRNs to our phonotactic data.

3 Experiments with SRNs

In this section we will describe the experiments we have performed with SRNs. We will start with a presentation of the general set-up of the experiments. After that we

will discuss a small experiment which was used for determining optimal network parameters. The next two sections will describe the experiments that we have performed with orthographic data: one with random initialization and one with additional linguistic information. The linguistic information was supplied to the network by supplying the training data in an order based on the complexity of the training strings.

3.1 General experiment set-up

We will use the experiment of (Cleeremans 1993) with the Reber grammar as an example for the set-up of our own experiments involving the acquisition of phonotactic structure. We will use Simple Recurrent Networks (SRNs) which will process a list of words. The words will be presented to the networks character by character. The SRNs have no access to previous or later characters. They will have to use their internal memory for storing a representation of the previous characters. The SRNs will be trained to predict the next character of a word.

Like Cleeremans et al. we will use the localist mapping to represent characters. So in the SRN input and output layers every cell will correspond to one character. We will be using the localist mapping and this means that input and output patterns will contain 29 cells for the complete orthographic data. So the number of input and output cells for networks that process orthographic data will be 29. We will work with different hidden layer sizes: 3, 4, 10 and 21 cells. The number of cells in the context layer will be equal to the number of cells in the hidden layer.

We will adopt Cleeremans et al.'s measurement definition, the way the scores of characters and words are computed. The WORD SCORE is the value that is assigned by a network to the word. If the word score is high we will assume that the word is probable according to the training data. The CHARACTER SCORE is the value assigned by a network to the character in a certain context. If the character score in a certain context is high we will assume that the character is probable in that context. In the work of Cleeremans et al. these two scores are defined as follows:

> **Measure 1**
>
> The character score of character c in the context of some preceding substring s is the output value of the output cell corresponding to c of a network after the network has processed s.
>
> The word score of word w consisting of the characters $c_1 ... c_n$ is equal to the lowest value assigned by a network to any of these characters at their specific positions during the processing of w.

Note that these scores can only be interpreted in a reasonable way if they are computed by the same network. It makes no sense to compare scores of words that are computed by different networks. Because of the random initialization of SRNs before learning, two SRNs trained on the same problem with the same data may well assign different

scores to the same word. We are only interested in the relation between scores of words computed by the same network.

We will compare the performance of measure 1 with two other measures. In our second measure the character scores will be computed in the same way as in measure 1 but the word scores will be equal to the product of the character scores:

> **Measure 2**
>
> The character score is computed in the same way as in measure 1.
>
> The word score of word *w* consisting of the characters $c_1...c_n$ is equal to the product of the scores assigned by the network to these characters at their specific positions during the processing of *w*.

In our third measure we will also multiply the character scores to get the word score but this time we will use the Euclidean distance between actual network output and target output as character score. This means that we compute character scores by comparing all network output values with the target values instead of only looking at one cell:

> **Measure 3**
>
> The character score of character *c* in the context of some preceding substring *s* is 1 minus the normalized Euclidean distance between the target output pattern of a network after processing *s* and the actual output pattern of the network after processing *s*:
>
> $$score_c = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}(target_i(s) - output_i(s))^2}$$
>
> in which $target_i(s)$ is the target output pattern of cell *i* and $output_i(s)$ is the actual output pattern.
>
> The word score is computed in the same way as in measure 2.

Since the patterns consist of values between 0 and 1, the largest difference sum between these two vectors is n so we divide the sum of products by n in order to obtain a value between 0 and 1.[4] This value is equal to 0 for a perfect match and equal to 1 for a non-match so we subtract it from 1 to obtain similar values as in the previous measures.

We have explained in the description of our work with Hidden Markov Models that it is necessary to append an end-of-word character to the words in our training corpus. We will also add an end-of-word character to the words we are processing in our SRN experiments. The end-of-word character is necessary to enable the network to recognize the start of a new word. We will present the words to the network character by character in one long list. If the network is unable to detect the start of a new

[4] Actually this maximum difference is smaller than n because the target vector has length one and we have observed that the output vectors of the network after training have about unit length. The Euclidean distance between two positive vectors of unit length has a maximal value of $\sqrt{2}$.

word then the scores of the characters of a word will be influenced by the characters of previous words.

We want the scores of the characters of a word to be influenced only by the characters of the same word. Therefore we have put the end-of-word characters as separators between words and made the networks reset their internal memory, the context cells, after processing such an end-of-word character. We have used the name end-of-word character rather than start-of-word character or word-separator to keep a consistent usage of terminology with the Hidden Markov Model chapter. Adding these characters to the data means adapting that task to the learning algorithm. We are not favoring that approach but in this particular case adding end-of-word characters is the best way to enable the networks to interpret the data in a reasonable way.

Our goal will be to obtain a Simple Recurrent Network which predicts the next character of a word given the previous characters. Because this task is nondeterministic and the network is too small to memorize all words, it cannot perform this task perfectly. We will be satisfied if, by using our measures, we can determine that the network has discovered the difference between words which are valid and words which are invalid in a certain language. The network will assign scores to all words. We will accept words which have a score above a certain threshold and reject all other words. The threshold value will be equal to the smallest word score occurring in the training set.

The parameters of the network will be initialized with the values mentioned in (Cleeremans 1993) (page 209): learning rate η between 0.01 and 0.02 and momentum α between 0.5 and 0.9. The number of cells used in the hidden layer influences the performance of the network. However, it is hard to find out the best size of the hidden layer. Cleeremans et al. have chosen a hidden layer size of three because the grammar they have trained their network with can be represented with a finite state network with a minimal number of six states. These six states correspond with the vectors which can appear at the hidden layer. If we assume that these vectors are binary, we need 3 cells to be able to represent 6 patterns ($2^3 > 6 > 2^2$). In our HMM experiments we have used HMMs with eight internal states. Therefore we will also start with a hidden layer of three cells.

3.2 Finding network parameters with restricted data

In our first experiment we will try to find out if our network set-up is correct. We are not interested in a network covering all data yet, so we start with training the SRN with part of the data: 184 words in orthographic representation starting with the character *t*. We trained the network 10,000 rounds with η=0.02 and $\alpha = 0.5$ and examined its performance every 1000 rounds. The network was tested with two word lists: one list consisting of 11 invalid t-words and one list of 16 valid Dutch t-words that were not present in the training corpus. All words receive a score. A word is rejected if it receives a score smaller than the smallest score that occurred in the training set. The target of the network is to reject all 11 negative words and accept all

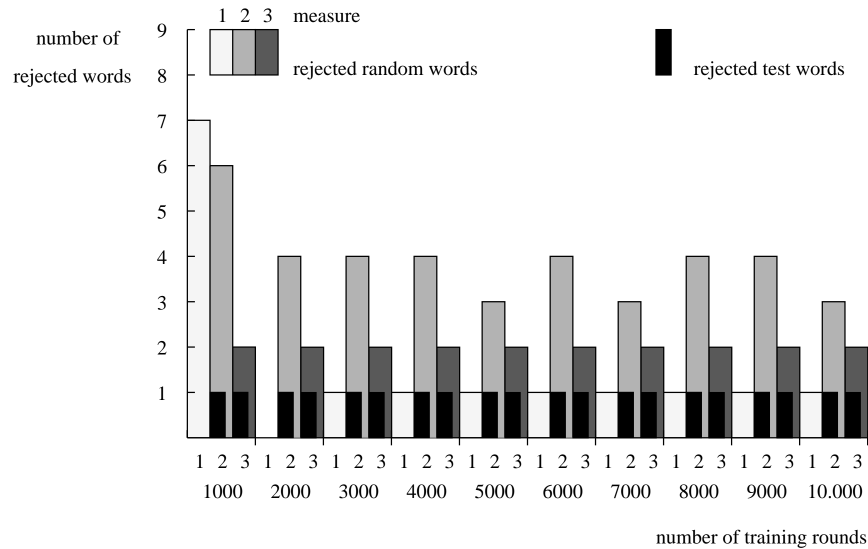


Figure 3.8: The performance of an SRN with three hidden cells for orthographically represented words starting with *t* using measures 1, 2 and 3. The target of the network was to reject 11 invalid t-words and accept all positive test words. The network reached the best performance after 1000 rounds of training. From 2000 rounds onwards the performance stabilized at a lower level.

16 test words. The behavior of the network is shown in figure 3.8.

The best results were achieved by measure 1 after 1000 rounds. At that point this measure rejected seven of the 11 negative words and accepted all positive words. The other two measures also performed best after 1000 rounds. At that point measure 2 rejected six negative words and one positive word and measure 3, which reached the same performance for all test points, rejected two negative words and one positive word. The performances of measure 1 and 2 were worse for all other test points: they accepted equally many positive words while rejecting fewer negative words.

Two observations can be made from the results shown in figure 2. First of all, the fact that the performance of measure 1 and 2 decreases after longer training is an example of OVERTRAINING. This is common behavior for SRNs: when they are trained too long they adapt themselves better to the training data and their generalization capabilities decrease. This will result in a poorer performance on patterns they have never seen.

We will attempt to avoid the overtraining problem by using a smaller learning rate η in our future experiments: 0.01 instead of 0.02. By decreasing the learning rate we will decrease the speed with which the network will approach the optimal set of weights. Another modification we will make in our future SRN experiments is that

word length	number of words	average scores for measure 2	factors for measure 2	average scores for measure 3	factors for measure 3
2	1	$4.432*10^{-4}$	-	$6.460*10^{-1}$	-
3	26	$7.401*10^{-4}$	$5.988*10^{-1}$	$5.532*10^{-1}$	1.168
4	81	$4.054*10^{-4}$	1.826	$4.903*10^{-1}$	1.128
5	61	$1.941*10^{-4}$	2.089	$4.219*10^{-1}$	1.162
6	13	$7.663*10^{-5}$	2.532	$3.537*10^{-1}$	1.193
7	2	$2.737*10^{-7}$	$2.800*10^{2}$	$2.694*10^{-1}$	1.313

Figure 3.9: The average scores for the orthographically represented training t-words of similar length (excluding the end-of-word character). The factors indicate average score for length divided by the average score for length n-1. Apart for the unique two character word the average score for words with length n+1 is always smaller than the average for words of length n.

we will test the performance of the network more frequently: every 50 rounds instead of every 1000 rounds. According to the results we have achieved here the network performed best after 1000 rounds but it might have been performing better after 550 rounds. By testing the performance of the network more often, we hope to determine the best performance point more precisely.

The question, of course, is when the training process of the SRN should be stopped. We will follow the solution the general literature has proposed for this and stop training when the total sum squared error (see section 1.2 of this chapter) stabilizes. We will stop training when the total sum squared error at a test point remains within a 1% distance of the error at the previous test point.

The second observation we can make from figure 3.8 is that measures 2 and 3 perform worse than measure 1. A possible explanation for this fact is that measure 2 and 3 compute word scores by multiplying character scores with each other. The character scores are values between 0 and 1. Almost all multiplications will make the word score smaller and this causes long words to have a smaller average score than short words.

The average word scores are shown in figure 3.9: in almost all cases words with length n receive a smaller average score than words with length $n + 1$. This is also the fact for the negative and the positive test set. The result of this is that a short negative word might receive a higher score than a long positive word which can result in the negative word being accepted and the positive word being rejected. We want to avoid this.

A possible solution to this problem is multiplying the scores with a value that depends on the length of the score. The longer the words are the larger the multiplication factor should be. The aim of this extra computation is making the word scores less dependent on the length of the words. We call this computation SCORE CORRECTION FOR LENGTH.

The question now is what the size of the multiplication factor should be. We chose as factor the quotient of the word average of length 4 and the word average of length 5 to the power of word length:

$$new\ word\ score = old\ word\ score * factor^{word\ length} \tag{3.7}$$

$$length\ correction\ factor = \frac{average\ score\ word\ length\ 4}{average\ score\ word\ length\ 5} \tag{3.8}$$

It seems reasonable to choose a value connected with the word lengths 4 and 5 because these are the two most frequently occurring lengths in this set of words. The quotients of the averages for word length *n* and word length *n-1* are more or less the same, so introducing a power function here seems reasonable as well. In this particular experiment we would have multiplied words with $2.089^{word\ length}$ for measure 2 and with $1.162^{word\ length}$ when we are using measure 3. No score correction for length is necessary for measure 1 because in this measure the word score dependents on one character score only.

We have repeated this experiment with words starting with *t* with the improved set-up: η is 0.01, network performance is tested after every 50 training rounds and score correction for length. After 350 rounds the error of the network stabilized. After training the network performed best with measure 3 rejecting seven of the eleven negative words and accepting all positive words. Measure 2 performed only slightly worse (rejecting six negative words and one positive word). Measure 1 performed poorly. It accepted all negative and all positive test words.

The performance of the network is not perfect. However, it seems reasonable to attempt to apply an SRN with this setup to the complete training data set.

3.3 Orthographic data with random initialization

We have trained SRNs to model the orthographic structure of the set of 5577 Dutch monosyllabic words described in section 2.2 of chapter 1. The networks have been trained with the setup which has been used in the previous section: learning rate η is 0.01, momentum α is 0.5, network performance is tested after every 50 training rounds and score correction for length. The length correction factors were equal to the average value of the words with length 4 divided by the average value of the words with length 5. These were the two most common lengths of the words in the training data. We have continued training until the change of the total sum squared error of the network resulting from a 50 rounds training period was smaller than 1%. We have explained that using an SRN for this experiment requires a separation character between the words but in order to make the comparison between these experiments and the bigram experiments of chapter 2 more fair we have provided all words with both a word start token and a word end token.

The model we are going to derive for the 5577 training words will be larger than the one for the 184 words starting with t. The size of the hidden layer in an SRN is

hidden cells	training rounds needed	total sum squared error	measure	accepted positive strings	rejected negative strings
4	250	26134	1	600 (100%)	0 (0%)
			2	599 (99.8%)	33 (5.5%)
			3	600 (100%)	50 (8.3%)
10	200	25711	1	600 (100%)	6 (1.0%)
			2	600 (100%)	3 (0.5%)
			3	600 (100%)	31 (5.2%)
21	250	24772	1	600 (100%)	0 (0%)
			2	600 (100%)	5 (0.8%)
			3	600 (100%)	6 (1.0%)

Figure 3.10: The performance of three different network configurations for the complete orthographic data set of monosyllabic Dutch words (5577 training words, 600 positive words and 600 negative words). The networks perform well with regards to accepting the words of the positive test set. However they reject far too few words of the negative test set.

closely related to the size of the model it can acquire. Therefore we expect that the number of hidden cells that was sufficient for the 184 words may be insufficient for the larger data set. But what would be the best size of the hidden layer? One can argue that the *t*-word SRN already contained a model for the words except for the first character (which always was a t), so adding one or two cells to the hidden layer should be enough to capture a complete model for monosyllabic words. At the other hand, someone might say that the training data increased with a factor 5577/184=30.3 so the number of weights should increase with a large factor. We tested both approaches and an intermediate one and performed additional experiments with SRNs with hidden layer size of 4, 10 and 21.[5]

We have noticed that the random initialization of the networks has some influence on their performance. In order to get a network performance that is reliable and stable, we performed five parallel experiments with each network configuration. Of these five experiments we chose the one with the lowest total sum squared error for the training data and tested its performance on the data set. Our motivation for choosing the performance of the network with the smallest error rather than the average performance of the networks is that we are interested in the best achievable performance of the network.

The network with 4 hidden cells in combination with measure 3 performed best,

[5]The number of four was derived by adding one to three (the number of hidden cells in the previous experiment). With 21 hidden cells we obtain a network that has approximately 30.3 times as many links as the t-words network. The value 10 lies somewhere between the other two numbers.

set 1:	V + , C V +	152 words
set 2:	V + , C V + , V + C , C V + C	1397 words
set 3:	the complete training set	5577 words

Figure 3.11: Increasingly complex training data sets. V is a vowel, C is a consonant and a character type followed by a plus means a sequence of 1 or more occurrences of that character type. The network will be trained with the first set first. When the network error stabilizes training will be resumed with the second data set. The training process will continue with the third set when the training process for the second set has stabilized.

rejecting 50 of 600 negative words (8.3%) while accepting all 600 positive words (see figure 3.10). The network tends to accept all words and this is not what we were aiming at. The performance even becomes worse when the number of hidden cells is increased. The total sum squared error decreases for a larger number of hidden cells which indicates that the network performs better on the training set. So a larger number of hidden cells makes it easier for the network to memorize features of the training set but it also degrades the network's generalization capabilities. This fact has already been recognized in other research using feed-forward networks.

If the SRNs are not able to learn the structure of our orthographic data with random initialization, we will have to rely on a linguistic initialization for a better performance.

3.4 Orthographic data with linguistic initialization

In the previous chapter we described how a Hidden Markov Model can start learning from some basic initial linguistic knowledge. Adding initialization knowledge to a neural network is more difficult. The most obvious way to influence the network training phase is by forcing network weights to start with values that encode certain knowledge. However, for a network of reasonable size it is hard to discover the exact influence of a specific weight on the performance of the network. Therefore it is difficult to find a reasonable set of initialization weights for a network.

Instead of trying to come up with a set of initialization weights, one can make the network discover such a set of weights. The idea would be to train the network on some basic problem and use the weights that are the result of that training phase as starting weights for a network that attempts to learn a complex problem. This approach to network learning was first suggested in (Elman 1991). In this work Jeffrey Elman described how a network that was not able to learn a complex problem was trained on increasingly more complex parts of the data. This approach was successful: after training the network was able to reach a satisfactory performance on the task.

We will describe an approach to our problem that is similar to the approach chosen by Elman. We divide our orthographically represented monosyllabic data in three sets

of increasing complexity. The first set contains words without consonants and words containing one initial consonant and one or more vowels. The second set contains all words of the first set plus the words that consist of vowels and a one consonant suffix and consonant prefix that is either empty or contains one consonant. The third set contains all words (see figure 3.11).

This approach seems implausible from a cognitive point of view since children do not receive language input with simple words first and complex words later. However if we look at the language production of children we see that the complexity of the structure of their utterances increases when they grow up. Young children start with producing V and CV syllables (V: vowel and C: consonant) before they produce CVC syllables and progress to more complex syllables. While their language input is not ordered according to syllable complexity, their language production model seems to develop from simple syllable output to output of more complex syllables. With the incremental approach we attempt to make our networks go through the same development process.

Our definition of vowels contained *a*, *e*, *i*, *o*, *u*, *y* and the diphthong *ij*. All other characters including the *j* without a preceding *i* were regarded as consonants. We have considered the *ch* sequence as one consonant because the pronunciation of this consonant sequence in Dutch is equal to that of the *g*.

