AN ABSTRACT OF THE THESIS OF

Ghulum Bakiri for the degree of Doctor of Philosophy in Computer Science presented on January 9, 1991.

Title: Converting English Text to Speech: A Machine Learning Approach

Signature redacted for privacy.

Abstract approved:____

Thomas G. Dietterich

The task of mapping spelled English words into strings of phonemes and stresses ("reading aloud") has many practical applications. Several commercial systems perform this task by applying a knowledge base of expert-supplied letter-to-sound rules. This dissertation presents a set of machine learning methods for automatically constructing letter-to-sound rules by analyzing a dictionary of words and their pronunciations. Taken together, these methods provide a substantial performance improvement over the best commercial system—DECtalk from Digital Equipment Corporation. In a performance test, the learning methods were trained on a dictionary of 19,002 words. Then, human subjects were asked to compare the performance of the resulting letter-to-sound rules against the dictionary for an additional 1,000 words not used during training. In a blind procedure, the subjects rated the pronunciations of both the learned rules and the DECtalk rules according to whether they were noticeably different from the dictionary pronunciation. The error rate for the learned rules was 28.8% (288 words noticeably different), while the error rate for the DECtalk rules was 44.3% (443 words noticeably different). If, instead of using human judges, we required that the pronunciations of the letter-to-sound rules exactly match the dictionary to be counted correct, then the error rate for our learned rules is 35.2% and the error rate for DECtalk is 63.6%. Similar results were observed at the level of individual letters, phonemes, and stresses.

To achieve these results, several techniques were combined. The key learning technique represents the output classes by the codewords of an error-correcting code. Boolean concept learning methods, such as the standard ID3 decision-tree algorithm, can be applied to learn the individual bits of these codewords. This converts the multiclass learning problem into a number of boolean concept learning problems. This method is shown to be superior to several other methods: multiclass ID3, one-tree-per-class ID3, the domain-specific distributed code employed by T. Sejnowski and C. Rosenberg in their NETtalk system, and a method developed by D. Wolpert. Similar results in the domain of isolated-letter speech recognition with the backpropagation algorithm show that error-correcting output codes provide a domain-independent, algorithm-independent approach to multiclass learning problems.

Converting English Text to Speech: A Machine Learning Approach

by

Ghulum Bakiri

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed January 9, 1991

Commencement June, 1991

To my wife: Haya

ACKNOWLEDGEMENTS

I am indebted to my advisor, Tom Dietterich, who was a constant source of ideas and encouragement throughout the research period culminating in writing up this dissertation. I would also like to thank Bella Bose, Hussein Almuallim and especially Tom Dietterich for their insightful comments on several earlier drafts of this thesis.

Thanks are also due to the many people who contributed in one way or another to the successful completion of this research: Terry Sejnowski for providing the NETtalk phonemic dictionary, without which this work would have been impossible; Tony Vitali of Digital Equipment Corporation (DEC), for running the entire NETtalk dictionary through DECtalk and providing us with the outputs from DECtalk's rule-base on all the words in the dictionary; Sulaiman Al-Bassam for providing us with a routine for generating BCH error-correcting codes and for numerous discussions about error-correcting codes in general; David Wolpert for making available the best set of weights that he arrived at for his method on the NETtalk domain; Bahrain University for supporting my graduate studies through a four year scholarship (albeit with strings attached); and the National Science Foundation (NSF) and SUN Microsystems for providing the computing facilities that made this research feasible (NSF grant numbers CCR-87-16748 and IRI-86-57316 with matching support from SUN Microsystems).

Finally, I would like to thank my wife Haya, to whom this thesis is dedicated. Without her patience, understanding and personal sacrifice, none of this would have been possible.

Table of Contents

List of Figures

List of Tables

Table Page

Table | Page

Converting English Text to Speech:
A Machine Learning Approach

Chapter 1

Introduction

This chapter gives an overview of the setting in which this research was conducted and the motivation behind pursuing this line of research. It is intended to aid the reader in developing a frame of mind suitable for reading the remaining chapters.

1.1 Text to Speech: An Overview

The automatic conversion of English text to synthetic speech is an important task that has a wide range of practical applications [Klatt87]. In spite of their curren-t limitations, devices capable of performing this task are beginning to find their way to commercial applications ranging from telephone access to information and computerized data bases (such as weather, ski conditions, yellow pages, airline schedules, etc.) to talking warning and alarm systems (e.g. in an airplane cock-pit), systems that can not only alert the pilot in case of any malfunction, but that can potentially guide him/her through a sequence of diagnostic steps as suggested by the on-board computerized diagnostic expert system. Besides these commercial applications, such text to speech devices also have a number of unique humanitarian applications such as their use as reading aids for the blind and/or talking aids for the vocally handicapped. The interested reader is referred to Section V of [Klatt87] for a more detailed discussion of these and other applications.

The overall process of converting English text into speech is quite difficult and involved. Figure 1 illustrates the various steps involved in the conversion process as performed by DECtalk, a typical system commercially available for performing this task. One particularly difficult step involves mapping words (i.e strings of letters) into strings of phonemes and stresses. This mapping is typically performed by a large[1] rule based system that is hand-crafted to handle the "regular" conversions, while an exception dictionary is relied upon to take care of those irregular words that defy the rule base. DECtalk, for example, provides a built-in dictionary containing some 6,000[2] common words, abreviations and exceptions. To map any word to its corresponding string of phonemes and stresses, the dictionary is searched first. If a match is found, the pronunciation is taken from the dictionary entry. Otherwise, the rule base is invoked to perform the required mapping. The overall performance of the system for any *particular* application can be arbitrarily increased over the performance of the rule base alone by incorporating more and more of the words commonly encountered in *that* application as part of the exception dictionary. Nevertheless, it is essential to have an accurate (competent) rule base in order to limit the exception dictionary to a reasonable size.

This thesis explores the feasibility of utilizing machine learning techniques to automatically build a "rule base" suitable for performing the mapping of isolated English words onto strings of phonemes and stresses. In order to simplify the exposition in the rest of the thesis, we will refer to this mapping task as *text to speech* mapping, even though it is strictly only one step—albeit an important one—in the overall process of text to speech conversion. We will be taking a

[1]500 to 1000 rules are typical for this application.

[2]Expandable by the user.