We have trained the network with orthographic data using an SRN with 4 hidden cells because that one has performed best in the previous experiments. We have used the same experiment set up as described in the previous section: learning rate η is 0.01, momentum α is 0.5, network performance is tested after every 50 training rounds and score correction for length. Training data was of increasing complexity but as test data we have used the complete positive and negative test data sets for all experiments. We have used the performance of the network on the different training data sets for determining the threshold value and the score correction for length values for measure 2 and 3.

We have performed 5 experiments for all data sets. When the performance of an SRN for data set 1 stabilized, we continued training with data set 2. When the performance of that SRN stabilized we continued training with data set 3. The network with the lowest error rate after training with data set 3 was chosen for the final performance tests. Its two predecessors were used for the two intermediate performance tests.

This learning procedure was not been successful. The network performed even worse on the data than the SRNs with random initialization (see 3.12). Measure 3 performs best by accepting all positive words and rejecting 29 negative words (4.8%, while the SRN with random initialization was able to reject 8.3%). Aiding the network by structuring the training data does not improve the final performance of the network. The results that the SRNs have achieved for orthographic data are disappointing. For the time being we will refrain from testing the performance of the SRNs on phonetic data. Instead we will try to find out why SRNs are performing much worse on our task than we had expected.

training data set size	training rounds needed	total sum squared error	measure	accepted positive strings	rejected negative strings
152	300	452	1	548 (91.3%)	121 (20.2%)
			2	104 (17.3%)	489 (81.5%)
			3	392 (65.3%)	209 (34.8%)
1397	100	4529	1	560 (93.3%)	150 (25.0%)
			2	481 (80.2%)	224 (37.3%)
			3	398 (66.3%)	269 (44.8%)
5577	100	25334	1	599 (99.8%)	0 (0%)
			2	600 (100%)	27 (4.5%)
			3	600 (100%)	29 (4.8%)

Figure 3.12: The performance of the network when trained with orthographic data of increasing complexity. Network weights were initialized by values obtained from the experiment with less complexity. The network was tested with the complete positive test set and the complete negative test set. The errors of the network have been computed for the training sets which has different sizes for the three parts of the experiment.

4 Discovering the problem

In this section we will explain why the SRNs perform worse on our data than on the data of Cleeremans et al. First we will explain the major difference between our experiments and the ones of (Cleeremans 1993). Then we will show that complicating the data used in the experiments by Cleeremans et al. will decrease the performance of the SRNs. After this we will examine a possible solution for improving the performance of SRNs on our phonotactic data and present the results of this method.

4.1 The influence of the number of valid successors of a string

In the experiments described in the previous section the SRNs have been unable to acquire a good model for the phonotactic structure of Dutch monosyllabic words. While our target was rejecting all negative test data our SRNs have never been able to reject more than 8.3%. The performance of the Cleeremans measure was very disappointing: the measure never rejected more than 1.0% of the negative data. This performance is a sharp contrast with the performance on grammaticality checking reported in (Cleeremans 1993) in which *all* ungrammatical strings were rejected by the network. This fact surprised us so we took a closer look at the differences between

our experiments and the experiment described in (Cleeremans 1993) chapter 2.

The most obvious difference we were able to find was the maximal number of valid successors of grammatical substrings.[6] In the finite state model for the Reber grammar used by Cleeremans et al.(figure 3.6), at any point in a string the maximal number of valid successors is two. This is very important for the format of the network output. This output consists of a list of values between zero and one which are estimations for the probability that certain tokens are successors of a specific substring. For example, if a network trained with Reber grammar data is processing a string, it might present on its output the pattern [0.0 , 0.53 , 0.00 , 0.38 , 0.01 , 0.02 , 0.00] ((Cleeremans 1993), page 43). In this list the numbers estimate the probabilities that B, T, S, P, X, V, and E are the next token in some string. As we can see the two valid successors T (0.53) and P (0.38) receive an output value that is significantly larger than the output values of the other tokens. The network does not perfectly predict the correct successor but it outputs some average pattern that indicates possible successors. This network behavior was already recognized in (Elman 1990).

In the grammars for Dutch phonotactic structure the maximal number of valid successors is much larger than two. For example, according to our training corpus for the orthographic experiments the number of different tokens which are valid successors of the string *s* is 19. Some tokens occur frequently as successors of this character, for example *c* (23%) and *t* (24%) but others are very infrequent: *f*, *q* and *w* occur less than 1%. The frequency of the successors will be mirrored by the network output, as in the example from Cleeremans et al. But the network output contains random noise and therefore it will be difficult to distinguish between low frequency successors and invalid successors, like *b* and *x* in this example. For example, figure 3.7 shows network output in which *P* receives value 0.1 (ideally this should be 0.0) and *S* receives value 0.4 (ideally this should be 0.5). Cleeremans et al. have detected in the output of their network values of 0.3 where 0.5 was expected. One can understand that in an environment where signals have an absolute variation of 0.2 the difference between a signals 0.01 and 0.00 are in practice impossible to detect.

An alternative approach to choosing a threshold value is to take a larger threshold value in advance in order to reject more non-grammatical strings. This will not bring us closer to a solution because a larger threshold value will immediately reject the element with the smallest value of the training set. It might even cause the rejection of other grammatical elements from the test set and the training set. This is not acceptable to use: we want our models to accept all training data.

4.2 Can we scale up the Cleeremans et al. experiment?

We decided to test the influence of the number of valid successors by repeating the experiment described in (Cleeremans 1993) chapter 2. Apart from training an SRN to decide on the grammatically of strings according to the Reber grammar shown

[6] The standard term for the *average* number of valid successors is PERPLEXITY: $PP(W) = 2^{H(W)}$ where W is some data set and H(W) is the entropy of this data set (Rosenfeld 1996).

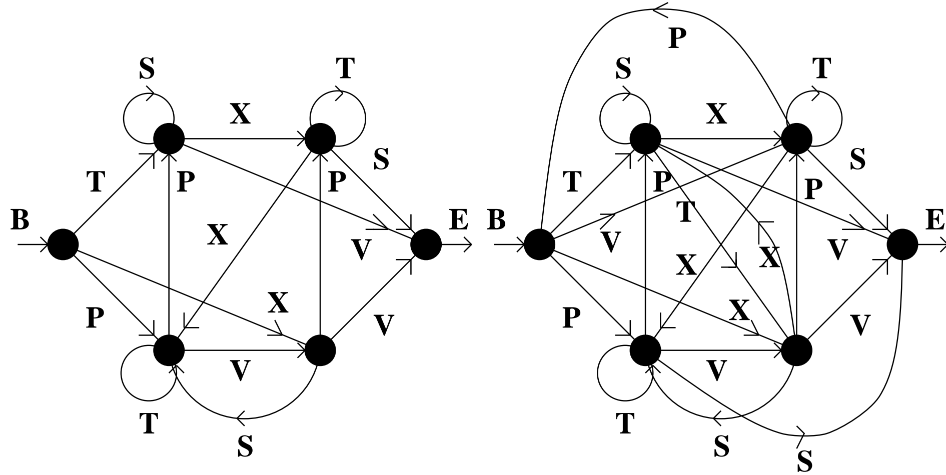


Figure 3.13: Finite state networks representing a Reber grammar with three valid successors for every valid substring (left) and one with four valid successors (right). The valid successors of, for example, substring *BT* are *S*, *X* and *V* for the three-successor grammar and *S*, *X*, *V* and *T* for the four-successor grammar.

in figure 3.6, we trained SRNs with strings from two alternative grammars. The first alternative grammar was an extension of the Reber grammar such that valid substrings have three possible successors (see figure 3.13). The second alternative grammar was an extension of the first and here valid substrings have four possible successors (see figure 3.13). For all grammars we have generated 3000 training words, 300 positive test words and 300 random words. Of the 300 random words for the Reber grammar 1 was grammatical and for the other two grammars 5 and 8 strings were valid.

We have trained the SRNs in steps of 50 training rounds using learning rate η=0.01, momentum α=0.5 and a network configuration that was similar to the one of Cleeremans et al. (7 input cells, 3 hidden cells, 3 context cells and 7 output cells). After each step we checked if the total sum squared error of the network had changed more than 1%. If this was the fact we continued training otherwise training was stopped. Only one experiment was performed for each grammar. As in the previous experiments we will reject words when their score is below the lowest score of the training data.

As we can see in figure 3.14 the performance of the SRN decreases when the complexity of the grammar increases. The SRN accepted all valid words for all grammars but the rejection rate of the random data decreased when the number of valid successors increased: for the grammar with two valid successors (one valid string in the random data) 100% of the invalid random strings were rejected, for three valid successors (5 valid strings) this dropped to 92.2% and for four valid successors (8

maximum number of successors	training rounds needed	total sum squared error	measure	accepted positive strings	rejected random strings
2	150	9079	1	300 (100%)	299 (99.7%)
			2	300 (100%)	296 (98.7%)
			3	300 (100%)	299 (99.7%)
3	250	12665	1	300 (100%)	272 (90.7%)
			2	300 (100%)	254 (84.7%)
			3	300 (100%)	241 (80.3%)
4	100	13944	1	300 (100%)	240 (80.0%)
			2	298 (99.3%)	172 (57.3%)
			3	299 (99.7%)	119 (39.7%)

Figure 3.14: The performance of the SRN for Reber grammars of increasing complexity. Some random strings are valid: 1 for the first grammar, 5 for the second grammar and 8 for the most complex grammar. The actual rejection percentages for measure 1 for the invalid data are 100.0% for the first grammar, 92.2% for the second grammar and 82.2% for the third grammar. The performance of the network deteriorates when the number of valid successors increases.

valid strings) the rejection rate was 82.1%. We can conclude that the number of valid successors influences the difficulty of the problem. (Cleeremans et al. 1989), (Servan-Schreiber et al. 1991) and (Cleeremans 1993), showed that SRNs are capable of acquiring finite state grammars in which a grammatical substring has two valid successors. We have showed that the performance of the SRNs will deteriorate rapidly when the maximal number of successors increases.

4.3 A possible solution: IT-SRNs

The output patterns of the SRNs after training shown in (Cleeremans 1993) page 43 show that the sum of the output signals is approximately equal to 1. One of the patterns shown is the output pattern for input *B* in an SRN modeling the Reber grammar: [0.0, 0.53, 0.00, 0.38, 0.01, 0.02, 0.00] (sum is 0.94). We have observed the same behavior from our SRNs. This behavior can be explained by the fact that the patterns we use as required output patterns during training consist of a single 1 and zeroes for the other values. For every input pattern there are usually different required output patterns in the training data and the output pattern of a network after training will be an average of all possible output patterns for that particular input. The values of each pattern sum up to one and therefore the values of an exact average will also sum up to one.

In an earlier section we have argued that it is difficult to distinguish a small output value (caused by an infrequent successor) from network output noise. This is the

cause of poor performance of the SRNs in the previous experiments. We would like to increase the difference between the output values for infrequent successors and the network noise. We cannot do much about the network noise but we can try to increase the output values for the successors.

We will change the required SRN output patterns during training in such a way that they will satisfy the following requirement:

> INCREASED THRESHOLD REQUIREMENT
> If SRN training data that has been encoded by using the localist mapping allows token Y as the successor of token X then the required value of the output cell which is 1 for token Y should be equal to or larger than some threshold value during training whenever X is presented as input.

We will call SRNs which satisfy this requirement INCREASED THRESHOLD SRNS or in short IT-SRNs. Let us look at how this requirement influences the training process. We will examine a network trained to model the standard Reber grammar with two valid successors (figure 3.6) and look at the valid successors for the start character *B*.

B can be followed by *T* or *P*. In the output pattern of the network *T* is represented by the second value and *P* is represented by the sixth value. Therefore a *B* at the input of a standard SRN should require [0.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0] on the output when its successor is *T* and [0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 1.0 , 0.0] when its successor is *P*. However, in a IT-SRN the output values corresponding with all valid successors of a pattern should at least be equal to some threshold value. If we choose 0.5 as threshold value and present *B* as input the IT-SRN should require [0.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.5 , 0.0] as output when the successor is *T* and [0.0 , 0.5 , 0.0 , 0.0 , 0.0 , 1.0 , 0.0] when the successor is *P*.

By making a network satisfy this increased threshold requirement it should become easier to recognize the difference between low frequency output values and noise. All valid successors of a character will be represented in every output pattern of this character during training. The average output values of the network will increase and so will the values assigned by the network to the training and the test data. We assume that IT-SRNs will be able to reject more negative test data than standard SRNs. In the next section we will test this assumption.

4.4 Experiments with IT-SRNs

We have trained the IT-SRNs to model the orthographic data first with random initialization and after that with linguistic initialization. Data and experiment set up were equal to the experiments described in section 3.3 and 3.4: learning rate η is 0.01, momentum α is 0.5, network performance is tested after every 50 training rounds and score correction for length. We have chosen threshold value 0.5 for valid successors and used IT-SRNs with four hidden cells because SRNs with 4 hidden cells have achieved the best result in the experiments described in section 3.3. The training data consisted of 5577 monosyllabic Dutch words and tests were performed with 600

initialization	training rounds needed	total sum squared error	measure	accepted test strings	rejected random strings
random	200	163496	1	600 (100%)	15 (2.5%)
			2	600 (100%)	26 (4.3%)
			3	600 (100%)	20 (3.3%)
linguistic: 157 training strings	800	1631	1	11 (1.8%)	593 (98.8%)
			2	11 (1.8%)	594 (99.0%)
			3	242 (40.3%)	410 (68.3%)
linguistic: 1397 training strings	300	26587	1	468 (78.0%)	247 (41.2%)
			2	434 (72.3%)	409 (68.2%)
			3	600 (100%)	6 (1.0%)
linguistic: 5577 training strings	250	164042	1	600 (100%)	0 (0%)
			2	600 (100%)	2 (0.3%)
			3	600 (100%)	21 (3.5%)

Figure 3.15: The performance of the 4 hidden cell IT-SRN with random initialization and IT-SRNs with linguistic initialization for the orthographic data. The total sum squared errors are smaller for the first two linguistic experiments because they were computed for the small training sets. The IT-SRNs performed best with random initialization for measure 2 but it only rejected 4.3% of the negative test data. Neither randomly initialized nor linguistically initialized IT-SRNs were able to reject more negative words than randomly initialized SRNs.

test words which were not present in the training data and 600 ungrammatical strings. In the experiments with linguistic initialization the training data was divided in three groups of increasing complexity just like in section 3.4. In both experiment groups 5 parallel experiments were performed and the IT-SRN which achieved the lowest error for the training data was used for the tests.

The results of the two experiments can be found in figure 3.15. The error of the network has become much larger than in the SRN experiments because the IT-SRN output patterns during training are different from the output patterns during testing. IT-SRNs do not perform better on this data than SRNs. The randomly initialized IT-SRN has achieved the best performance with measure 2 accepting all positive test data but rejecting only 26 negative strings (4.3% while the randomly initialized SRNs have achieved 8.3%). The linguistically initialized IT-SRN has achieved the best performance with measure 3 accepting all positive test data but rejecting only 21 negative strings (3.5% while the linguistically initialized SRNs have achieved 4.8%).

The performance difference between IT-SRNs and SRN can only be made visible if we change the threshold used for deciding if a string is grammatical or not. We have defined this threshold value as being equal to the lowest score assigned to a word in

network	measure	100%		99%		95%		90%	
		positive	negative	pos.	neg.	pos.	neg.	pos.	neg.
SRN	1	0	0	5	14	28	105	61	208
	2	1	33	12	207	31	325	53	395
	3	0	50	7	270	25	399	65	469
IT-SRN	1	0	15	12	306	35	371	52	438
	2	0	26	11	316	32	405	58	449
	3	0	20	3	56	24	110	53	156

Figure 3.16: Number of words that were rejected in the orthographic positive and negative test data in the randomly initialized SRN and IT-SRN when the acceptance threshold is increased and only part on the training data will be accepted. The percentage indicates the amount of training data that will be accepted. Now the network is able to reject 78.2% of the negative words (SRN, measure 3, 90% column) but the price is high: 10.8% of the positive test data and 10.0% of the training data will be rejected as well.

the training data. In that way we have made sure that all words in the training data will be accepted. If we allow some words of the training data to be rejected then we can raise the value of the threshold. The result of this has been made visible in figure 3.16.

When we allow 1% of the training data being rejected, the IT-SRN will reject at best 316 words of the negative test data (52.7%, measure 2). The SRN will reject at best 270 words of the training data (45.0%, measure 3) when we apply the same constraint. Increasing the acceptance threshold seems to be the only way to improve the performance of the SRNs and the IT-SRNs on the negative data. However, it is not a good solution. The best performance that we have achieved by increasing the acceptance threshold was rejection of 469 words of the negative test data (78.2%, SRN with measure 3) but then we also had to reject 10% of the training data and 10.8% of the positive test data.

Our conclusion is that neither IT-SRNs nor an increasement of the acceptance threshold will produce acceptable behavior of the networks on our learning task.

5 Concluding remarks

In this chapter we have described our attempts to solve the central problem in this thesis, automatic acquisition of the phonotactic structure of Dutch monosyllabic words, by using connectionist techniques, in particular the Simple Recurrent Network (SRN) developed in (Elman 1990). We compared two of our word evaluation measures with a word evaluation measure used in (Cleeremans 1993), (Cleeremans et al. 1989) and

(Servan-Schreiber et al. 1991). In the experiments of Cleeremans et al. the latter measure reached a 100% performance in a grammaticality checking task using a small artificial grammar (the Reber grammar, see figure 3.6). Our experiments resulted in SRNs which performed well with respect to accepting the positive orthographic test data. However, they performed poorly in rejecting negative orthographic data: they never rejected more than 8.3% of our 600 negative strings. In these experiments the measure used by Cleeremans et al. came out as the worst of the three measures used. It never rejected more than 1.0% of the negative data.

We have shown that the reason for the difference in performance lies in the complexity of the grammars that we are teaching the networks. Cleeremans et al. have used a grammar in which for every grammatical substring the maximal number of valid successors never is larger than two. In the grammars we use this number is about 10 times as large. Therefore the relevant output values of the network become smaller and it is more difficult to distinguish them from random network noise. We have shown that even a small increase of the complexity of the grammar that models the training data will lead to a deterioration of the behavior of SRN models trained on this data. Increasing the maximal number of valid successors in the Reber grammar to three resulted in the SRN being unable to reach perfect performance in a grammaticality checking task.

We have attempted to solve this problem by changing the required output patterns during training. We have forced the network to use information about all valid successors of a character instead of just learning the relation between two characters at each training step. However the resulting modified SRN, Increased Threshold SRN (IT-SRN), did not perform better than the standard SRN. At best it rejected only 4.3% of the negative test data.