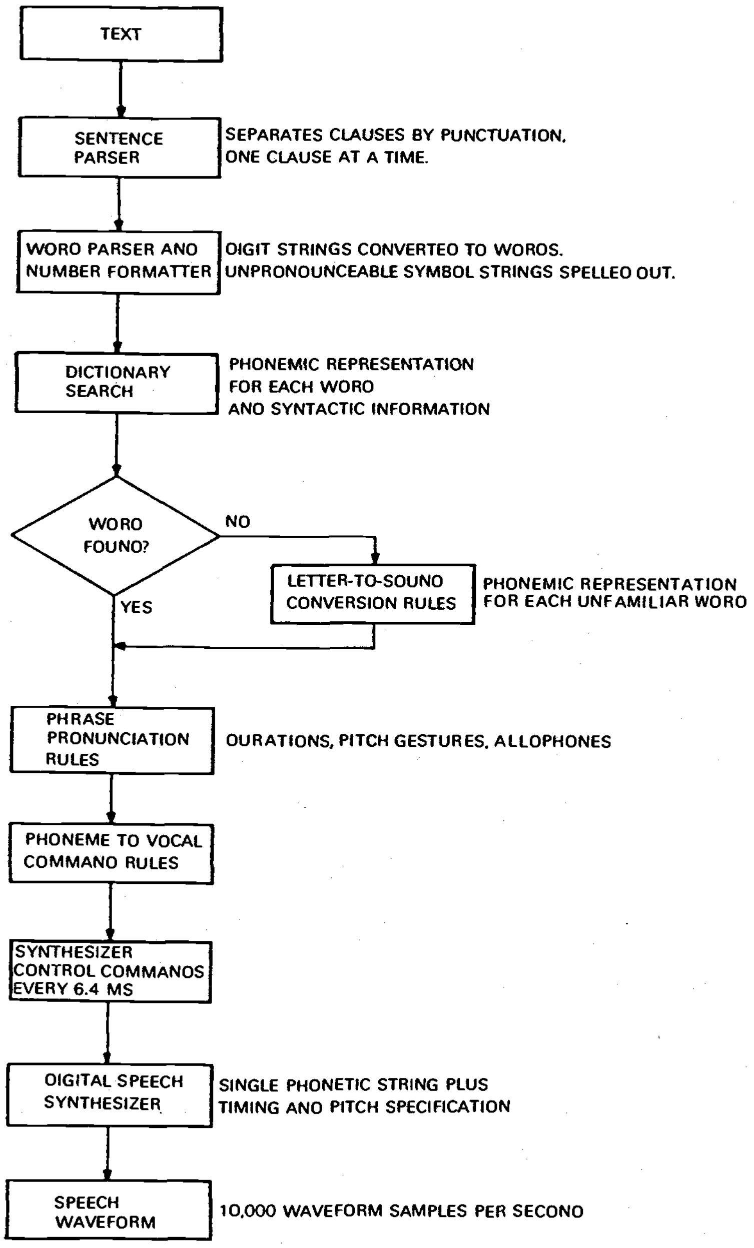
TEXT
SENTENCE PARSER
SEPARATES CLAUSES BY PUNCTUATION, ONE CLAUSE AT A TIME.
WORO PARSER ANO NUMBER FORMATTER
OIGIT STRINGS CONVERTEO TO WOROS. UNPRONOUNCEABLE SYMBOL STRINGS SPELLEO OUT.
DICTIONARY SEARCH
PHONEMIC REPRESENTATION FOR EACH WORO ANO SYNTACTIC INFORMATION
WORO FOUNO?
NO
YES
LETTER-TO-SOUNO CONVERSION RULES
PHONEMIC REPRESENTATION FOR EACH UNFAMILIAR WORO
PHRASE PRONUNCIATION RULES
OURATIONS, PITCH GESTURES, ALLOPHONES
PHONEME TO VOCAL COMMANO RULES
SYNTHESIZER CONTROL COMMANOS EVERY 6.4 MS
OIGITAL SPEECH SYNTHESIZER
SINGLE PHONETIC STRING PLUS TIMING ANO PITCH SPECIFICATION
SPEECH WAVEFORM
10,000 WAVEFORM SAMPLES PER SECONO

Figure 1. How DECtalk converts English text into speech (Reproduced from DECtalk DTC01 owner's manual, Digital Equipment Corporation).

machine learning approach, and will cast the mapping problem in the framework of inductive learning, or more precisely: *Learning multiple valued functions from examples.*

1.2 Objectives

The objectives of this research were two fold:

Domain objective: The automatic generation of a high performance rule base that can compete with current expert systems for converting isolated English words to strings of phonemes and stresses.

Main objective: Even though the domain objective is an important goal in its own right, our primary motivation for pursuing this line of research was to use this domain as a test bed for developing general machine learning techniques for the task of learning multiple valued functions from examples. Our goal was to develop efficient algorithms and/or techniques that outperform existing methods for this task.

In the next section we will present a formal definition of the learning task. Section 1.4 will then outline the typical procedure followed in order to cast the problem of text to speech mapping in the framework of learning from examples.

1.3 Learning from Examples

In recent years, several advances have been made in machine learning in the area of learning from examples by inductive inference.

The technology of building decision trees from examples is fairly robust for the case of boolean functions expressed in terms of boolean features [Quinlan86]. Several enhancements to Quinlan's basic algorithm for building decision trees have been proposed—enhancements that allow it to deal with noisy data, missing features, as well as procedures for extracting and refining production rules from decision trees. Methods for learning boolean concepts expressed in other representations (e.g. decision lists, [Rivest87]) have also been developed.

On the theoretical side, a formal model for the task of inductive learning from examples has emerged [Valiant84]. This model gained wide-spread acceptance in the machine learning community, since it was the first model that provided a much needed insight into the complexity of the learning task. Since its introduction, the Valiant model spurred a flurry of activity in this field. Researchers like [Haussler88], [Blum87a] and [Blum87b] extended the model, proved upper and lower bounds on the number of examples needed for learning and introduced what is known as the class of Occam algorithms; a formal justification for Occam's razor. Other researchers working with the Valiant model have published results on the learnability of various classes of concepts. On another front, progress is being made toward the characterization of "optimal coverage" inductive learning, and theoretical bounds are being computed on the number of concepts learnable by a hypothetical optimal coverage algorithm [Dietterich89a, Almuallim90].

Despite the progress made in the above areas, little work has been done to tackle several challenging inductive learning tasks that arise in real life. The main reason for this is that most of the earlier work in inductive learning has been restricted to the task of learning boolean functions: where the values the functions can take are restricted to be either 1 or 0. The general problem of learning discrete multi-valued functions is to a large extent left untackled and, until recently, little experimental work was done to extend the well known boolean learning algorithms to cover the more general case of learning multi-valued functions.

1.3.1 Problem Definition

We are interested in the general problem of learning multiple-valued functions from examples. Our goal is to develop efficient algorithms that outperform all existing methods for this task.

Let us introduce a formal definition of the learning task.

Assume that there are n features, $\{a_1 a_2 \dots a_n\}$, characterizing a given domain, and let D_i be the set of allowable values for feature $i, i = 1, \dots, n$. We then define the event space X as:

$$X = D_1 \times D_2 \times \dots \times D_n.$$

Let there be a mapping function $g : X \longrightarrow V = \{v_1, \dots, v_C\}$. Hence, V is the set consisting of C elements comprising the output values that the function g is allowed to take.

Our task can be stated as learning a mapping function f which is an approximation of g from a limited number of examples, each of the form: $(\vec{x}, v)$ where

$\vec{x}$ is an attribute (feature) vector:$\langle x_1 x_2 \dots x_n \rangle \in X$ and,

$v \in V$.

Our learning algorithm will take as input a set of examples and produce as output a mapping function f in some representation language (e.g. a decision tree).

We will normally restrict the features to be boolean[3] , i.e. each $x_i \in \{0,1\}(\vec{x} \in \{0,1\}^n)$. Following the above notation, each D_i is the set {0,1} and X is $\{0,1\}^n$. Hence, the task can be re-stated as learning the mapping function:

$$f : \{0,1\}^n \longrightarrow V.$$

The set V of allowable values for the function f is often called the set of output classes and each $v \in V$ is the name of an output class. Hence, this learning task can also be referred to as the multiclass learning problem.

It is often convenient to give each output value (class name v_i) a unique number $\{1, \dots, C\}$ that we will refer to as the class number. Since there is a one-to-one correspondence between $\{1, \dots, C\}$ and $\{v_1, \dots, v_C\}$, we can learn the equivalent mapping function

$$f : \{0,1\}^n \longrightarrow \{1, \dots, C\}.$$

[3]This restriction does not impose any serious limitations in practice as will become evident in the next sections. Reasons for preferring binary attributes are discussed at length in [Lucassen83].

Hence, C is also referred to as the number of classes in the domain.

In this work, we will allow the number of classes to be arbitrarily large. (Note that $C = 2$ corresponds to the special case of boolean concept learning.)

1.3.2 General Approaches to the Multiclass Learning Problem

A few techniques have already been developed for multiclass learning problems. Each of these techniques involves finding a way to apply existing boolean concept learning algorithms, such as ID3 [Quinlan86], the perceptron algorithm [Rosenblatt58], and Backpropagation [Rumelhart86], to multiclass problems. We will describe each approach briefly. More details will be given in later sections.