It seems that the only way to make the SRNs and the IT-SRNs reject more invalid data is increasing the acceptance threshold value and thus allowing that a part of the training data will be rejected. But even when we allow 10% of the training data to be rejected we cannot make the networks reject more than 78.2% of the invalid data This is a low percentage compared with the performance of the HMMs. Furthermore this solution is not acceptable to us: we want our models to accept all training data.

We conclude from the experiments described in this chapter that Simple Recurrent Networks do not seem to be able to reach a good performance on our learning task.[7]

[7] Some alternative approaches with SRNs to this learning task will be discussed in section 2 of chapter 5.

Chapter 4

Rule-based Learning

Theories in theoretical linguistics usually consist of sets of discrete non-statistical rules. However, the models that we have developed in the previous chapters have a different format. It would be interesting to obtain phonotactic models that consist of rules. We could obtain such models by converting our HMMs and our SRN models to sets of rules. However, there are also rule-based learning methods available which generate rule models. Given linguists's preference for rule-based description, we might expect these methods to perform better than the other learning methods we have applied to our problem, learning the structure of monosyllabic words. Therefore we would like to apply rule-based learning methods to this problem and compare their performance with the performances we have obtained in the previous chapters.

In this chapter we will describe a rule-based approach to learning the phonotactic structure of monosyllabic words. In the first section we will sketch the fundamentals of rule-based learning and point out some of its problems. The second section will outline the learning method we have chosen: Inductive Logic Programming (ILP). The learning experiments that we have performed with ILP will be presented in the third section. In the fourth section we will describe our work with more elaborate rule-based models which might perform better than the models used in section three. The final section contains some concluding remarks.

1 Introduction to Rule-based Learning

In this section we will present some basic learning theory and evaluate a few rule-based learning algorithms. We will start with the influence of positive and negative examples on the learning process. After this we will explain what kind of output we

expect from the learning method. We will conclude the section by presenting some rule-based learning methods and discussing their problems.

1.1 Positive versus negative examples

We want to use a symbolic or rule-based learning technique for acquiring a rule model for the phonotactic structure of monosyllabic words. This means that we are looking for a learning method that can produce a rule-based model that can accept or reject strings for a certain language. The model should accept a string when it is a possible word in the language and it should reject it when it is a sequence of characters that cannot occur as a word in the language.

Theory about learning methods that can produce language models like described above is already available. One of the main papers in learning theory was written by E. Mark Gold in the sixties (Gold 1967). In the paper Gold uses a division of languages in different mathematical classes to characterize the type of information that a hypothetical learner requires to learn them. He proves that no infinite language can be learned by using positive examples only. This means that learners cannot be guaranteed to acquire a perfect model of an infinite language without being presented with examples of strings or sentences that do not appear in the language.

The research result of Gold is important for the work we will present in this chapter. In our earlier experiments we have presented the learning algorithms only with positive examples. We want to be able to compare the results of this chapter with the results of the previous chapters in a fair way. This prevents us from using negative examples in the remaining experiments. We want our language models to generate as few errors as possible and therefore it is important to find out if the language we are trying to learn is finite.

In chapter one we have restricted our dataset to monosyllabic words. The longest word in the Dutch orthographic data set contains nine characters. We can imagine that it is possible to construct a word of ten or eleven characters that could qualify to be a monosyllabic Dutch word. However we cannot imagine that it is possible to construct monosyllabic Dutch word of twenty characters or longer.[1] The class of monosyllabic words will be finite for all human languages. Thus it should be possible to acquire a good monosyllabic structure model by using positive examples only. This would not have worked for multisyllabic words since in languages like Dutch and German there does not seem to be a maximum length for words, particularly not for compound words.

[1] In some unusual contexts strings with an arbitrary number of repetitions of the same character, like *aaaah* and *brrrr*, may qualify as a monosyllabic word. The models which we will use will be able to learn these by adopting the following strategy: a sequence of two or more repeated characters indicates that the repetition in that context is allowed for any number of characters.

1.2 The expected output of the learning method

In his landmark paper Gold introduces the concept *learning in the limit*. Learners are presented with positive and/or negative examples of a certain domain. They have a partial model of this domain and adjusts this model for every example that they receive. At some moment they will have constructed a perfect model of the domain and future examples will not cause a change of the model. If the learners manage to construct such a perfect model at some point of time then they are said to have learned the training data in the limit.

The learner model put forward by Gold makes no assumptions about the format of the domain model that has to be learned. In case of a finite domain the model may well consist of rules which state that some X is in the domain if and only if X was presented as a positive example. This is the most simple domain model structure. A model structure like that will not perform well in the experiments we are planning to do because it is unable to generalize. In the previous chapters we have presented the learning methods with only a part of the positive data. After this we have tested the domain models with the remaining unseen positive data and some negative data. A simple domain model like described above would reject all the unseen positive test data. This is unacceptable.

In order to avoid this problem the domain model must be able to generalize. Based on positive data only it must be able to decide that the unseen positive test data is correct. In order for the model to be able to do that, we will have to put generalization knowledge in the learning model. The question is what kind of generalization knowledge the model needs to contain. If the knowledge is too restrictive the domain model will reject too much positive test data and if it is too extensive the domain model will accept negative data as well. We will return to this question in section 2.

In the previous two chapters we have worked with Hidden Markov Models and neural network models that represent the structure of monosyllabic words. A disadvantage of these two groups of models is that they hide knowledge in sets of numbers. It is hard to explain the behavior of these models in a way that makes much sense to humans. We believe that the behavior of the models generated by the symbolic learning method should be comprehensible to humans - at least to linguists - in a sensible way. An advantage of this is that explainable models allow finding the cause of possible errors in the model more easily so we will be able to correct them. The behavior of a model consisting of rules can be explained in a sensible way and therefore we want the models generated by the symbolic learning method to consist of rules.

1.3 Available symbolic learning methods

There are many symbolic or rule-based learning methods available (Winston 1992). In this chapter we will concentrate on two groups of symbolic learning methods which are currently the most popular ones: lazy learning and decision trees. Learning methods that fall in these classes are generally used for classification tasks (Van den Bosch et al. 1996b).

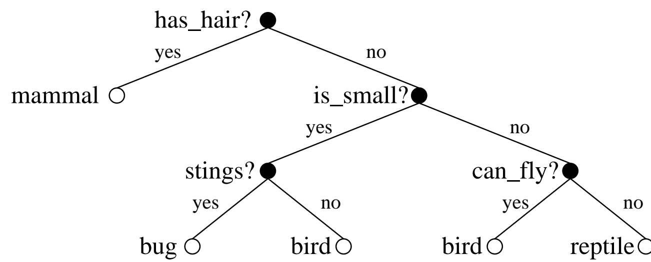


Figure 4.1: An example decision tree for animal classification. A number of questions have to be answered in order to come to a classification. This tree would classify a mosquito as a bug because it does not have hair, is small and stings.

Lazy learning methods or memory-based methods learn the structure of a domain by storing learning examples with their classification (Van den Bosch et al. 1996b). The domain model that results from a lazy learning process is able to generalize by using a predefined distance function. When the domain model is required to give the classification for an unseen domain element then it will use the distance function for finding the stored example that is closest to this unseen example.

The following is a simple application of a lazy learning method to animal classification. Suppose we know the classification of the two animals lion and mosquito. A lion has been classified as a mammal and a mosquito has been classified as a bug. This information is presented as information to the learning algorithm together with features representing the two animals, for example [big, has_hair , eats_animals] for lion and [small, can_fly , stings] for mosquito. The algorithm will compare the features for new animals and classify them either as mammal or bug depending on how many features the new animals have in common with the stored animals. For example, a bat with the features [small , can_fly , has_hair] would have been classified as a bug because it has two features in common with mosquito and only one with lion.[2]

Decision tree methods like, for example, C4.5 build a tree describing the structure of the domain (Quinlan 1993). The nodes of the tree consist of questions regarding the values of the elements of the domain. When one wants to know the classification of a certain domain example one will start at the root of the tree and answer the first question with regard to the example. The answer to the question will lead to another node with another question. In this way one will traverse the tree until a leaf node is reached. This leaf node contains the classification of the example.

A decision tree method applied to our animal classification example could result in a tree which contained only one question. There are several alternative questions possible and the method has too few learning examples to make a good choice. A

[2]In our example the classification algorithm assign the same weight to all features but this does not necessarily have to be so.

possible question discriminating the two learning examples could have been *Does it have hair?*. The answer *yes* would result in a mammal classification because a lion has hair and the answer *no* would result in a bug classification. This question would classify bat as a mammal because it has hair.

Both decision tree methods and lazy learning methods have some problems when applied to our learning problem with the constraints we have put on the output from the learning method. First of all these learning methods require learning examples of all possible qualification classes. Our research problem contains two qualification classes: valid strings and invalid strings. However we want to use the same learning input as used in our previous experiments and this means that we want to train the symbolic learning method by using positive examples only. Neither decision tree methods nor lazy learning were designed for training with positive examples only.[3]

A second disadvantage of these two symbolic learning methods is that they do not generate rules. Instead they hide knowledge in a big database of examples combined with a distance function (lazy learning) or a tree with decision nodes (decision trees). There are ways of converting the behavior of decision tree models to comprehensible rules but for the behavior of lazy learning models this will be very difficult. Still we would like to keep our constraint of generating a model that consists of rules and use that to mark these two model structures as a disadvantage.

These two disadvantages make the use of either lazy learning or decision tree learning unacceptable to us. Other rule-based methods suffer from similar problems. Version Spaces require negative examples in order to derive reasonable models ((Winston 1992), chapter 20). C4.5 is a decision tree method and thus it requires negative examples as well (Quinlan 1993). Explanation-Based Learning is able to learn from positive data only by relying on background knowledge (Mitchell et al. 1986). The learning method that we have chosen uses the same approach. We will describe this learning strategy in the next section.

2 Inductive Logic Programming

In this section we will describe the symbolic learning method Inductive Logic Programming (ILP). The description has been divided in four parts: a general outline, a part about the background knowledge concept, a part on how ILP can be used in language experiments and a part which relates our models to grammar theory.

2.1 Introduction to Inductive Logic Programming

Inductive Logic Programming (ILP) is a logic programming approach to machine learning (Muggleton 1992). The term induction in the name is a reasoning technique

[3] See the section 1.1 of this chapter and section 1.3 of chapter 1 for a motivation about using only positive examples

which can be seen as the reverse of deduction. To explain inductive reasoning we first look at a famous example of deduction:

P_1	All men are mortal.
P_2	Socrates is a man.
DC	Socrates is mortal.

This example contains two premises P_1 and P_2. By using these two premises we can derive by deduction that DC must be true. Thus deduction is used to draw conclusions from a theory.

In inductive reasoning we start with facts and attempt to derive a theory from facts. An example of this is the following derivation:

P_1	All men are mortal.
P_2	Socrates is mortal.
IC	Socrates is a man.

Again we have two premises P_1 and P_2. By using these two we can derive by induction that IC could be true. An inductive derivation like this one can only be made if the inductive conclusion (IC) and one premise (here P_1) can be used for deductively deriving the other premise (here P_2). The inductive derivation is not logically sound: that is, the result of the derivation might be wrong. Yet inferences like these are made by humans in everyday life all the time and they are exactly what we are looking for. Inferences like these give us a method for deriving a theory from a set of facts.

ILP theory makes a distinction between three types of knowledge namely background knowledge, observations and hypotheses (Muggleton 1992). Background knowledge is initial knowledge that learners have before they start learning. Observations are the input patterns for the learning method with their classifications. The hypotheses are the rules that ILP derives from the training data. The relation between these three knowledge types can be described with two rules:

DR: $B \wedge H \vdash O$	Example:	B	All men are mortal.
		H	Socrates is a man.
		O	Socrates is mortal.
IR: $B \wedge O \leadsto H$	Example:	B	All men are mortal.
		O	Socrates is mortal.
		H	Socrates is a man.

These rules contain three symbols: $\wedge$ stands for *and*, $\vdash$ stands for *leads deductively to* and $\leadsto$ stands for *leads inductively to*. DR represents the deductive rule which states that the observations (O) are derivable from the background knowledge (B) and the hypotheses (H) (Muggleton 1992). The inductive rule IR represents the inductive step that we want to make: derive hypotheses from the background knowledge and the observations.

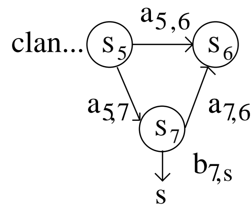


Figure 4.2: A schematic representation of the addition of a character after a valid word in an HMM. The character *s* is appended to word *clan* and as a result of this the path from state s_5 via state s_7 to s_6 will be used during processing. The score assigned by the HMM to the word *clans* will be equal to the score assigned to *clan* multiplied with the factor $(a_{5,7} * b_{7,s} * a_{7,6})/a_{5,6}$.

The inductive rule IR can be used for deriving many different hypotheses sets and the most difficult problem in ILP is to constrain the number of hypotheses sets. Initially only one restriction will be placed on the hypotheses sets: they must satisfy the deductive rule. In our inductive Socrates example the fact *All men are mortal* would be background knowledge and *Socrates is mortal* would be the observation. Then *Socrates is a man* would be a valid hypothesis because we can derive the observation from this hypothesis and the background knowledge. The fact *Socrates is a philosopher* cannot be used in combination with the background knowledge to derive the observation and therefore this fact is not a valid hypothesis.

In our Socrates example we could also have generated the hypothesis *All things named Socrates are man*. This is an example of an alternative valid hypothesis. The facts in the background knowledge and the observations are usually larger in number than in this toy example. The number of valid hypotheses will grow rapidly with any fact we add to the background knowledge and the observations. (Muggleton 1992) suggests four ways for reducing the number of hypotheses: restricting the observations to ground literals,[4] limiting the number of the hypotheses to one, restricting the hypotheses to the most general hypotheses relative to the background knowledge and choosing the hypotheses which can compress the observations as much as possible. In section 2.3 we will discuss these reduction ways and outline which one we will use.

If we want to apply ILP to the problem of acquiring the structure of monosyllabic words then we will have to make two decisions. First we have to decide what we should use as background knowledge. Second we should decide how we are going to use the induction rule for generating monosyllabic word models in practice. We will deal with these two topics in the next two sections.

[4] A ground literal is a logical term without variables. For example the term *man(Socrates)* is a ground literal but *man(x)→mortal(x)* is not because it contains a variable (x).

2.2 The background knowledge and the hypotheses

Before we are going to deal with the format and contents of the background knowledge and the hypotheses that are appropriate for our problem we first have to note that there is an important constraint on these bodies of knowledge. We will compare the results of ILP with the results of the previous chapters in which Hidden Markov Models (HMMs) and Simple Recurrent Networks (SRNs) were used. If we want this comparison to be fair then we should take care that no learning method receives more learning input than another. This imposes a constraint on ILP's background knowledge and the format of the hypotheses. There is a danger that we use background knowledge that give ILP an advantage over the other two algorithms. This should be avoided.

In our application of ILP to the acquisition of monosyllabic word models the background knowledge and the hypotheses will contain explicit and implicit knowledge about the structure of syllables. In order to make sure that this knowledge does not give ILP an advantage over the other two learning algorithms we will show how this knowledge is implicitly used in HMMs. This explanation will require some basic HMM knowledge of the reader. You can look back at chapter 2.1 if necessary.

In our ILP models we want to use rules which add a character to a valid word and thus produce another valid word. An example of such a rule is: *When an* s *is added behind a valid word that ends in* n *the resulting word will be valid as well.* Rules of this format are implicitly used in HMMs:

Suppose an HMM is able to process the words *clan* and *clans* and accepts both words. If we ignore all but the most probable path, processing *clan* involves using six states because it contains four visible characters, a begin-of-word character and an end-of-word character. We can label the states in this path with the numbers one to six. The word *clans* can be processed with the same path with one extra state inserted between state five and state six. This extra state will take care of producing the suffix *s* and we will call it state seven (see figure 4.2).

The HMM will assign a different probability to the two words. We can encode the difference with the formula:

$$P(clans) = P(clan) * (a_{5,7} * b_{7,s} * a_{7,6})/a_{5,6}$$

in which P(W) is the score assigned by the HMM to the word W, $a_{i,j}$ is the probability of moving from state i to state j, and $b_{i,c}$ is the probability that state i produces character c. The link from state five to six ($a_{5,6}$) has been replaced by the links from state five to seven ($a_{5,7}$) and state seven to six ($a_{7,6}$). Furthermore the probability of producing character s in state seven has to be taken into account ($b_{7,s}$). Figure 4.2 contains a schematic representation of the change in the HMM processing phase. The probabilities of the words may be different but the HMM accepts both words. This could have been explained if we knew that the factor $(a_{5,7} * b_{7,s} * a_{7,6})/a_{5,6}$ is equal to one. We don't know if that is true but we will assume that it is true.[5]

[5] Here we have assumed that we have a perfect HMM; one that assigns the score one to all valid strings

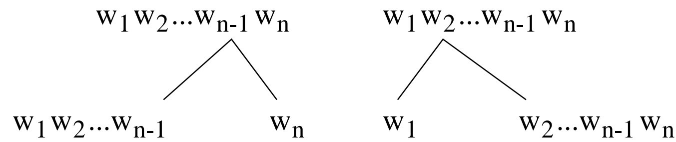


Figure 4.3: A tree representation of the background knowledge suffix rule (left tree) and the background knowledge prefix rule (right tree). Word W=$w_1 w_2 ... w_{n-1} w_n$ is a valid word if its final character w_n is a valid suffix for words ending in its penultimate character w_{n-1} and if $w_1 w_2 ... w_{n-1}$ is a valid word. Word W is a valid word if its initial character w_1 is a valid prefix for words starting with its second character w_2 and if $w_2 ... w_{n-1} w_n$is a valid word.

If we assume that the factor is equal to one then we can add s_7 to any path for words that end in *n* and create new words that end in *ns*. So we can translate the assumption to a rule that states that from every valid word which ends in *n* one can make another valid word by adding a suffix *s* to the word. Our motivation for choosing a context string of length one is that we have used context strings of the same length in the HMM chapter. The final version of the assumption is exactly the rule which we started with: *When an* s *is added behind a valid word that ends in* n *the resulting word will be valid as well*.

Now we have derived a rule which is being used implicitly in the bigrams HMMs we have worked with in chapter 2. This means that we can use the rule in the ILP model. The rule could be used as a hypothesis. However we want our hypotheses to be as simple as possible and therefore we will split the rule in a background knowledge part and a hypothesis part. The background knowledge part has the following format:

> BACKGROUND KNOWLEDGE SUFFIX RULE
> Suppose there exists a word W=$w_1 ... w_{n-1} w_n$ ($w_1 ... w_n$ are n characters) and a suffix hypothesis SH(w_{n-1},w_n).
> In that case the fact that W is a valid word will imply that $w_1 ... w_{n-1}$ is a valid word and vice versa.