The Direct Multiclass Approach

A straightforward method that can be applied to some symbolic learning algorithms such as ID3, FRINGE [Pagallo88], and GROVE [Pagallo88], is to generalize these to algorithms that handle multiclass learning directly without the need for converting the problem first to boolean classes. The algorithms can be easily modified to store (and output upon evaluation) an integer class number instead of a boolean class. We will refer to this kind of algorithm as a Multiclass Algorithm.

The One-per-class Approach

Another approach for dealing with multiclass learning is to learn one boolean function f_i for every output class $i, i \in \{1, \ldots, C\}$, where C is the number of classes in the domain. Each function f_i should decide whether a particular example is a member of that class or not. The functions can be learned by the application of well established binary concept learning algorithms such as ID3. Problems arise in this approach when several functions (or none) classify a novel instance as a member of the class they represent. Some means of resolving the class ambiguity that results in such cases will be needed.

The Distributed Coding Approach

We can reformulate the problem definition given in Section 1.3.1 by choosing a suitable boolean encoding for the classes. Let m be the codeword length so that each class is represented by a unique codeword: $\langle u_1 u_2 \ldots u_m \rangle$ where each $u_j \in \{0,1\}, j = 1, \ldots, m$. The code is sparse if $C \ll 2^m$, i.e. the number of classes is much smaller than the space of codewords. The learning task can then be stated as learning the mapping function:

$$f : \{0,1\}^n \longrightarrow \{0,1\}^m$$

Since the individual bits of the class codes can be learned separately, the problem is now reduced to learning the collection of m boolean functions:

$$f_1 : \{0,1\}^n \longrightarrow \{0,1\} \mid f_1(\vec{x}) = u_1$$

$$f_2 : \{0,1\}^n \longrightarrow \{0,1\} \mid f_2(\vec{x}) = u_2$$

$$f_m : \{0,1\}^n \longrightarrow \{0,1\} \mid f_m(\vec{x}) = u_m$$

However, this method introduces another "decoding phase" after evaluation, since the bit vector $\langle u_1 u_2 \ldots u_m \rangle$ evaluated from f_1 to f_m must be mapped to a single class u. This leads to two new problems that must be addressed:

The aggregation problem: Since the correctness criterion is that all the bits of u be correct, a severe requirement is imposed on the accuracy of the learning of each of the individual functions f_1 to f_m because a small probability of error of a few percent in each of the individual boolean functions can result in a significant overall probability of error in the final class. Suppose $m = 25$ and each f_i is 99% correct. The overall rate is $(0.99)^{25} = 0.78$, only 78% correct.

The need for a decoding strategy: The bit vector $\langle u_1 u_2 \ldots u_m \rangle$ may not correspond to any legal codeword for any of the classes $\{1, \ldots, C\}$. A strategy for mapping such a bit vector to the class that best matches may lead to some improvement in the performance of the learning method.

1.4 Domain

As mentioned in the previous sections, we intend to test our ideas in the domain of text to speech conversion. The learning task will be to discover mappings of isolated English words from a dictionary to a string of phonemes and stresses that can be pronounced by a hardware device, such as the DECtalk machine. This domain is challenging and serves as a representative real life domain of a multiclass learning problem in which C, the number of classes, is very large.[4]

To achieve perfect performance in this task, one has to understand the structure of the English text in order to determine the correct pronunciation of a word. In the sentence "I have read that many people can't read.", the pronunciation of the word "read" depends on its grammatical context. However, this is an extreme case. The pronunciation for most other words can be determined by looking at the spelling independently of context. In this section we will introduce the domain. A more detailed description can be found in [Hild89].

A dictionary of 20,002 English words and their pronunciations was made available to us by T. Sejnowski [Sejnowski87]. Following [Sejnowski87], we will refer to this dictionary as the "NETtalk dictionary" and to the domain as the "NETtalk domain".

Each entry in the dictionary consists of four fields:

1. A character string representing the English word.

2. A string of "phonemes".

[4]For the general case, C will be the number of English words (with distinct pronunciations) in the dictionary.