Now the SUFFIX HYPOTHESIS for this specific case would be defined as SH(*n*,*s*) which states that an *s* is a valid suffix for words ending in *n*. This hypothesis format regards a character in a context containing one neighboring character. This implies that we are working with character bigrams just like in our HMM experiments.

The convention in the ILP literature is to represent knowledge in Prolog-like rules. These rules have the format A←B,C which means that A is true when both B and C are true. The capital characters can be replaced by predicates like abcd(X) which

---

and the score zero to all invalid strings. In practice there will be some deviation in the string score. The factor $(a_{5,7} * b_{7,s} * a_{7,6})/a_{5,6}$ will nearly always be smaller than one but this problem was fixed in our HMMs by score correction for length (section 3.4 of chapter 2) so that it does not affect our assumption.

stands for the proposition *X has property abcd*. We will include a similar notation in our definitions of background knowledge. The background knowledge suffix rule can be represented with two rules: valid($w_1...w_{n-1}$)←valid($w_1...w_{n-1}w_n$),SH(w_{n-1},w_n) and valid($w_1...w_{n-1}w_n$)←valid($w_1...w_{n-1}$),SH(w_{n-1},w_n).

Now that we have derived one background rule we can build other background rules in a similar way. We need a rule which adds a character to the front of a word, for example: a valid word that starts with *p* can be converted into another valid word if we put an *s* in front of it. Again we will split this rule into a general part for the background knowledge and a specific part which will be a hypothesis. The background part of this prefix rule has the following format:

> BACKGROUND KNOWLEDGE PREFIX RULE
> Suppose there exists a word W=$w_1w_2...w_n$ and a prefix hypothesis PH(w_1,w_2). In that case the fact that W is a valid word implies that $w_2...w_n$ is a valid word and vice versa.
> Rule notation: valid($w_2...w_n$)←valid($w_1w_2...w_n$),PH(w_1,w_2)
> valid($w_1w_2...w_n$)←valid($w_2...w_n$),PH(w_1,w_2)

And the PREFIX HYPOTHESIS for this specific case can be specified as PH(*s*,*p*) which states that an *s* is a valid prefix for words starting with *p*.

The two rules can be used for explaining why complex words are valid based on the fact that basic words are valid. We need a rule that specifies what specific basic words will be valid. This rule will be very simple: a word will be valid when it has been defined as a valid basic word. Again we will divide this rule in two parts: a background knowledge part and a hypothesis part. The definition of the background part is:

> BACKGROUND KNOWLEDGE BASIC WORD RULE
> The existence of a basic word hypothesis BWH(W) implies that word W is a valid word.
> Rule notation: valid(W)←BWH(W).

And an example of a BASIC WORD HYPOTHESIS is BWH(*lynx*) which states that *lynx* is a valid word.

This concludes the derivation of the background knowledge and the format of the hypotheses. By starting from an HMM that processed a word we have derived the format of three hypotheses and three background knowledge rules. In the background knowledge we have defined that words can be regarded as a nucleus to which prefix characters and suffix characters can be appended. This knowledge is implicitly available in HMMs so by using it in or ILP rules we have not supplied the ILP algorithm with knowledge that was not available to the learning algorithms used in the previous chapters.

2.3 Deriving hypotheses

Now that we have defined the background knowledge and the format of the hypotheses we should explain how an ILP algorithm can derive the hypotheses from the learning input. This means that we must make explicit how the inductive rule IR defined in section 2.1 will be realized in practice. As we have explained there are many sets of hypotheses that can be derived by this rule. The most important problem in using this inductive rule is cutting down the number of acceptable hypothesis sets without removing the interesting ones.

We will derive three inference rules based on the background knowledge format we have described in the previous section. This derivation will be based on an example. We will try to use ILP for generating a model which describes the three words *clan*, *clans* and *lans*. These will be our observations and the background knowledge will be as described in section 2.2. Our task is to derive suffix hypotheses, prefix hypotheses and basic word hypotheses for these observations.

In this particular example the words can be explained by each other in combination with appropriate suffix hypotheses and prefix hypotheses. If this had not been the case we would have been forced to define the validity of the three words with three basic word hypotheses without being able to add any other hypotheses. We will consider the recognition of basic word hypotheses as the initial step of the ILP algorithm. This initial step can be made explicit with the following inference rule:

> BASIC WORD INFERENCE RULE
> If word W is a valid word then we will derive the basic word hypothesis BWH(W). All observed words are valid words.
> Rule notation: $\dfrac{\text{valid(W)}}{\text{BWH(W)}}$

In our example this inference rule will produce the hypotheses BWH(*clan*), BWH(*clans*) and BWH(*lans*). These hypotheses can be used in combination with the background knowledge basic word rule to prove that the three words are valid. By using the basic word inference rule we have derived a model that explains the learning input data. This model is an example of the simple domain model we have discussed in section 1.2. The problem of that model was that it will reject all unseen data because it is unable to generalize. This means that we should restructure the model to enable to generalize. Two other inference rules will take care of that.

We can prove that *clans* is a valid word by using the basic word hypothesis BWH(*clans*) and the background knowledge basic word rule. The fact that this word is valid could also be explained by the fact that *lans* is valid in combination with the background prefix rule and a prefix hypothesis PH(*c*,*l*). The latter hypothesis does not exist in our present model and we want to be able to derive it. The hypothesis is not necessary for explaining the valid word but it will add generalization possibilities to the model. It is very important that the final model is able to generalize and therefore we will include in it all prefix hypotheses and all suffix hypotheses which can be used for explaining valid words.

Prefix hypotheses can be derived with the following inference rule:

PREFIX HYPOTHESIS INFERENCE RULE
If W=$w_1 w_2 ... w_n$ is a valid word and $w_2 ... w_n$ is a valid word as well then we will derive the prefix hypothesis PH(w_1,w_2).
Rule notation: $\dfrac{\text{valid}(w_1 w_2 ... w_n),\text{valid}(w_2 ... w_n)}{\text{PH}(w_1,w_2),\text{valid}(w_2 ... w_n)}$

Since both *clans* and *lans* are valid words we can use this inference rule to derive the prefix hypothesis PH(*c*,*l*). This hypothesis can in turn be used in combination with the background knowledge prefix rule and the fact that *clan* is valid to prove that *lan* is valid. The latter word was not among the observations so this is an example of the generalization capabilities of an extra prefix hypothesis. Note that the background knowledge prefix rule can be used in two directions: with PH(c,l) and BWH(lan) we can prove that *clan* is valid and with PH(c,l) and BWH(clan) we can prove that *lan* is valid.

The suffix hypothesis inference rule has a similar format as the prefix hypothesis inference rule:

SUFFIX HYPOTHESIS INFERENCE RULE
If W=$w_1 ... w_{n-1} w_n$ is a valid word and $w_1 ... w_{n-1}$ is a valid word as well then we will derive the suffix hypothesis SH(w_{n-1},w_n).
Rule notation: $\dfrac{\text{valid}(w_1 ... w_{n-1} w_n),\text{valid}(w_1 ... w_{n-1})}{\text{SH}(w_{n-1},w_n),\text{valid}(w_1 ... w_{n-1})}$

In our example model we can use this suffix hypothesis inference rule with either of the valid word pairs *clan* and *clans* or *lan* and *lans* for deriving the suffix hypothesis SH(*n*,*s*). After having removed redundant basic word hypotheses, our final model for the observations *clan*, *clans* and *lans* will consist of the background knowledge defined in the previous section in combination with one basic word hypothesis BWH(*lan*), one prefix hypothesis PH(*c*,*l*) and one suffix hypothesis SH(*n*,*s*).

Now we have defined a method for deriving hypotheses for a specific format of background knowledge and observations. The derivation method uses only one of the four hypotheses space reduction methods mentioned in (Muggleton 1995): the observations are ground literals. We did not limit the number of derivable hypotheses to one. On the contrary we want to derive as many valid hypotheses as possible because they will improve the generalization capabilities of the final model. The derived hypotheses were neither restricted to the most general hypotheses nor to the ones that compressed the observations as much as possible. Instead of that we have limited the inference rules in such a way that one rule can only derive one particular hypotheses as the explanation of a single item of data or a pair of data. The inference process is deterministic and we will accept all the hypotheses it derives.

We will not use the two general purpose ILP software packages available: Golem and Progol (Muggleton 1995). These impose restrictions on either the background knowledge or the observations which we cannot impose on our problem. Golem tries

to limit the number of acceptable hypotheses by restricting the background knowledge to ground knowledge. This means that there are no variables allowed in the Golem background knowledge. In order to be able to process background knowledge that has been derived using clauses containing variables, Golem provides a method for converting parts of that knowledge to variable-free clauses. Our background knowledge contains clauses with variables. Since we are working with finite strings the background knowledge can in principle be converted to variable free rules. However this will make it so large that it will be unwieldy to work with in practice. The background knowledge cannot be converted to a usable format that is acceptable for Golem and this makes Golem unusable for our data.

Progol does not have the background knowledge restriction of allowing only variable-free clauses. However the Progol version that we have evaluated required both positive and negative examples in order to generate reasonable output. Muggleton has described a theoretical extension to enable Progol to learn from positive data only (Muggleton 1995). As far as we know this extension has not been added to the software yet.[6] Because we only want to supply positive examples to our learning problem Progol was inadequate for tackling this problem.

2.4 The hypothesis models and grammar theory

The models that we will derive for our data consist of three types of rules: basic word hypotheses, prefix hypotheses and suffix hypotheses. It is theoretically interesting to find out to which grammar class these models belong. In this section we will show that the behavior of our learning models can be modeled with regular grammars. These grammars generate exactly the same class of languages as finite state automata.

Regular grammars consist of rules that have the format $S \rightarrow xA$ or $A \rightarrow y$ (Hopcroft et al. 1979). Here A and S are non-terminal character while x and y is a terminal character. The grammars generate strings by applying a sequence of rules usually starting with a start non-terminal S. For example, with the two rules presented here we could change the start token S in xA with the first rule and successively in xy with the second rule. The result is a string of terminal symbol and our grammar has generated this string.

Words in our rule-based model are built by starting from a nucleus and subsequently adding prefix characters and suffix characters. We will derive an equivalent regular grammar which builds words by starting with the first character and subsequently adding extra characters or character strings until the word is complete. The regular grammar will make use of non-terminal symbols to simulate context dependencies within the word. Each character x_i will have two corresponding non-terminal symbols: a prefix symbol P_{x_i} and a suffix symbol S_{x_i}.

In our earlier examples we have used the word *clans*. This word will be build in our rule-based model as follows: start with the basic word *lan*, add the character c before it to obtain *clan* and add the character s behind it to obtain *clans*. In the

[6] A new version of Progol which allowed learning with positive training data only was released in 1997.

equivalent regular grammar we will start with cP_l, replace P_l with $lanS_n$ to obtain $clanS_n$, replace S_n with sS_s to obtain $clansS_s$ and replace S_s with the empty string to obtain *clans*.

There are a number of constraints on the intermediate strings that the regular grammar produces in derivations. First, the intermediate strings will always consist of a sequence of terminal symbols (*a*, *b*, *c* etc.) followed by one optional non-terminal symbol (P_{x_i} or S_{x_i}). Second, the symbol P_{x_i} can only be replaced with an intermediate string that starts with x_i. Third, when an intermediate string ends in $x_iS_{x_j}$ then x_i and x_j will always be the same character. Fourth, the symbol S_{x_i} can only be replaced with a string of the format $x_jS_{x_j}$ or with the empty string.

In the previous section we have derived a small rule-based model containing three hypotheses: BWH(*lan*), PH(*c*,*l*) and SH(*n*,*s*). We will define conversion rules which can be used for converting this model to a regular grammar containing the rules:

BWH(*lan*)	$S \rightarrow lanS_n$	(1)
	$P_l \rightarrow lanS_n$	(2)
	$S_n \rightarrow e$	(3)
PH(*c*,*l*)	$S \rightarrow cP_l$	(4)
SH(*n*,*s*)	$S_n \rightarrow sS_s$	(5)
	$S_s \rightarrow e$	(6)

(*e* stands for the empty string) This grammar produces exactly the same strings as the original rule based model: *lan*, *clan*, *lans* and *clans*. For example, we can generate the string *clans* by applying rule (4) to the start symbol *S* (result cP_l) after which we successively apply rules (2) ($clanS_n$), (5) ($clansS_s$) and (6) to obtain the target string *clans*.

Now that we have seen an example for the regular grammar at work we can define the necessary rules for converting the hypotheses to regular grammar rules. We will start with a conversion rule for suffix hypotheses:

> SUFFIX HYPOTHESIS CONVERSION RULE
> A suffix hypothesis $SH(x_i,x_j)$ is equivalent to the set of regular grammar rules $\{ S_{x_i} \rightarrow x_jS_{x_j} , S_{x_j} \rightarrow e \}$.

The suffix hypothesis $SH(x_i,x_j)$ allows appending x_j to a word that has final character x_i. In the regular grammar model an intermediate string of which the final terminal symbol is x_i will initially be followed by the non-terminal S_{x_i}. Thus we can simulate the suffix hypothesis by creating a rule which replaces this non-terminal symbol with $x_jS_{x_j}$. The S_{x_j} non-terminal symbol in the added string makes possible the addition of extra characters. However, if we want to create proper words that end in x_j we need to be able to remove the non-terminal symbol from the intermediate string. Therefore we have included the second rule which replaces S_{x_j} with the empty string.

The following conversion rule can be used for prefix hypotheses:

> PREFIX HYPOTHESIS CONVERSION RULE
> A prefix hypothesis $PH(x_i,x_j)$ is equivalent to the set of regular grammar rules $\{ S \rightarrow x_iP_{x_j} , P_{x_i} \rightarrow x_iP_{x_j} \}$

The prefix hypothesis PH(x_i,x_j) will allow placing character x_i before a word that starts with the character x_j. The regular grammar works the other way around: it will append the character x_j to a prefix string which ends in x_i.[7] We can tell that an intermediate string is a prefix string by examining the final token. If that token is a prefix non-terminal symbol then the string is a prefix string and otherwise it is not.

While converting the prefix hypothesis to the regular grammar we have to distinguish two cases. First, the added character x_i can be the first character of the word. This case has been taken care of by the first rule which replaces the starting symbol S by $x_i P_{x_j}$. The symbol P_{x_j} makes sure that the next character of the string will be the character x_j. Note that according to our second intermediate string format constraint we can only replace P_{x_j} that starts with the character x_j.

The second possible case is that the added character x_i is a non-initial character of the word. In that case we will add this character to a prefix string which contains final symbol P_{x_i}. The second rule will replace this symbol by $x_i P_{x_j}$. Again we add P_{x_j} to make sure that the next character will be x_j.

The third conversion rule can be used for basic word hypotheses:

> BASIC WORD HYPOTHESIS CONVERSION RULE
> A basic word hypothesis BWH(x_i...x_j) is equivalent to the set of regular grammar rules { $S \rightarrow x_i ... x_j S_{x_j}$, $P_{x_i} \rightarrow x_i ... x_j S_{x_j}$, $S_{x_j} \rightarrow e$ }

The basic word hypothesis BWH(x_i...x_j) defines that the string $x_i ... x_j$ is valid. It is not trivial to add this hypothesis to the regular grammar because of the differences in processing between the grammar and the rule-based models. In the grammar the basic word will be added to a prefix string. An extra suffix non-terminal needs to be added to the string as well.

Just as with the prefix hypothesis conversion we need to distinguish between two cases of basic word hypothesis processing when we want to convert the hypothesis to regular grammar rules. First the basic word can be the initial part of the word we are building. In that case no prefix rules will be used for building the word. The first rule takes care of this case. It replaces the starting symbol S with the string $x_i ... x_j S_{x_j}$. The suffix non-terminal S_{x_j} has been included in this string to allow that other characters can be added to the string by using suffix rules. It belongs to the previous terminal symbol x_j and this is in accordance with the third constraint on the format of the intermediate strings. Since the basic word on its own is valid as well we need the possibility to remove the non-terminal symbol S_{x_j} from the word. That is why the third rule has been included here. It can be used for replacing the non-terminal symbol with the empty string.

If the basic word is not the initial part of the word then we will add the basic word to a prefix string. This string will contain a final prefix non-terminal symbol which specifies that the next character needs to be x_i. The second rule takes care of replacing

[7] Actually things are a little bit more complex than this. In the regular grammar the prefix hypothesis PH(x_i,x_j) will be modeled by rules that add the character x_i in an x_i context and allow the next character of the intermediate string to be x_j.

this symbol P_{x_i} by $x_i...x_jS_{x_j}$. Again we include the non-terminal suffix symbol S_{x_j} to have the possibility for adding extra suffix characters to the word. If no suffix characters are necessary then third rule can be used for removing the symbol again.

We have presented an algorithmic method for converting a model expressed in hypotheses to a regular grammar. The existence of such a method proves that the behavior of our models can be simulated with regular grammars and finite state automata.

3 Experiments with Inductive Logic Programming

In this section we will describe our initial experiments with Inductive Logic Programming (ILP). First we will outline the general setup of these experiments. After that we will present the results of applying ILP to our orthographic and phonetic data. That presentation will be followed by a description of the application of ILP to the same data but while starting the learning process from basic phonotactic knowledge. The section will be concluded with a discussion of the results of the experiments.

3.1 General experiment setup

We have used the three inference rules described in section 2.3 for deriving a rule-based model for our orthographic data and a model for our phonetic data. The input data for the learning algorithm consisted of the background knowledge rules described in section 2.2 and observation strings which were the words in the training corpus. The background knowledge rules contain information about the structure of words. However we have shown in section 2.2 that this information is implicitly present in HMMs so encoding it in the background knowledge rules does not give ILP an advantage over HMMs.

The format of the training corpora in these experiments is slightly different from the previous two chapters. In HMMs it was necessary to add an end-of-word token and a start-of-word token to each word. In these experiments both word boundary tokens have been omitted. Processing the data with these tokens would also have been possible but it would have required more complex background suffix rules and more complex inference rules. We have chosen to keep our rules as simple as possible. Processing data with word boundary tokens and more complex rules will lead to models with the same explanatory power as processing data without end-of-word tokens and simple suffix rules. The choice of dropping the word boundary tokens here has no influence on our goal to make the three learning model experiments as comparable as possible.

The ILP hypotheses inference algorithm will process the data in the following way:

1. Convert all observations to basic word hypotheses.
2. Process all basic words, one at a time. We will use the symbol W for the word being processed and assume that W is equal to the character sequence $w_1w_2...w_{n-1}w_n$. We will perform the following actions:
   (a) If $w_2...w_{n-1}w_n$ is a valid word then derive the prefix hypothesis PH(w_1,w_2) and remove the basic word hypothesis for W.
   (b) If $w_1w_2...w_{n-1}$ is a valid word then derive the suffix hypothesis SH(w_{n-1},w_n) and remove the basic word hypothesis for W.
   (c) If the prefix hypothesis PH(w_1,w_2) exists then derive the basic word hypothesis BWH($w_2...w_{n-1}w_n$) and remove the basic word hypothesis for W.
   (d) If the suffix hypothesis SH(w_{n-1},w_n) exists then derive the basic word hypothesis BWH($w_1w_2...w_{n-1}$) and remove the basic word hypothesis for W.
3. Repeat step 2 until no new hypotheses can be derived.

Steps 1, 2(a) and 2(b) are straightforward applications of the inference rules for basic words, prefix hypotheses and suffix hypotheses which were defined in section 2.3. The steps 2(c) and 2(d) are less intuitive applications of the background knowledge rules for prefixes and suffixes (see section refsec-ch4-background) in combination with the basic word inference rule. In the background knowledge suffix rule we have defined that $w_1...w_{n-1}$ will be a valid word whenever $w_1...w_{n-1}w_n$ is a valid word and a suffix hypothesis SH(w_{n-1},w_n) exists. This is exactly the case handled by step 2(d) and because of the fact that $w_1...w_{n-1}$ is a valid word we may derive BWH($w_1...w_{n-1}$) by using the basic word inference rule. Step 2(c) can be explained in a similar way.

The steps 2(c) and 2(d) will be used to make the basic words as short as possible. This is necessary to enable the algorithm to derive all possible prefix and suffix hypotheses. Consider for example the following intermediate configuration hypotheses set:

BWH(yz)
BWH(yx)
SH(y,z)

By applying step 2(d) we can use SH(y,z) and BWH(yz) to add the basic word hypothesis BWH(y) and remove BWH(yz). On its turn this new basic word hypothesis in combination with BWH(yx) can be used for deriving the suffix hypothesis SH(y,x). In this example shortening a basic word has helped to derive an extra suffix hypothesis. We cannot guarantee that the new hypothesis will be correct. However, since we

		number of hypotheses			accepted	rejected
data type	rounds	basic word	prefix	suffix	positive strings	negative strings
orthographic	4	30	347	327	593 (98.8%)	379 (63.2%)
phonetic	3	54	383	259	593 (98.8%)	517 (86.2%)

Figure 4.4: The performance of the ILP algorithm and the models generated by this algorithm. The ILP algorithm converts the training strings in models that contain approximately 700 rules. The models perform well on the positive data (more than 98% was accepted) but poorly on the negative data (rejection rates of 63% and 86%).

do not have negative data available which can be used for rejecting new hypotheses we cannot do anything else than accept all proposed hypotheses.

The ILP hypotheses inference algorithm will repeat step 2 until no more new hypotheses can be derived. We will call each repetition of step 2 a training round and list the number of required training rounds in the experiment results.

3.2 Handling orthographic and phonetic data

We have used ILP to derive a rule-based model for our training data of 5577 orthographic strings. We used the hypothesis inference algorithm of the previous section which was based on the inference rules defined in section 2.3. The background knowledge consisted of the three background knowledge rules defined in section 2.2 and the training words were used as observations. The algorithm required four training rounds before the hypothesis set stabilized. The final model consisted of 30 basic word hypotheses, 347 prefix hypotheses and 327 suffix hypotheses (see figure 4.4).

The final rule-based model was submitted to the same tests as the models which were generated in the previous chapters. It was tested by making it evaluate 600 positive test strings which were not present in the training data and 600 negative test strings. The model performed well on the positive test data. It accepted 593 words (98.8%) and rejected only 7: *d*, *fjord*, *f's*, *q's*, *schwung*, *t's* and *z*. The rule-based model achieved a lower score on the negative test data. It was only able to reject 379 of the 600 strings (63.2%) Some examples of accepted invalid strings are *bswsk*, *kwrpn*, *ntesllt*, *rdtskise* and *ttrpl*.

After having applied the ILP algorithm for orthographically encoded words we have used it for deriving a rule-based model for our phonetic data. The algorithm started with 5084 observations and required three training rounds before the hypothesis set stabilized. The final model consisted of 54 basic word hypotheses, 383 prefix hypotheses and 259 suffix hypotheses (see table 4.4). It performed equally well on the correct test words as the orthographic model accepting 593 words (98.8%) and rejecting only 7 words: *fjord [fjɔrt]*, *fuut [fyt]*, *square [skwɛ:r]*, *squares [skwɛ:rs]*, *schmink*

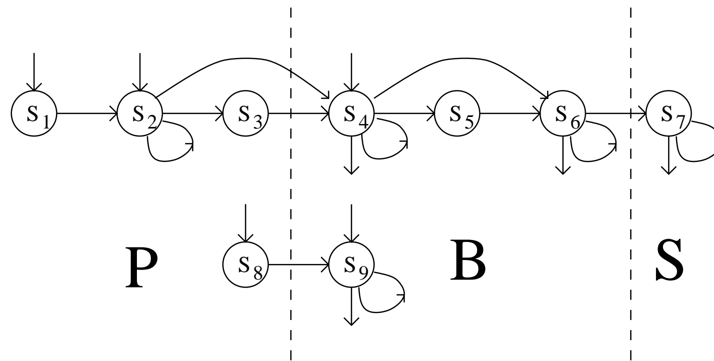


Figure 4.5: An adapted version of the initial HMM model for orthographic data, based on the Cairns and Feinstein model. The original model was presented in figure 2.18 of chapter 2. The model has been divided in three parts: a part P in which the prefix rules operate, a part B generated by the basic word hypotheses and a part S in which the suffix hypotheses work. The new states s_8 and s_9 are copies of s_1 and s_2 and take care of the production of words that do not contain vowels.

[ʃmiŋk], *schminkt [ʃmiŋkt]* and *schwung [ʃwuŋ]*. The phonetic model performed worse on the negative test strings but its performance was better than the performance of the orthographic model on the negative strings. It was able to reject 517 of 600 strings (86.2%).[8]

3.3 Adding extra linguistic constraints

In the previous chapters we have used the phonetic model by (Cairns and Feinstein 1982) as a model for possible innate linguistic knowledge. We will use this model in our ILP experiments as well. One problem we have to solve is the conversion of the Cairns and Feinstein model to the ILP rule structure. For this purpose we will use our the modified bigram HMM initialization model shown in figure 2.18 of chapter 2. This model consists of a set of states with a limited number of links and with restrictions on the characters that can be produced by each state. We will restructure this model and derive some usable constraints from it.

We want to divide the model in three parts: one part in which only prefix hypotheses operate, one part in which only suffix hypotheses work and one part that is generated by basic word hypotheses. This division is shown in figure 4.5: part P is the part for the prefix hypotheses, part B is for the basic word hypotheses and part S

[8]When we take into account that 26 of the negative strings are reasonable (see section 4.4 in chapter 2) then the phonetic model rejects 514 of 574 negative strings (89.5%).

is for the suffix hypotheses. Each character production by states in the parts P and S corresponds to a set of prefix or suffix hypotheses. The states s_5 and s_6 have been put in the basic word hypothesis part because s_6 is able to produce vowels and we want to produce all vowels in the basic word hypotheses.

The division of the model caused one problem. The original model contained an exit link from state s_2. This link would make it impossible to include state s_2 in the prefix hypothesis part. States with exit links must be part of either the basic word hypothesis part or the suffix hypothesis part. State s_2 in figure 2.18 is on its own capable of producing a word like *t* which is present in the orthographic data. However, in our ILP rule models the use of a basic word hypothesis is obligatory in the production of a valid word. Having an exit link in state s_2 would allow word productions that do not include the use of a basic word hypothesis.

We have solved this problem by splitting the model in two parallel finite state automata (figure 4.5). The states s_1 and s_2 and their links have been copied to two new states s_8 and s_9. These new states will take care of the production of words that were produced by using the exit link from state s_2. This made it possible to remove this exit link from state s_2. Since s_9 has been put in the basic word hypothesis part, the words produced by s_8 and s_9 will also require the application of a basic word hypothesis. The larger automaton will produce all words that used the exit links from states other than s_2. Therefore there is no necessity for links from s_9 to s_3 or s_4.

Dividing the model in two parts has solved the problem we had with state s_2. We can put the states s_1, s_2, s_3 and s_8 in the prefix hypothesis part because all exit links from this group of states go to states in the basic word hypothesis part. That part will include states s_4, s_5, s_6 and s_9. The suffix part contains only one state: s_7.

The finite state automaton of figure 4.5 is equivalent to the finite state automaton of figure 2.18 in chapter 2. However, for learning purposes there are differences between the two. Each character production in the P and the S parts can be modeled with one prefix or one suffix hypothesis. But we cannot produce every single character in the B part with one basic word hypothesis because basic words consists of character sequences and we do not have the opportunity for combining basic word hypotheses in the ILP rule model. Therefore we will use one basic word for modeling the production of a character sequence by a group of cells. This may cause problems when there are internal parts which repeat themselves an arbitrary number of times. The behavior invoked by the self-links from the states s_4, s_6 and s_9 cannot be modeled with the basic word hypotheses. The ILP learning process cannot generate models with extendible basic word hypotheses and this means that the generalization capabilities of the resulting rule models will be weaker than those of the previously used HMMs.

Now that we have changed the Cairns and Feinstein initialization model to the structure that we are using in the chapter, we can attempt to derive usable constraints from this model. The Cairns and Feinstein model imposes constraints on the characters that can be generated by a particular state. We have defined these constraints explicitly for our orthographic data in the B matrix in figure 2.17 of chapter 2: the vowels *a*, *e*, *i*, *o*, *u* and the quote character ' can only be generated by the states s_4 and s_6, the ambiguous vowel/consonant *y* can be generated by any state and all other

characters are consonants which can be generated by any state except s_4. The new states s_8 and s_9 are consonant states: they can generate any character except the six characters *a*, *e*, *i*, *o*, *u* and ' (in this chapter we will regard the quote character as a vowel).

When we inspect the model with these character production constraints in mind we can make two interesting observations. First, the prefix hypothesis states cannot produce the characters *a*, *e*, *i*, *o*, *u* and '. We will call these characters PURE VOWELS. Since the characters produced by these states are put before a word by a prefix hypotheses, this means that prefix hypotheses cannot add a pure vowel prefix to a word. Second, the suffix hypothesis state cannot produce a pure vowel. A character produced by this state is a character appended to a word by a suffix hypothesis. This means that suffix hypotheses cannot append a pure vowel to a word. We can summarize these two observations in the following two rules:

> PREFIX HYPOTHESIS CONSTRAINT
> In a prefix hypothesis PH(I,S) the character I that is appended to a word cannot be a pure vowel.
>
> SUFFIX HYPOTHESIS CONSTRAINT
> In a suffix hypothesis SH(P,F) the character F that is appended to a word cannot be a pure vowel.

It is not possible to derive a similar constraint for the basic word hypotheses because these can contain both vowels and consonants. The derivation presented here applies only to orthographic data. In a similar fashion one can take the initial phonetic model from figure 2.22 in chapter 2, generate an adapted model like presented in figure 4.5 and derive similar constraints for prefix and suffix hypotheses.[9] Our phonetic data contains 18 vowels.

We have repeated our ILP experiments for deriving rule-based models for our orthographic and our phonetic data by using the prefix and the suffix hypothesis constraints presented in this section. Apart from these extra constraints the experiment setups were the same as described in the previous section The resulting models were submitted to our standard test sets of 600 correct words and 600 incorrect strings. The results of these tests can be found in figure 4.6.

The models needed approximately the same number of training rounds to stabilize as in our previous experiments. Because of the constraints on the format of the prefix and the suffix hypotheses, the initialized models contain fewer prefix hypotheses and fewer suffix hypotheses. This means that fewer words have been divided in smaller parts and as a result of that the models contain more basic word hypotheses. The size differences between these models and the previous ones are largest for the orthographic data.

[9] An additional constraint can be derived for phonetic data: the basic word hypotheses cannot consist of a mixture of vowels and consonants. We did not use this constraint because we expected it would cause practical problems in the learning phase.

data type	rounds	number of hypotheses			accepted positive strings	rejected negative strings
		basic word	prefix	suffix		
orthographic	4	128	166	178	586 (97.7%)	564 (94.0%)
phonetic	2	64	324	207	593 (98.8%)	565 (94.2%)

Figure 4.6: The performance of the ILP algorithm with the prefix and the suffix hypothesis constraints and the models generated by this algorithm. The ILP algorithm converts the training strings in models that contain fewer rules than the previous models (472 and 595 compared with approximately 700). The models perform well on the positive test data (best rejection rate 2.3%) and a little worse on the negative test data (rejection rates of about 6%).

The added constraints during learning make the ILP process generate better models. The orthographic model performs worse in accepting positive test data (97.7% compared with the earlier 98.8%) but remarkedly better in rejecting negative data (94.0% versus 63.2%). The phonetic model performs exactly as well in accepting positive test data (98.8%) and a lot better in rejecting negative data (94.2% versus 86.2%).[10]

3.4 Discussion

The orthographic model derived by the ILP algorithm without constraints for orthographic data performs quite poorly in rejecting negative test strings. It has become too weak. The fact that this model consist of a set of rules gives us the opportunity to inspect it and find out why exactly it is making errors. This is an advantage of rule-based models over statistical and connectionist models. In the experiments described in the previous section we did not have the opportunity to correct models by changing their internal structure.

The orthographic model accepts the incorrect consonant string *kwrpn*. The model can use only one set of rules for proving this string: the four prefix rules PH(*k*,*w*), PH(*w*,*r*), PH(*r*,*p*) and PH(*p*,*n*) and the basic word rule BWH(*n*). The first two prefix rules are correct as we can see from the two correct Dutch words *kwarts* and *wrakst*. The third prefix rule is wrong[11] and the fourth one is a rare one for Dutch but correct. The basic word hypothesis is strange. Like in English it is possible to say that *n is the 14th character in the alphabet*. However with consonants as basic words we cannot

[10]When we take into account that 26 of the negative strings are reasonable (see section 4.4 in chapter 2) then the phonetic model rejects 562 of 574 negative strings (97.9%).

[11]Incorrect hypotheses are caused by the presence of single consonants in the training data. For example, in combination with the word *in* the single consonant word *n* would cause the incorrect prefix hypothesis PH(i,n) to be derived.

expect to be able to create a good model.

Unfortunately all single characters are present in the orthographic data: 21 in the training corpus and 5 in the test corpus. We had expected the final basic word hypothesis set to include the six Dutch vowels and a few rare vowel sequences which could not be made smaller by the learning algorithm. This was indeed the case but apart from that the basic word hypotheses contained 17 single consonants. Most of them were present in the training data as complete words. We supposed that the latter fact caused the consonants to appear in the basic hypothesis set. In order to test this we have removed the single consonant words from the training words and performed an extra training session. However all consonants returned as basic words in this extra orthographic model.

We have inspected the training data to find out which words did not fit in the simplest word model we could think of. This model assumes that every word contains a vowel with possibly adjacent vowels and an arbitrary number of consonants in front or behind the vowels. The following word types did not fit in this model:

1. Single character consonant words (16 items). All characters except for *a*, *e*, *i*, *o*, *u*, *y* and ' have been regarded as consonants. Examples of words in this class are *n* and *t*.

2. Words with two or more vowel groups (111 items). Nearly all these words were loan words. Examples of this class are *toque* en *leagues*

3. Multiple character words without vowels (3 items). These were the three interjections which were present in our training corpus: *st*, *sst* and *pst*.

When we remove all three data types from the training corpus and apply the ILP learning algorithm we obtain an orthographic model with only 19 basic word hypotheses, 188 prefix rules and 198 suffix rules. This model accepts 97.6% of the complete training data, accepts 97.0% of the positive test strings and rejects 96.7% of the negative test data. This is an improvement with respect to our orthographic experiments without extra constraints. However we are looking for models which accept all training data and thus this model is unacceptable.

The problems in the phonetic model that was generated without using constraints are not that obvious. The model contains five single consonant basic word hypotheses, 55 prefix hypotheses which append a vowel and 51 suffix hypotheses which add a vowel to a word. The data contains the two interjections st and pst, one single consonant word *s* but no words with more than one vowel group. When we remove these three words from the training data and run the ILP algorithm one more time then we obtain a model which accepts 99.9% of the training data, accepts 98.8% of the positive test data and rejects 94.7% of the negative test data. The model contains 324 prefix rules, 207 suffix rules and 62 base rules. Performance and model size are almost exactly the same as for the phonetic model with linguistic initialization.

We may conclude that application of ILP to our learning task leads to a reasonable performance. The models that are generated by the ILP algorithm without the

constraints derived from the model of Cairns and Feinstein perform worse than the models generated while using these constraints. The problems of the earlier models can be explained by a few problematic words. Application of ILP with the extra linguistic constraints is more robust with respect to these words and thus generates better models.

4 Alternative rule-based models

In the previous section we have described how ILP can be used for deriving rule-based orthographic and phonetic models. In this section we will modify the internal structure of these models. First we will create more elaborate models, that is models which have a richer internal structure. We will show how these models can be build by using ILP and perform some learning experiments with them. After that we will decrease the size of the models by making the rules process character sets rather than single characters.

4.1 Extending the model

In the previous section we have used the Cairns and Feinstein model for initializing a rule-based model. We have seen that our ILP models do not have an internal structure that is as rich as the Cairns and Feinstein model. The latter model divides the word production process in seven different stages which correspond with the seven states in the initial model. Our rule-based models only contain three different stages: one for prefix hypotheses, one for suffix hypotheses and one for basic word hypotheses. We have mentioned in section 3.3 that this restriction has a negative influence on the generalization capabilities of the models.

We want to test a more elaborate rule-based model to see if it performs better than our current three-state model. We will aim for a similar structure as the Cairns and Feinstein model with seven states but we want to keep our concepts of prefix hypotheses, suffix hypotheses and basic word hypotheses. Therefore we have defined the following extended hypotheses concepts:

> EXTENDED BASIC WORD HYPOTHESIS
> An extended basic word hypothesis BWH($w_1...w_n$,s$_i$) defines that the string $w_1...w_n$ can be produced in state s$_i$.
> Rule notation: producible($w_1...w_n$,s$_i$)←BWH($w_1...w_n$,s$_i$).
> EXTENDED SUFFIX HYPOTHESIS
> An extended suffix hypothesis SH(w_{n-1},w_n,s$_i$) defines that when a string $w_1...w_{n-1}$ can be produced in a predecessor state of state s_i then the string $w_1...w_{n-1}w_n$ can be produced in state s_i.
> Rule notation: producible($w_1...w_{n-1}w_n$,s$_i$)←SH(w_{n-1},w_n,s$_i$),
> producible($w_1...w_{n-1}$,s_j),
> predecessor(s$_j$,s$_i$).