3. A string of "stresses".

4. An integer in {0,1,2} indicating whether the word is regular (0), irregular (1) or foreign (2).

Example:

```
abandon    xb@ndxn      0>1<>0<      0
```

In our functional notation, this can be written as:

$$f(\text{“abandon”}) = (\text{“xb@ndxn”} \ , \text{“}\texttt{0>1<>0<}\text{”}).$$

For the purposes of the rest of the subsections, Let

$\mathcal{A}$	be the set containing the 26 letters and the three symbols "–", "_", and "." .
$\mathcal{P}$	be the set of 54 phonemes as defined in [Sejnowski87].
$\mathcal{S}$	be the set of 6 stress symbols: {0,1,2,>,<,-}.

1.4.1 Representation of Pronunciation Rules

If we represented pronunciation rules at the full word level, i.e. in the form:

$$f(\text{“abandon”}) = (\text{“xb@ndxn”} \ , \text{“}\texttt{0>1<>0<}\text{”})$$

they would be uninteresting, since there would be as many rules as there are words in the dictionary. Furthermore, f defined in such a manner is a very complex discrete mapping with a very large range. If we assume no word contains more than 28 letters (the length of "antidisestablishmentarianism"), this range would contain more than 10^{70} elements.

One approach to representing pronunciation rules is to write them in terms of individual letters. For example, we could try to learn rules of the form:

$$f(\text{“a”}) = (\text{“x”} \ , \text{“0”}).$$

Such a rule would correctly pronounce the first letter in "abandon". Unfortunately, individual letters can be pronounced many different ways in English, depending on context. Hence, the approach that we will take in most of our experiments is to learn rules that map from a letter and some context around it to the pronunciation for that letter. The context will be typically depicted by 3 letters on each side of the letter, hence, the function we want to learn is the mapping from a 7-letter window to a phoneme/stress pair:

$$f : \mathcal{A}^7 \longrightarrow \mathcal{P} \times \mathcal{S}.$$

For example, mapping the first letter in "abandon" will be represented as:

$$f(\text{“}\texttt{---aban}\text{”}) = (\text{“x”}\ ,\ \text{“0”}).$$

This representation was introduced by [Sejnowski87]. Sejnowski also experimented with windows of various sizes, and found that a seven letter window is adequate for capturing most of the context, even though it is not large enough in some cases.[5] A more detailed analysis by [Hild89] found a 7-letter window to be adequate for uniquely determining the mappings of 98.5% of all letters in the various contexts they occur in the 20,002 word NETtalk dictionary. We will be using a 7-letter window for most of our experiments. Larger window sizes will be considered in Chapter 6 where the best performance of each learning algorithm is sought for comparisons with DECtalk.

1.4.2 Decomposing the Dictionary Entries into Learning Examples

Each entry in the dictionary corresponding to a k-letter word is converted to k examples; one for each letter of the word. For each letter L in the word, a 7-letter window is created consisting of L at the center, the 3-letters preceding L and the

[5]Distinguishing between the pronunciations of the first letter in "thought" and "though", for example, will require more than 3 letters on each side.

three letters following L. The character "-" (symbol for space) is used if there are not enough neighbors on either side of the letter L to complete the number of letters on each side to be 3. The 7-letter windows play the part of the feature-vector (after converting them to a binary encoding). The output is the pair of phoneme and stress symbols that L maps to in this particular word. Throughout the thesis, we will refer to L—the letter at the center of the window—as the *current letter*. As an example, the following entry in the dictionary

("aback" "xb@k-" "0>1<<")

will generate the following examples :

```
("---abac"  x  0)
("--aback"  b  >)
("-aback-"  @  1)
("aback--"  k  <)
("back---"  -  <)
```