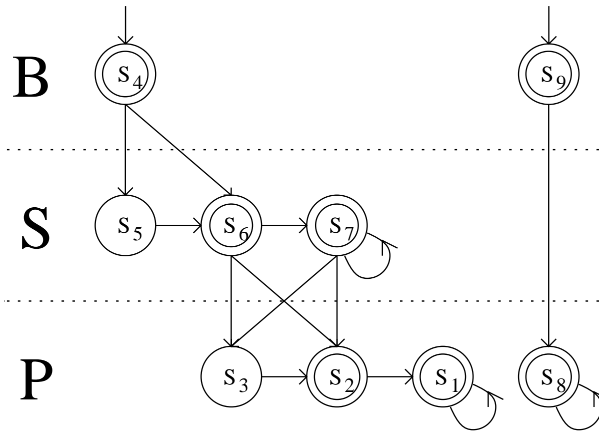


Figure 4.7: An adapted version of the initial HMM model for phonetic data, based on the Cairns and Feinstein model. The original model was presented in figure 2.22 of chapter 2. The model has been divided in three parts: a part P in which the extended prefix hypotheses operate, a part S in which the extended suffix hypotheses work and a part B generated by the extended basic word hypotheses. States s_8 and s_9 are copies of s_1 and s_2 that take care of the production of words that do not contain vowels. All states except state s_3 and state s_5 are final states. The connections from s_4 to s_3 and s_2 have not been included in this diagram.

> EXTENDED PREFIX HYPOTHESIS
> An extended prefix hypothesis PH(w_1,w_2,s_i) defines that when a string $w_2...w_n$ can be produced in a predecessor state of state s_i then the string $w_1w_2...w_n$ can be produced in state s_i.
> Rule notation: producible($w_1w_2...w_n$,s_i)←PH(w_1,w_2,s_i),
> producible($w_2...w_n$,s_j),
> predecessor(s_j,s_i).

Furthermore, final state hypotheses are necessary for defining which states can be final states since only a few states will be allowed to act as a final state. Final state hypothesis FS(s_i) defines that processing may end in state s_i. There is no equivalent definition necessary for the concept of initial state because these will implicitly be defined by the basic word hypotheses. Nothing can be produced before a basic word and processing a word can only start with processing a basic word. Therefore any state present in an extended basic word hypothesis will automatically be an initial state.

Figure 4.7 shows an initial model that uses extended hypotheses. Like the model shown in figure 4.5 it has been divided in three parts for the three types of hypotheses. Note that the processing order has been changed slightly. The model will start processing basic words, continue with working on suffix characters and finally deal with

the prefix characters. This unusual processing order was caused by our wish to obtain a model that was closely related to the three hypothesis types. The more intuitive processing order prefix characters - basic word - suffix characters was not possible because basic words impose restrictions on prefix characters. In a string generation model processing must start with the basic word.

Example: suppose that we want the model to produce the word *bAst* and we know that the hypotheses BWH(A,s_4), PH(b,A,s_2), SH(A,s,s_6) and SH(s,t,s_7) are available. The model would start in state s_4 and produce *A*. After this it would continue with state s_6 (*As*) and state s_7 (*Ast*). Processing would end in state s_2 (*bAst*). The word is valid since processing has ended in a final state and the complete word has been produced.

4.2 Deriving extended hypotheses

Deriving extended hypotheses is more difficult than deriving standard hypotheses. For example, we cannot use the suffix inference rule of section 2.3 for deriving a suffix hypothesis from basic word hypotheses BWH(clans) and BWH(lans) because we need to define the state which produces the *c*. The basic words do not provide a clue to which state would be correct.

We will design a method for deriving the rules based on the assumption that we have some general initial model to start with. The task of the ILP learning process will be to fill in the details of the model. The initial model will be a quadruple consisting of a set of states, a set of predecessor state definitions, a set of final state definitions and a set of character classes that can be produced by the states. An example of such a model is shown in figure 4.7. The states and the links have been shown in the figure. The characters that can be produced by the states are vowels for state s_4 and consonants for all other states (see also the B matrix in figure 2.21 in chapter 2).

These constraints on the production of characters leave some processing freedom. For example, the suffix characters of our example word *bAst* can be produced by two different state combinations. The *s* can be produced by state s_5 and the *t* by s_6. Alternatively the *s* can be produced by state s_6 and the *t* by s_7. In the Cairns and Feinstein model for Dutch the second option is the only one that is permitted. In order to enable the ILP algorithm to find this solution we need to supply it with extra information. An example of such extra information could be dividing the consonants in subclasses and putting more restrictions on the characters that the states can produce. However, we do not want to make an extra partition in the character sets because we have not done something like that in the experiments with HMMs and SRNs. Using a finer division here would provide the ILP algorithm information the other two algorithms did not have and this would make a performance comparison unfair.

We have chosen to accept both suffix generation possibilities. This means that for generating the word *bAst* we would derive the following set of hypotheses:

BWH(A,s_4)
PH(b,A,s_2)

SH(A,s,s_5)
SH(s,t,s_6)
SH(A,s,s_6)
SH(s,t,s_7)

The derivation process of the extended hypotheses will contain the following steps:

1. Take some initial model of states, predecessor state definitions, final state definitions and producible character classes. The set of hypotheses starts empty.

2. Take a word from the training data and derive all hypotheses which can be used for explaining the word with the initial model. Add these hypotheses to the set of hypotheses

3. Repeat step 2 until all words in the training data have been processed.

4. The result of this ILP process is the initial model combined with the set of hypotheses.

In this derivation process we will use modified versions of the background knowledge rules we have defined in section 2.2 for prefix hypotheses, suffix hypotheses and basic words. We also need final state definitions, predecessor state definitions and a validity definition. Furthermore we will apply an extended hypothesis inference rule which will replace the three inference rules we have defined in section 2.3.

EXTENDED BACKGROUND KNOWLEDGE BASIC WORD RULE
The existence of a basic word hypothesis BWH($w_1...w_n$,s_i) implies that string $w_1...w_n$ can be produced by state s_i.
Rule notation: producible($w_1...w_n$,s_i)$\leftarrow$ BWH($w_1...w_n$,s_i)

EXTENDED BACKGROUND KNOWLEDGE SUFFIX RULE
Suppose there exists a suffix hypothesis SH(w_{n-1},w_n,s_i).
In that case the fact that string $w_1...w_{n-1}$ can be produced in a predecessor of state s_i implies that $w_1...w_{n-1}w_i$ can be produced in s_i.
Rule notation: producible($w_1...w_{n-1}w_n$,s_i)$\leftarrow$ SH(w_{n-1},w_n,s_i),
producible($w_1...w_{n-1}$,s_j),
predecessor(s_j,s_i)

EXTENDED BACKGROUND KNOWLEDGE PREFIX RULE
Suppose there exists a prefix hypothesis PH(w_1,w_2,s_i).
In that case the fact that $w_2...w_n$ can be produced in a predecessor of state s_i implies that $w_1w_2...w_n$ can be produced in state s_i.[12]
Rule notation: producible($w_1w_2...w_n$,s_i)$\leftarrow$ PH(w_1,w_2,s_i),
producible($w_2...w_n$,s_j),
predecessor(s_j,s_i)

[12] Because of the processing order shown in figure 4.7 prefix character states can have predecessor states.

FINAL STATE DEFINITION
The existence of a final state definition finalState(s_i) implies that state s_i is a final state and vice versa.
PREDECESSOR STATE DEFINITION
The existence of a predecessor definition predecessor(s_i,s_j) implies that state s_i is a predecessor of state s_j and vice versa.
VALIDITY DEFINITION
A string is valid according to a model consisting of a set of states, a set of predecessor state definitions, a set of final state definitions, a set of character classes produced by the states and a set of hypotheses if and only if the string can be produced in one of the final states of the model.
EXTENDED HYPOTHESIS INFERENCE RULE
Any ground prefix hypothesis, suffix hypothesis and basic word hypothesis that can be used for proving that a string in the training data is valid according to the initial model should be derived.

The extended hypothesis inference rule derives sets of hypotheses when they can be used for producing a string in the model. It is difficult to capture this in the rule notation and therefore no rules have been included in the definition of this inference rule.

4.3 Experiments with the extended model

We have used ILP for deriving models for our orthographic data and our phonetic data. For each data type we have performed two experiments: one that started from a random model and one that started from a linguistic model derived from the Cairns and Feinstein model. The linguistic model for phonetic data can be found in figure 4.7. The linguistic model for orthographic data is similar to the model shown in figure 4.5 but this model has the same processing order as the phonetic model: first prefix hypothesis states, then suffix hypotheses states and finally the basic word hypothesis states. In the initial orthographic model the states s_4, s_5 and s_6 have been combined into one state which is capable of producing multicharacter strings.

Just like in the previous experiments we wanted to compare initialized extended models with non-initialized extended models in order to be able to measure the influence of the initialization. Constructing the initial models for the two non-initialized experiments was a non-trivial task. We wanted the models to contain some random initialization like the random initialization models we have used for HMMs and SRNs. However, our extended rule-based models do not contain numeric values which we can initialize with arbitrary values. We have decided to use open models as starting models. Open models are models without the restrictions imposed on the linguistically initialized model. In these models the prefix hypotheses, suffix hypotheses and basic word hypotheses may use any state, states may produce any character and all states are connected with each other.

data type	initialization	number of hypotheses			accepted positive strings	rejected negative strings
		basic word	prefix	suffix		
orthographic	random	27	376	376	595 (99.2%)	360 (60.0%)
phonetic	random	41	577	577	595 (99.2%)	408 (68.0%)
orthographic	linguistic	116	273	190	587 (97.8%)	586 (97.7%)
phonetic	linguistic	20	528	450	595 (99.2%)	567 (94.5%)

Figure 4.8: The performance of the ILP algorithm for the extended models described in this section. The models with a random initialization perform worse than the standard models when it comes to rejecting negative data (average rejection score 64% compared with 75%). The extended initialized models perform slightly better than the standard initialized models (compare with figure 4.5).

In our ILP experiments we want to derive any hypothesis that can be used for explaining a string. In a fully connected open model there will be many different possible explanations for words. All the processing paths found by the ILP algorithm will be equivalent. It does not matter if a path contains the state sequence s_1-s_2-s_3-s_4 or s_1-s_1-s_1-s_1 because all states will be linked to the same states and produce the same characters and after training and there will be a duplicate of every hypothesis for every state. For that reason an open model with one state will have the same explanatory power as an open model with more states. We want our initial model to be as simple as possible and therefore we have limited the number of states in the open initial models to one.

The 5577 orthographic training words and the 5084 phonetic words have been supplied to the ILP algorithm. From this data the algorithm derived extended hypotheses defined in section 4.1 by using as background knowledge three background hypothesis rules, three definitions and the extended hypothesis inference rule which were defined in section 4.2. One training round was sufficient in each experiment because all possible hypotheses for one word could be derived by considering the word independently of other words or other hypotheses. The results of the experiments can be found in figure 4.8.

The models which were built by starting from an open initialized model performed poorly with respect to rejecting negative strings. The extended orthographic model rejected only 360 negative test strings (60.0%). This performance is even worse than that of the standard non-initialized orthographic model (63.2%). The phonetic model performs slightly better by rejecting 408 negative strings (68.0%).[13] However this is a lot worse than our standard phonetic model which rejected 517 negative test strings (86.2%).

[13]When we remove the 26 plausible strings from the negative phonetic data (see section 4.4 of chapter 2) then the extended random model has rejected 405 of 574 strings (70.6%).

The poor performance of these models can be explained by the fact that the current learning experiments are lacking an important implicit constraint that was present in the previous experiments. In our earlier experiments we have only derived a prefix hypothesis when it could be used for explaining one existing word with another one. In these experiments we derive any prefix hypothesis that can be used for explaining a word regardless of the other words. The same is true for suffix hypotheses. When we use an open model as initial model then this derivation strategy will result in models that accept all possible substrings of the training words. For example, the word *bak* will not only lead to the correct derivation of PH(b,a,s_1), BWH(a,s_1) and SH(a,k,s_1) but also to the incorrect cluster PH(b,a,s_1), PH(a,k,s_1) and BWH(k,s_1) and the incorrect cluster BWH(b,s_1), SH(b,a,s_1) and SH(a,k,s_1). Thus each single token will be a basic word and there will be many prefix hypotheses and many suffix hypotheses. The resulting models are too weak. They accept too many strings.

The models that were built starting from the linguistically initialized model perform much better when it comes to rejecting strings of the negative test data. The orthographic model rejects 586 strings (97.7%) while the phonetic model rejects 567 strings (94.5%). The latter model performs as well as the previous initialized phonetic model (94.2%)[14] and the orthographic model achieves a higher score than before (94.0%, see figure 4.6). The current models perform slightly better when it comes to accepting positive test data. The orthographic model accepts 587 correct words (97.8%) compared with the earlier 586 words (97.7%). The phonetic model accepts 595 words (99.2%) while the earlier figure was 593 words (98.8%).

In this section we have applied the ILP algorithm to our learning problems with as a goal deriving more elaborate target models. However, the linguistically initialized models after training achieve approximately the same performance as the earlier models. Extending the models does not seem to have achieved much.

4.4 Compressing the models

The models built by ILP while starting from a linguistically initialized model perform reasonably well. However, the models are quite large. The number of rules per model varies from 472 for the standard orthographic model (figure 4.6) to 998 for the extended phonetic model (figure 4.8). We would like to decrease the size of these models because smaller models are easier to understand and easier to work with.

One of the main reasons for the models being this big is that they contain separate rules for every character. For example, the orthographic models contain seven separate rules for defining that the seven vowel characters can appear in a basic word hypothesis on their own. We would like to encode information like this in one rule.

The number of rules can be decreased by dividing the characters into character classes. The character class boundaries could have been determined by the phonetic features of the characters. We do not want to use these features because we did not

[14]Without the 26 plausible strings from the negative phonetic data the extended initialized model would have rejected 564 of 574 strings (98.3%).

48.0	a e
46.0	a e o
44.0	a e i o
42.0	a e i o u
42.0	l r
41.0	s t
38.4	c k l m n p r s t

Figure 4.9: The result of the clustering process for orthographic data: the top 25% of the token groups that frequently appear in the same context. The scores indicate how often the tokens occurred in the same context: a score of 40.0 means that on average the tokens occurred together in 40 rules with the same context. In total there were 140 rule contexts and 26 clusters.

use them in our earlier experiments with statistical learning and connectionist learning. Instead we will use data-unspecific clustering algorithms for dividing tokens into token clusters (Finch 1993). Our modified rules will specify that a character class is possible in a character class context rather than specifying that a single character is possible in some context. Working with character classes will create the possibility to cover gaps that are present in the rule set and create models that generalize better.

We have applied a clustering algorithm to the rules we have obtained in the previous section. The clustering algorithm computed the frequencies of the occurrences of tokens in similar rule contexts. For example, the orthographic prefix rule PH(?,h,s_1) states which characters can be placed before an *h* in state s_1. According to our extended orthographic model, the question mark in the rule can be replaced by one of five possible tokens: *c*, *k*, *s*, *t* and *w*. We will assume that the fact that these characters can occur in the same context means that they are somehow related to each other. The clustering process will count the number of times that tokens occur in the same context and output lists of tokens which frequently appear in similar contexts.

Example: Among the extended prefix hypotheses for the orthographic data produced by our ILP algorithm with linguistic initialization we have the two hypotheses PH(v,l,s_2) and PH(v,r,s_2). They state that we can put a *v* before an *l* and an *r* in state s_2. So the *l* and the *r* occur in the same context: behind a *v* that is added in state s_2. This means that we can put the two characters in one cluster.

We cannot use all clusters that the algorithm produces. One cluster that is not very useful is the largest possible cluster which contains all tokens. We have chosen to work only with the top 25% of the clusters (a motivation for the size of the chosen cluster group will be given later). The clustering algorithm assigned scores to the clusters which indicate how close together the elements of the clusters are. These scores have been used to define which clusters are best.

The top 25% of the clusters for the orthographic data can be found in figure 4.9. The clustering process grouped the vowels together and created three probable con-

82.0	l r
75.0	p t
75.0	n l r
73.0	k p t
70.4	y/ i: o: u: E A O I a: e: U
69.7	k p t x
69.3	i: o: u: E A O I
68.0	n l r m
64.5	k p t x n l r m
60.0	s k p t x n l r m

Figure 4.10: The result of the clustering process for phonetic data: the top 25% of the token groups that frequently appear in the same context. The scores indicate how often the tokens occurred in the same context: a score of 60.0 means that on average the tokens occurred together in 60 rules with the same context. In total there were 220 rule contexts and 40 clusters.

sonant groups. The seven character groups presented here were used to decrease the number of rules for the orthographic data. All rules which contained 75% of more of the characters of a cluster were modified. The original characters were removed from the rules and replaced with a special token which represented the complete cluster.

Example: With a cluster that contains the characters *l* and *r* we can replace the two prefix hypotheses PH(v,l,s_1) and PH(v,r,s_1) by one hypothesis: PH(v,$cluster_{lr}$,s_1). Here $cluster_{lr}$ is the name for the cluster containing *l* and *r*. This replacement will only be made if there is no larger cluster available. For example, would it have been possible to have the characters of the cluster *cklmnprst* in place of the question mark in PH(v,?,$_1$) then we would have replaced the nine prefix hypotheses with PH(v,$cluster_{cklmnprst}$,s_1). The presence of 75% cluster tokens in a rule context is enough for replacement. This means that even if only seven characters of the cluster are possible in the example context, the seven hypotheses will be replaced. By putting the threshold at 75% rather than 100% we were able to deploy the clustering algorithm to cover gaps in the rule set that resulted from the training data.

We have applied the clustering algorithm to the models with extended hypotheses produced by the ILP algorithm that started with linguistic information. The derivation of these models was discussed in section 4.3 and the performance data for the models can be found in figure 4.8. We did not apply the clustering algorithm to the non-initialized models. These models performed poorly and modifying them by dividing the tokens in groups would make performance even worse. Clustering will implicitly add rules and thus make the models accept more strings. Non-initialized models already accepted to many negative test strings and we are not interested in models that accept even more strings.

The results of the clustering process can be found in figure 4.11. The number

data type	initialization	number of hypotheses			accepted positive strings	rejected negative strings
		basic word	prefix	suffix		
orthographic	linguistic	106	103	83	590 (98.3%)	585 (97.5%)
phonetic	linguistic	10	143	131	595 (99.2%)	560 (93.3%)

Figure 4.11: The performance of the ILP algorithm for the extended models with linguistic initialization after clustering. By grouping the characters in character classes the number of rules in the orthographic model has decreased with 50% and the number of rules in the phonetic model has decreased with 70%. The performance of the models is approximately the same as the original models presented in figure 4.7.

of rules in the orthographic model decreased by approximately 50% compared with the model presented in figure 4.8 from 579 to 292 hypotheses (excluding 5 cluster definitions).[15] Compared with the original model the model accepted three extra strings of the positive test data and one more string of the negative test data. Clustering had a greater influence on the phonetic model (the applied phonetic clusters can be found in figure 4.10). The number of hypotheses decreased by more than 70% from 998 to 284 (excluding 7 cluster definitions). This model accepted seven more negative test strings than the original model.[16] The size of accepted positive test data was the same.

The clustering process contains two parameters for which we have chosen rather arbitrary values. The first parameter is the percentage of used clusters. We chose to use the best 25% of the clusters. By increasing this percentage we would obtain more clusters and generate models with fewer hypotheses. However, the chance that the set of clusters includes nonsensical clusters will increase. Models which use nonsensical clusters might accept too many negative strings.

The second parameter is the acceptance threshold. Whenever 75% or more tokens of a cluster are allowed in a certain context then we will assume that all tokens of that cluster are allowed in that context. By increasing this value we can apply the clusters at more places and thus decrease the number of hypotheses. However, the extra generalizations that will be generated as a result of this might be wrong. We can also increase the acceptance threshold but that will result in fewer application possibilities and larger models.

We would have liked to decrease the phonotactic models with another 50% to models of approximately 150 hypotheses. However, we feel that modification the

[15] We have used seven clusters but we only needed five cluster rules because some clusters contained 75% of the characters of larger clusters. These subset clusters were automatically replaced by the larger clusters. They did not need to be defined because it was impossible for them to appear in the final models.

[16] When we remove the 26 plausible strings from the negative phonetic data then this model has rejected 558 of 574 strings (97.2%).

two clustering process parameters will decrease the performances of the models that will be generated. Therefore we have tried something else. We have added two extra clusters to the models: one containing the vowels as defined in the initial linguistic model and one containing the consonants of the same model. These two clusters were already part of our original linguistic initialization.

With the two extra clusters the orthographic model decreased to 241 hypotheses excluding 6 cluster definitions (-17%). Now the model accepted 594 positive test strings (99.0%) and rejected 580 negative strings (96.7%). The size of the new phonetic model was 220 hypotheses excluding 9 cluster definitions (-23%). It accepted 596 strings of the positive data (99.3%) and rejected 560 strings of the negative data (93.3%).[17] The performance of the phonetic model is almost the same as the performance of the previous phonetic model. The new orthographic model accepts more positive test strings but it also accepts more negative strings.

In this section we have used a clustering method for decreasing the size of the rule-based models we have obtained in the previous section. The clustering method was successful: the number of rules decreased at best with 58% for the orthographic model and 78% for the phonetic model. However, the number of rules in these models, 241 orthographic rules and 220 phonetic rules, is still quite large.

5 Concluding Remarks

We have started this chapter with an introduction to rule-based learning. We have explained that neither lazy learning nor decision trees are useful for our learning problem because we want to work with positive examples only. These two learning methods require both positive and negative learning input. We have chosen the learning method Inductive Logic Programming (ILP) for our experiments. An important constraint on ILP is that it should not contain information that was not available in our previous learning experiments. A violation of this constraint would make a comparison of the ILP results with the results we have obtained in the previous chapters unfair.

We have designed an ILP process which was capable of handling our orthographic and phonetic data. We performed two variants of the learning experiments: one that started without knowledge and one that was supplied with initial knowledge which was extracted from the syllable model by Cairns and Feinstein (Cairns and Feinstein 1982). The non-initialized process produced models which performed well in recognizing correct words but they performed poorly in rejecting negative test strings (see figure 4.4). The linguistically initialized process generated models that performed well in both cases (see figure 4.6).

After these experiments we have developed rule-based models with a richer internal structure. We wanted to know whether these extended models would be able to perform better than our previous rule-based models. However, we found only a small

[17] Again a removal of the 26 plausible strings from the negative phonetic data led to a rejection rate of 97.2% for this data set.

performance increase. The best-performing extended models, the initialized extended models, achieved results that were only a little better than the ones of the previous initialized models (see figure 4.8).

We have tried to decrease the size of the extended model in an attempt to obtain models which would be easier to work with for humans. By applying a clustering algorithm we were able to decrease the model size with more than 50% with almost no performance decrease (see figure 4.11). However with rule sets containing between 200 and 250 rules, these models are still quite large.

From the results of the experiments described in this chapter we may conclude that Inductive Logic Programming (ILP) is a good learning method for building monosyllabic phonotactic models. ILP with linguistic initialization generates models that perform much better than the models that were generated without initial knowledge. The results of section 4.3 show that the number of processing stages in the models (equal to the number of states) does not seem to have a large influence on their performance. There is room for optimization in the models generated by ILP since the number of rules that they contain could be decreased with more than 50% without a performance degradation. A clear advantage of the models produced by ILP is that their internal structure can be inspected and improved if necessary.

Chapter 5

Concluding remarks

This chapter will start with a section that summarizes and compares the results of the experiments described in the earlier chapters of this thesis. After that we will describe studies performed by others that were inspired by our work. The chapter will be concluded with a section that presents the research tasks that we see as a possible follow-up on this thesis.

1 Experiment results

In this thesis we have described the application of machine learning techniques to the problem of discovering a phonotactic model for Dutch monosyllabic words. We have performed experiments with three learning algorithms, two data representation methods and two initialization schemes. The learning algorithms have only been provided with positive training data. Our goals were to find out which of the learning methods would perform best and to find out what data representation and what initialization scheme would enable the learning process to generate the most optimal model. The results of the experiments have been summarized in figure 5.1.

The first learning method we have examined was the statistical learning method Hidden Markov Model (HMM). We have used bigram HMMs which consider two characters at a time during the string evaluation process because regarding a context of one character is necessary for building a good phonotactic model. This learning method has produced good phonotactic models: after training the HMMs would accept around 99% of unseen positive test data and reject between 91 and 99% of the negative test data. There was only a small difference between training with random and initialized models but the models performed better with phonetic than with orthographic data. One observed difference was that the training process of linguistically

orthographic data	random initialization		linguistic initialization	
learning algorithm	% accepted positive data	% rejected negative data	% accepted positive data	% rejected negative data
HMM	98.9	91.0	98.9	94.5
SRN	100	8.3	100	4.8
ILP	99.2	60.0	97.8	97.7

phonetic data	random initialization		linguistic initialization	
learning algorithm	% accepted positive data	% rejected negative data	% accepted positive data	% rejected negative data
HMM	99.1	98.3	99.1	99.1
ILP	99.2	70.6	99.2	98.3

Figure 5.1: The results of our experiments with generating phonotactic models for Dutch monosyllabic words. The experiments included three learning algorithms, two initialization configurations and two data representations. No experiments have been performed with SRNs for the phonetic data representation because of the discouraging SRN results for the orthographic data representation. The HMM results for orthographic data with linguistic initialization come from the modified initialization (figure 2.19). The ILP results have been obtained with extended models (figure 4.8). The rejection scores for the negative phonetic data have been computed for the set of 574 incorrect phonetic strings.

initialized HMMs required significantly less time than that of the ones with a random initialization.

The connectionist method Simple Recurrent Network (SRN) was the second method that we have tested. This method performed surprisingly worse than the perfect results reported in (Cleeremans 1993). SRNs produced phonotactic models that accepted all unseen positive test data. However, none of the SRN models has been able to reject more than 8.3% of the negative test data. We have been able to show that the poor performance was caused by the large complexity of our training data. Characters in the data of Cleeremans et al. could be followed by at most two different characters while characters in our data can be followed by up to twenty characters. For the phonetic data this difference is even larger. Therefore we have refrained from performing SRN experiments with phonetic data.

The third learning method that we have looked at was the rule-based learning algorithm Inductive Logic Programming (ILP). This algorithm has generated good phonotactic models with linguistic initialization but the models generated with random initialization had problems with rejecting negative test strings. The large score difference for rejecting negative strings (on average 98% versus 65%) indicates that training with linguistic initialization enables this learning method to produce better

models than training without this basic knowledge. We have performed two extra experiment groups with more elaborate rule formats and with rule set compression but these have not led to large performance differences.

It is easier to determine which of the learning methods has generated the worst phonotactic models than to point at a single method that has done best. The performance of the SRN models was much worse than the models generated by the other two methods because they failed to reject many negative test strings. HMMs generated better models than ILP in training processes without linguistic initialization. However when the algorithms were equipped with basic linguistic knowledge they would generate models which performed equally well. When it comes to choosing one of the learning methods for future studies, we would recommend using ILP for three reasons. First because ILP is capable of generating good phonotactic models when equipped with initial linguistic knowledge. Second because it, unlike HMMs, generates models that consist of rules which can be inspected and understood by humans. And third because the algorithm trained faster than its closest rival HMMs.

An inspection of figure 5.1 will reveal the answer to the question which data representation format, orthographic or phonetic, has suited the learning processes best. When we compare the results of the experiments with HMMs and ILP we see that in all cases the scores for the phonetic experiments are as least as good or better than the scores for the corresponding orthographic experiments. Although this is not a proof, it is an indication that it has been easier for the learning algorithms to discover regularities in the phonetic data than in the orthographic data. This result has surprised us. The larger number of different characters in the phonetic data and the larger entropy of this data had led us to the expectation that it would be more difficult to build a good model for the phonetic data than for the orthographic data.

The answer to the question whether starting the learning process from an initial linguistic model would enable the generation of better phonotactic models can also be found by inspecting figure 5.1. In the HMM and ILP results the scores of the initialized experiments are as least as good as the noninitialized experiments in all but one case (ILP: accepted positive orthographic test data).[1] The difference is largest for the ILP rejection rate of negative test data. This is an indication that initial basic linguistic knowledge will help learning algorithms to generate better phonotactic models. In the HMM experiments initial linguistic knowledge also sped up the training process. These results have been in accordance with what we had expected.

So HMMs and ILP have generated good phonotactic models but the SRN models performed poorly We favor ILP over HMMs because ILP trains faster and generates models which are understandable for humans. The results of our experiments indicate that representing the data in phonetic format and having access to basic linguistic information ables the learning processes to generate better phonotactic models.

---

[1] This exception may have been caused by a less suitable initial orthographic model.

2 Recent related work

The publication in (Tjong Kim Sang 1995) of the research results mentioned in chapter three of this thesis has led to follow-up research by others. In this section we will discuss work by Stoianov and Nerbonne with Simple Recurrent Networks (SRNs), work by Bouma with SRNs and Synchronous-Network Acceptors and work by Klungel with genetic algorithms.

(Stoianov et al. 1998) discusses a series of experiments in which SRNs were trained and tested on building phonotactic models for orthographically represented monosyllabic and multisyllabic Dutch words. With a different data set than ours, the authors have achieved an SRN performance that is better than any of the learning methods tested in this thesis: a total error of 1.1% on accepting positive monosyllabic test data and rejecting negative data. The results were obtained in a sequence of three experiment set-ups in which each set-up improved the performance of the previous one.

In the third experiment set-up the authors changed their word evaluation routine from a function similar to the Cleeremans measure (our measure 1 from section 3.1 of chapter 3) to an evaluation routine in which the word score was equal to the product of the character scores. This decreased the error rate of the SRN from approximately 3.5% to 1.1%. In our experiments the word evaluation measure used by (Cleeremans 1993) performed worst. We have suspected that the Cleeremans measure can be improved and this new result provides more empirical support for that suspicion.

In the second experiment group the authors switched from a stand-alone SRN training process to a parallel competitive training process. In this training process the networks are tested at different time points and networks that perform poorly are replaced. This technique is borrowed from the genetics algorithms field and was suggested by Marc Lankhorst (Lankhorst 1996). It helps the training process to get out of local minima and increases the possibility of finding an SRN that performs well. A disadvantage of this approach is that it requires supplying the network with negative information during the training process. This method was explicitly excluded in our learning experiments.

Using the competitive training process enabled the SRNs to go down from a total error rate of 7.5% to 3.5% on monosyllabic data. The set-up of experiments in the first group, which reached the 7.5% error rate, comes closest to our own experiment set-up. However there are three important differences between this first group and our own experiments. The first difference is that (Stoianov et al. 1998) have implicitly dropped our constraint that all training data has to be accepted. The error rate was obtained by choosing a word acceptance threshold which accepted as many positive data as possible while rejecting as many negative data as possible. We have experimented with this approach and reached an error rate of 16.3% at best (chapter 3, figure 3.16, SRNs, measure 3, 90%). Again the consequence of this approach is that to obtain the best threshold one has to make the network evaluate negative data.

The second difference between this group of experiments and our experiments was that the training data was weighted by frequency. Frequent words occurred more

often in the training data. The number of times that a word appeared in the training data was equal to the logarithm of its frequency observed in a big text corpus. (Bouma 1997) has shown that incorporating frequency information in the training data will help SRNs to generate better phonotactic models.

The third difference was that negative data that was close to positive data was removed from the test data set. For the data evaluation the authors used the string distance function Levenshtein distance. The only strings that were allowed in the negative data set were strings that differed in two or more characters from any word in the positive data set. This restriction will simplify the task of the networks but we do not know how large the influence will be on the performance.

Stoianov and Nerbonne have provided empirical evidence for the fact that SRNs can be used as phonotactic models for Dutch. This is not something which we want to dispute. We have taken the same position as (Cleeremans 1993): we were interested in finding out whether SRNs can *learn* a good representation for the phonotactic data. The authors have shown that this question should be answered with yes. However, the question whether SRNs are able to build good phonotactic models from positive data only, remains unanswered.

(Bouma 1997) presents a study in which SRNs and Synchronous-Network Acceptors (Drossaers 1995) have been used for generating phonotactic models for Dutch monosyllabic words. In the SRN experiments he used similar techniques as Stoianov and Nerbonne in their initial group of experiments: including frequency information and dropping the constraint that all training data must be accepted. No limitations were put on the negative data but the positive data was restricted to the top 67% of a frequency ordered word list. With this approach Bouma's SRN obtained a combined error of 10.2% at best.[2] In his experiments SRNs that worked with data in which frequency information was incorporated performed better (at best an error rate of 10.2%) than SRNs that were trained with data without such information (at best 15.3%).

A Synchronous-Network Acceptor (SNA) is a biologically-plausible self-organizing neural network developed by Marc Drossaers (Drossaers 1995). It can be considered as a two-layer feed-forward network with links between the cells in the output layer. SNAs use two different variants of Hebbian learning during the training process. Bouma has performed three experiments with SNAs: one with orthographic data and two with phonetic data. This data did not contained frequency information. The experiment with orthographic data was a success: the SNA achieved a combined error rate of 2.1%. The experiments with phonetic data generated results which were worse: an error rate of 3.8% for locally encoded data and 28.1% for data encoded with phonetic features.

The results of the experiments of Bouma show that there are neural networks which can acquire good phonotactic models for Dutch monosyllabic words with positive training data only. It came as a surprise to us that in these experiments phonetic

[2]The best result was obtained with a threshold value which was determined by examining the performance of the SRNs on the test data. We feel that the test data should not be taken in consideration when determining the evaluation measure.

training data resulted in a worse performance than orthographic data. The 28.1% can probably be improved by using a different phonetic feature encoding. Bouma's results for his orthographic data set, which differs from ours, are better than the results obtained in any of our orthographic experiments.

(Klungel 1997) describes a series of experiments with genetic algorithms which generate models for the phonotactic structure of Dutch monosyllabic words. These experiments have been inspired by work on generating finite state automatons with genetic algorithms by Pierre Dupont (Dupont 1994). Klungel has used finite state models as phonotactic models which he has represented as so-called chromosomes in the genetic algorithms. In each experiment 30 models were submitted to a continuous modification process in an environment in which the best ones had the largest chance to survive. The genetic algorithm had access to positive and negative phonotactic data.

Klungel has performed experiments with two evolution methods and three fitness functions. The evolution method Individual Replacement performed best. Of the three fitness functions the lowest error rate was obtained with the one that used the average of the accepted positive data and rejected negative data as a model evaluation score (combined error rate of 8.5%). However this function did not generate models which performed consistently in the later phase of the training process.

The experiments performed by Klungel show that it possible to generate reasonable phonotactic models with genetic algorithms. We believe that even better results are possible with different genetic operators and a different initialization phase. However improving the operators and the initialization phase for this learning problem is a nontrivial task. One can imagine that they would benefit from having access to basic linguistic knowledge. A disadvantage of genetic algorithms is that it seems necessary to supply this learning method with negative data during training.

3 Future work

The work presented in this thesis and the studies discussed in the previous section have left some questions unanswered. These can be dealt with by performing follow-up research. This section will discuss possible directions for such research.

In section 2.3 of chapter 1 we have attempted to compute the complexity of our data set in order to predict the difficulty of our learning problems. We have taken a look at data entropy and the Chomsky grammar hierarchy in order to achieve that goal. Neither of the two methods was able to give an unambiguous answer to the question whether the orthographic data set or the phonetic data set was more complex. We would be interested in finding data complexity measures which are better suited for predicting the difficulty of learning problems. One of the measures that could be suitable is the Kolmogorov complexity used in (Adriaans 1992).

The work of Mark Ellison (Ellison 1992) has been discussed in section 3.1 of chapter 1. Ellison has put five constraints on his learning algorithms. Of our learning methods Inductive Logic Programming (ILP) comes closest to Ellison's goal: it satisfies four of the five constraints. The first constraint, learning algorithms should work

in isolation, is not satisfied because we have trained ILP with monosyllabic data. This can be fixed by using multisyllable words as training and test data for ILP.

The experiments in this thesis have been performed with monosyllabic words rather than multisyllabic data in order to keep down the complexity of the learning problem. Now that we have shown that machine learning techniques can generate good phonotactic models for monosyllabic data, the next logical step is to test them on multisyllabic data. Others have already shown that some machine learning techniques can generate good phonotactic models for multisyllabic data (Stoianov et al. 1998). It would be interesting to apply other learning methods to this type of data as well.

Our work with Simple Recurrent Networks (SRNs) has generated a large response. The poor SRN performance reported by us has inspired many others to suggest and test modifications of the learning algorithm and the experiment set-up. One of the modifications that was suggested was incorporating information about the frequency of the words in the training data (Stoianov et al. 1998) (Bouma 1997). The experiments performed with frequency based training data have resulted in better phonotactic models than our experiments did. These results have been obtained with SRNs. We would be interested in finding out whether Hidden Markov Models could benefit from frequency information in the training data as well.

The SRN experiments by Stoianov, Nerbonne and Bouma have used negative data for determining optimal network acceptance threshold values. We suspect that these threshold values will also reject part of the training data (see figure 3.16 in chapter 3). This raises two interesting questions. First one could ask if every SRN experiment would benefit from disregarding part of the training data after the training phase. In other words: Can we improve the performance of SRNs on the test data by determining the acceptance threshold values from the best x% ($x < 100$) of the training data after training rather than from all training data? Our own experiments suggest that this x value would be around 90%. This leads us to the second question: Would such a cut-off value be the same for all SRN experiments? Finding a universal cut-off value would enable us to determine SRN acceptance threshold values by using training data only and relieve us from having to use negative data.

We would like to see other machine learning techniques applied to our data. One group of techniques which seems useful are the memory-based lazy learning algorithms used in the work of Walter Daelemans and Antal van den Bosch. The results reported for these learning techniques applied to phonological and morphological problems have been better than for decision trees and the connectionist method backpropagation (Van den Bosch et al. 1996). However, these learning methods require both positive and negative training examples. Perhaps it is possible to construct a clever problem representation for getting around this requirement.

In studies that follow-up our work there are two issues that should be taken care of. In the first section of this chapter we have compared the results of our learning experiments. We have been unable to give exact answers to our three research questions because the lack of statistical data. Most of our experiments have resulted in single test scores. It would have been better if they had generated average test scores with standard deviations. This would have made possible comparison statements which

were supported with significance information.

Obtaining statistical data for experiment results requires repeating experiments several times. This might not always be useful. For example, performing our ILP experiments one more time with the same data would lead to the same results because our ILP training method is deterministic and uses a static initialization. Here we need an experiment set-up called 10-fold cross validation used in the work of Daelemans and others and originally suggested by (Weiss et al. 1991). In this experiment set-up the positive data is divided in ten parts and the training and test phase are performed ten times while each time a different data part is excluded from the training data and used as test data. We suggest that future experiments with our data be performed in this way to enable a statistically based comparison of the different experiments.

A second issue that should be taken care of in future experiments is consistent usage of the same data sets. In our work we have used the same training and test data for the experiments with the three learning methods. However (Stoianov et al. 1998) and (Bouma 1997) have used different data sets and this makes comparison of their results with ours difficult. In order to prevent this from happening in the future we will make our data sets universally accessible so that future experiments can be performed with the same data.[3]

Future work in applying machine learning techniques to natural language will not be restricted to generating phonotactic models for monosyllabic or multisyllabic words. One goal of our work has been to show that these techniques can be applied successfully to a small section of this domain. We hope that this thesis will provide inspiration for others to continue with new applications of machine learning techniques in larger parts of the natural language domain.

[3]Our data sets can be found on http://stp.ling.uu.se/~erikt/mlp/

Bibliography

Adriaans, Pieter Willem (1992). *Language Learning from a Categorial Perspective*. PhD thesis, University of Amsterdam, The Netherlands. ISBN 90-9005535-5.

Alphen, Paul van (1992). *HMM-based continuous-speech recognition*. PTT Research, Leidschendam, The Netherlands. PhD thesis University of Amsterdam, ISBN 90-72125-32-0.

Baayen, R.H., R. Piepenbrock and H. van Rijn (1993). *The Celex Lexical Database (CD-ROM)*. Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA.

Blank, Douglas S., Lisa Meeden and James B. Marshall (1992). 'Exploring the Symbolic/Subsymbolic Continuum: A Case Study of RAAM'. In: J. Dinsmore (ed.), *Closing the Gap: Symbolism vs. Connectionism*. Lawrence Erlbaum Associates, 1992.

Bosch, Antal van den (1997). *Learning to pronounce written words – A study in inductive language learning*. Uitgeverij Phidippedes, Cadier en Keer. PhD thesis Katholieke Universiteit Brabant. ISBN 90-801577-2-4.

Bosch, Antal van den, Walter Daelemans and Ton Weijters (1996). 'Morphological Analysis as Classification: an Inductive-Learning Approach'. In: K. Oflazer and H. Somers (eds.), *NeMLaP-2. Proceedings of the Second International Conference on New Methods in Language Processing*, pp 79–89. Ankara, Turkey, 1996.

Bosch, Antal van den, Walter Daelemans and Ton Weijters (1996b). 'Morphological Analysis as Classification: an Inductive-Learning Approach'. In: *Proceedings of NeMLaP-2, Bilkent University, Turkey*. 1996. Also available as cmp-lg/9607021.

Bosch, Antal van den, Ton Weijters, H. Jaap van den Herik and Walter Daelemans (1997). 'When small disjuncts abound, try lazy learning: A case study'. In: P. Flach, W. Daelemans and A. van den Bosch (eds.), *Proceedings of the 7th Belgian-Dutch Conference on Machine Learning, BENELEARN-97*. 1997.

Bouma, H.H.W. (1997). *Learning Dutch Phonotactics with Neural Networks*. Master thesis, Alfa-informatica department, University of Groningen.

Cairns, Charles E. and Mark H. Feinstein (1982). 'Markedness and the Theory of Syllable Structure'. In: *Linguistic Inquiry*, 13 (2), 1982.

Charniak, Eugene (1993). *Statistical Language Learning*. MIT Press.

Chomsky, Noam (1965). *Aspects of the Theory of Syntax*. MIT Press.

Cleeremans, Axel (1993). *Mechanisms of Implicit Learning*. The MIT Press.

Cleeremans, A., D. Servan-Schreiber and J.L. McClelland (1989). 'Finite State Automata and Simple Recurrent Networks'. In: *Neural Computation*, , pp 372–381, 1989.

Daelemans, Walter and Antal van den Bosch (1996). 'Language-Independent Data-Oriented Grapheme-to-Phoneme Conversion'. In: J. van Santen, R. Sproat, J. Olive and J. Hirschberg (eds.), *Progress in Speech Synthesis*. Springer Verlag, 1996.

Daelemans, Walter, Peter Berck and Steven Gillis (1995). 'Linguistics as Data Mining: Dutch Diminutives'. In: Toine Andernach, Mark Moll and Anton Nijholt (eds.), *CLIN V: Papers from the Fifth CLIN Meeting*. Neslia Paniculata Taaluitgeverij, 1995.

Daelemans, Walter, Steven Gillis, Gert Durieux and Antal van den Bosch (1993). 'Learnability and Markedness in Data-Driven Acquisition of Stress'. In: T. Mark Ellison and James M. Scobbie (eds.), *Computational Phonology*, volume 8, pp pp157–158. Edinburgh Working Papers in Cognitive Science, 1993.

Drossaers, Marc (1995). *Little Linguistic Creatures – Closed Systems of Solvable Neural Networks for Integrated Linguistic Analysis*. PhD thesis, Twente University.

Dupont, P. (1994). 'Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG method'. In: R.C. Carrasco and J. Onica (eds.), *Grammatical Inference and Applications*, Lecture Notes in Artificial Intelligence 82. Berlin Heidelberg, 1994.

Ellison, T. Mark (1992). *The Machine Learning of Phonological Structure*. PhD thesis, University of Western Australia.

Elman, Jeffrey L. (1990). 'Finding Structure in Time'. In: *Cognitive Science*, 14, 1990.

Elman, Jeffrey L. (1991). 'Incremental Learning, or The importance of starting small'. In: *Proceedings Thirteenth Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Associates, 1991.

Finch, Steven P. (1993). *Finding Structure in Language*. PhD thesis, University of Edinburgh.

Freedman, David, Robert Pisani, Roger Purves and Ani Adhikari (1991). *Statistics*. W.W. Norton & Company. ISBN 0-393-96043-9.

Fudge, Erik and Linda Shockey (1998). 'The Reading Database of Syllable Structure'. In: *Linguistic Databases*. CSLI Lecture Notes Series.

Geerts, G. and H. Heestermans (1992). *Van Dale Groot woordenboek der Nederlandse taal*. Van Dale Lexicografie.

Gilbers, D.G. (1992). *Phonological Networks*. PhD thesis, University of Groningen. ISSN 0928-0030.

Gildea, Daniel and Daniel Jurafsky (1996). 'Learning Bias and Phonological Rule Induction'. In: *Computational Linguistics*, 22 (1), pp 497–530, 1996.

Gold, E. Mark (1967). 'Language Identification in the Limit'. In: *Information and Control*, 10, pp 447–474, 1967.

Heemskerk, J. and V.J. van Heuven (1993). 'MORPA, a lexicon-based morphological parser'. In: V.J. van Heuven and L.C.W. Pols (eds.), *Analysis and synthesis of speech; strategic research toward high-quality text-to-speech generation*. Mouton de Gruyter, Berlin, 1993.

Hopcroft, J.E. and J.D. Ullman (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Huang, X.D., Y. Ariki and M.A. Jack (1990). *Hidden Markov Models for Speech Recognition*. Edinburgh University Press.

Jordan, M.I. (1986). 'Attractor Dynamics and Paralellism in a Connectionist Sequential Machine'. In: *Proceedings of the Eight Annual Conference of the Cognitive Science Society*. Erlbaum, 1986.

Klungel, Anton (1997). *Leren van Mogelijke Monosyllabische Nederlandse Woorden – een genetische benadering*. Masters thesis, Alfa-informatica department, University of Groningen. (In Dutch).

Lankhorst, Marc M. (1996). *Genetic Algorithms in Data Analysis*. PhD thesis, University of Groningen.

Miikkulainen, Risto and Michael G. Dyer (1988). 'Forming Global Representations with Extended Backpropagation'. In: *IEEE International Conference on Neural Networks*, volume I. IEEE, 1988.

Mitchell, Tom M., Richard M. Keller and Smadar T. Kedar-Cabelli (1986). 'Explanation-Based Generalization: A Unifying View'. In: *Machine Learning*, 1, pp 47–80, 1986.

Muggleton, Stephen (1992). 'Inductive Logic Programming'. In: Stephen Muggleton (ed.), *Inductive Logic Programming*, pp 3–27. 1992.

Muggleton, Stephen (1995). 'Inverse entailment and Progol'. In: *New Generation Computing Journal*, 13, pp 245–286, 1995. Also available as ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Papers/InvEnt.ps.

Partee, B.H., A. ter Meulen and R.E. Wall (1993). *Mathematical Methods in Linguistics*. Kluwer.

Quinlan, J. Ross (1993). *C4.5 : programs for machine learning*. Morgan Kaufmann. ISBN 1-55860-238-0.

Rabiner, L.R. and B.H. Juang (1986). 'An Intrdiction to Hidden Markov Models'. In: *IEEE Acoustics, Speech and Signal Processing (ASSP) Magazine*, 3, pp 4–16, 1986.

Reber, A.S. (1976). 'Implicit learning of synthetic languages: The role of the instructional set.'. In: *Journal of Experimental Psychology: Human Learning and Memory*, 2, 1976.

Rich, Elaine and Kevin Knight (1991). *Artificial Intelligence*. McGraw-Hill.

Rosenfeld, Ronald (1996). 'A Maximum Entropy Approach to Adaptive Statistical Language Modeling'. In: *Computer, Speech and Language*, 10, pp 187–228, 1996. Also available at http://www.cs.cmu.edu/r̃oni/me-csl-revised.ps.

Rumelhart, D.E., G.E. Hinton and R.J. Williams (1986). 'Learning Internal Representations by Error Propagation'. In: *David E. Rumelhart and James A. McClelland*, volume 1. The MIT Press, 1986.

Servan-Schreiber, D., A. Cleeremans and J.L. McClelland (1991). 'Graded state machines: The representation of temporal continguincies in SImple Recurrent Networks'. In: *Machine Learning*, , pp 161–193, 1991.

Spellingscommissie, Nederlands-Belgische (1954). *Woordenlijst der Nederlandse Taal*. Staatsdrukkerij. (Available by anonymous ftp from ftp://donau.et.tudelft.nl/pub/words/platte_lijst.Z).

Stoianov, Ivelin and John Nerbonne (1998). *Connectionist Learning of Natural Language Lexical Phonotactics*. manuscript.

Tjong Kim Sang, Erik F. (1995). 'The Limitations of Modeling Finite State Grammars with Simple Recurrent Networks'. In: *CLIN V, Papers from the Fifth CLIN Meeting*. Taaluitgeverij, Enschede, The Netherlands, 1995.

Todd, Peter (1989). 'A Connectionist Approach to Algorithmic Composition'. In: *Computer Music Journal*, 13 (4), 1989.

Trommelen, M. (1983). *The Syllable in Dutch, with special reference to diminutive formation*. Foris Publications.

Wasserman, Philip D. (1989). *Neural Computing: theory and practice*. Van Nostrand Reinhold. ISBN 0-442-20743-3.

Weijters, Ton and Geer Hoppenbrouwers (1990). 'Netspraak: Een neuraal netwerk voor grafeem-fonomeen-omzetting (Netspraak: A neural network for converting text into a phonological representation)'. In: *TABU*, 20 (1), 1990. (dutch).

Weiss, S. and C. Kulikowski (1991). *Computer Systems that Learn*. Morgan Kaufmann.

Wexler, Kenneth and Peter W. Culicover (1980). *Formal Principles of Language Acquisition*. MIT Press.

Winston, Patrik Henry (1992). *Artificial Intelligence*. Addison-Wesley.

Zeidenberg, Matthew (1990). *Neural Network Models in Artificial Intelligence*. Horwood, Chichester. ISBN 0-13-612185-3.

Zonneveld, R.M. van (1988). 'Two Level Phonology: Structural Stability and Segmental Variation in Dutch Child Language'. In: F. van Besien (ed.), *First Language Acquisition*. ABLA papers no. 12, University of Antwerpen.

Samenvatting

Dit proefschrift bespreekt de toepassing van leertechnieken bij de computationele verwerking van natuurlijke taal. We hebben drie verschillende leermethoden gebruikt voor het genereren van fonotactische modellen. Deze modellen hebben als taak het beoordelen van reeksen letters. Zij moeten kunnen beslissen of een bepaald woord wel of niet mogelijk is in een taal. Een goed fonotactisch model voor het Nederlands zal bijvoorbeeld 'kraag' goedkeuren en 'grmbl' afkeuren.

In ons onderzoek hebben we geprobeerd antwoorden te vinden op de volgende drie vragen:

1. Welke leermethode genereert de beste fonotactische modellen?
2. Heeft de representatiemethode van de leergegevens invloed op de kwaliteit van de geproduceerde fonotactische modellen?
3. Worden de fonotactische modellen beter als de leermethode taalkundige basiskennis beschikbaar heeft?

We hebben tien groepen leerexperimenten uitgevoerd. In elk van de experimentgroepen kregen de leermethoden ongeveer vijfduizend eenlettergrepige Nederlandse woorden aangeboden. Op basis van deze leerdata moesten zij een fonotactisch model voor eenlettergrepige Nederlandse woorden bouwen. Deze modellen zijn daarna getest met zeshonderd nieuwe eenlettergrepige Nederlandse woorden en zeshonderd letterreeksen die geen Nederlands woord vormen.

De eerste door ons geteste methode heet Hidden Markov Model (HMM). HMMs gebruiken een statistische leertechniek. De door HMMs geproduceerde fonotactische modellen presteerden goed. De beste HMMs accepteerden 99% van de correcte testdata en keurden 99% van de incorrecte testdata af. HMMs die startten met taalkundige basiskennis bouwden sneller een goed fonotactisch model dan HMMs die geen taalkundige basiskennis kregen aangeboden. De modellen van beide groepen HMMs presteerden ongeveer even goed.

Als tweede hebben we een neuraal netwerk met de naam Simple Recurrent Network (SRN) onderzocht. Dit netwerk presteerde erg slecht: het accepteerde alle correcte testdata maar keurde nooit meer dan 5% van de incorrecte testdata af. Dit resultaat heeft ons verbaasd omdat in vergelijkbare experimenten beschreven in de SRN-

literatuur alle correcte testdata wordt goedgekeurd en alle incorrecte testdata wordt afgekeurd. We konden aantonen dat het prestatieverschil ligt aan de complexiteit van de leerdata. Onze leergegevens uit het Nederlands zijn veel complexer dan de data die is gebruikt in de SRN-literatuur.

De derde leermethode die we hebben toegepast is een regelgebaseerde methode genaamd Inductive Logic Programming (ILP). Deze leertechniek produceerde ook goede fonotactische modellen. Het beste model gegenereerd door ILP accepteerde 99% van de correcte testdata en keurde 98% van de incorrecte testdata af. Er was een opvallende prestatieverbetering merkbaar in de ILP-experimenten waar taalkundige basiskennis beschikbaar was. Het afkeurpercentage voor incorrecte data lag voor modellen die waren gegenereerd zonder het gebruik van deze kennis tussen 60 en 70%. ILP met taalkundige basiskennis produceerde modellen die gemiddeld 98% van de incorrecte testdata afkeurden.

Na het uitvoeren van onze experimenten konden we onze onderzoeksvragen beantwoorden. Wat betreft de prestatie van de leermethoden: HMMs en ILP genereren fonotactische modellen die veel beter zijn dan de modellen geproduceerd door SRNs. Hoewel de ILP-modellen iets slechter presteren dan de HMM-modellen adviseren we het gebruik van ILP voor vervolgonderzoek. ILP heeft namelijk minder rekentijd nodig en de gegenereerde modellen bestaan, in tegenstelling tot HMM-modellen, uit regels die mensen kunnen interpreteren en manipuleren.

Voor het beantwoorden van onze vraag over datarepresentatie hebben we twee verschillende representatiemethoden vergeleken: de orthografische methode en de fonologische methode. In de eerste methode zijn woorden gerepresenteerd als reeksen van letters. In de tweede representatiemethode zijn woorden gecodeerd als reeksen van fonologische symbolen. Zowel HMM-modellen als ILP-modellen presteerden beter voor fonologische data dan voor orthografische data. SRNs zijn alleen maar toegepast op orthografische data.

De experimenten met taalkundige basiskennis leverden modellen op die op bijna alle punten beter presteerden dan de modellen die waren gebouwd zonder deze kennis te gebruiken. Het verschil was het grootst voor de ILP-modellen in de scores voor het afkeuren van incorrecte testdata. Het beschikbaar stellen van taalkundige basiskennis helpt leeralgorithmen dus bij het produceren van betere datamodellen.

We concluderen dat HMMs en ILP goede fonotactische modellen kunnen bouwen voor eenlettergrepige woorden uit een natuurlijke taal zonder dat incorrecte data hoeft te worden aangeboden tijdens het leerproces. SRNs genereren slechtere modellen omdat zij moeite hebben met de complexiteit van de data. De beste fonotactische modellen kunnen worden verkregen door de data te representeren als reeksen van fonologische symbolen en door tijdens de leerfase taalkundige basiskennis ter beschikking te stellen.