1.4.3 Converting the Letters to Binary Representation

The feature vector (the 7-letter window) of a learning example is converted into a binary feature vector by transforming each letter into a binary representation. The encoding that we will adopt in most of our experiments will be that employed by [Sejnowski87], since it was also found to be a good representation by [Hild89] and [Shavlik89]. In this representation, every letter of the 7-letter window is encoded as a 29-bit vector. The i^{th} letter in the alphabet $\mathcal{A}$ is represented by a bit-vector $\langle b_1 \ldots b_i \ldots b_{29} \rangle$ in which all but the i^{th} bit are set to 0. By concatenating the seven 29-bit-vectors we obtain a 203-bit-vector that represents the entire 7-letter window. Hence, the mapping we want to learn can be stated as:

$$f : \{0,1\}^{203} \longrightarrow \mathcal{P} \times \mathcal{S}.$$

Chapter 4 explores several alternatives to this standard input representration.

1.4.4 Representing the Classes

Potentially, there are $54 \times 6 = 324$ phoneme/stress pairs. However, only 163 such pairs appear as valid pronunciations in the NETtalk dictionary.[6] We can consider each of these 163 pairs of phonemes and stress symbols as a class, and learn one function to do the mapping:

$$f : \{0,1\}^{203} \longrightarrow \{1,\ldots,163\}$$

Alternatively, we can learn two separate functions. One to map the 7-letter window to one of the 54 phonemes and another to map the 7-letter window to one of the 6 stress symbols:

$$f_p : \{0,1\}^{203} \longrightarrow \{1,\ldots,54\}$$

$$f_s : \{0,1\}^{203} \longrightarrow \{1,\ldots,6\}$$

We will refer to the former case as the combined-ps (combined phoneme/stress approach) and to the latter as the separate-ps approach.

1.4.5 Binary Output Codes

As outlined in Section 1.3.2, we can reformulate the multiclass problem by assigning to each class a unique binary string (or simply a binary code) of length m. We can then apply Boolean concept learning algorithms to learn m binary functions: one for each bit position in these binary codes. Several such encoding schemes are possible. For example, selecting a local encoding of the C classes (i.e. a C-bit weight-1 code for each class i, in which all but the i^th bit are 0) will be equivalent to the one-per-class approach since learning the individual bits of this code will be the same as learning an individual membership function for each class.

Another possible scheme is to employ the binary code developed by Sejnowski and Rosenberg [Sejnowski87]. They code each possible phoneme/stress pair as a

[6]Out of these, only 126 pairs appear in our "standard" 1000-word training set.

26-bit string, 21 bits for the phoneme and 5 bits for the stress. The bits are "meaningful" in the sense that each bit corresponds to some property of the phoneme or stress. Each bit in the 21-bit code employed for the phoneme encodes some articulatory feature of the phoneme. These describe physiological properties of the phoneme (or the way it is articulated) such as "vowel height" or "position in the mouth" (see Appendix B). The 5-bit code employed for the stress encodes word and syllable boundary and whether the first vowel in a nucleus of a syllable is receiving primary, secondary or no stress. The code is sparse since $2^{26} \gg 163$, the total number of phoneme/stress combinations appearing in the entire dictionary. We will refer to this output code as the "standard distributed code" or as the "Sejnowski & Rosenberg distributed code".

During learning, 26 separate Boolean functions, $f_1, \ldots, f_{26}$, will be learned: one for each of the 26 bits of the code. Each function f_i maps from a seven-letter window to $\{0, 1\}$. During evaluation, we assign a phoneme and stress to a window by a 2-step process. First, all 26 functions are evaluated to produce a 26-bit string. This string is then mapped to the nearest of the 126 bit strings representing observed phoneme/stress pairs. We use the Hamming distance between two strings to measure distance. Ties are broken in favor of the phoneme/stress pair that appeared more frequently in the training data. In [Dietterich90a,b], we called this "observed decoding." Several other decoding strategies for mapping the 26-bit output string to a legal phoneme/stress pair are possible. The impact of these strategies will be discussed in Section 2.4.

Figure 2 gives an overview of the NETtalk data and their decomposition into learning examples.

1.5 Base Configuration

In this section we define a standard configuration that will be used as a benchmark for guaging the effectiveness of all subsequent modifications. This will keep things

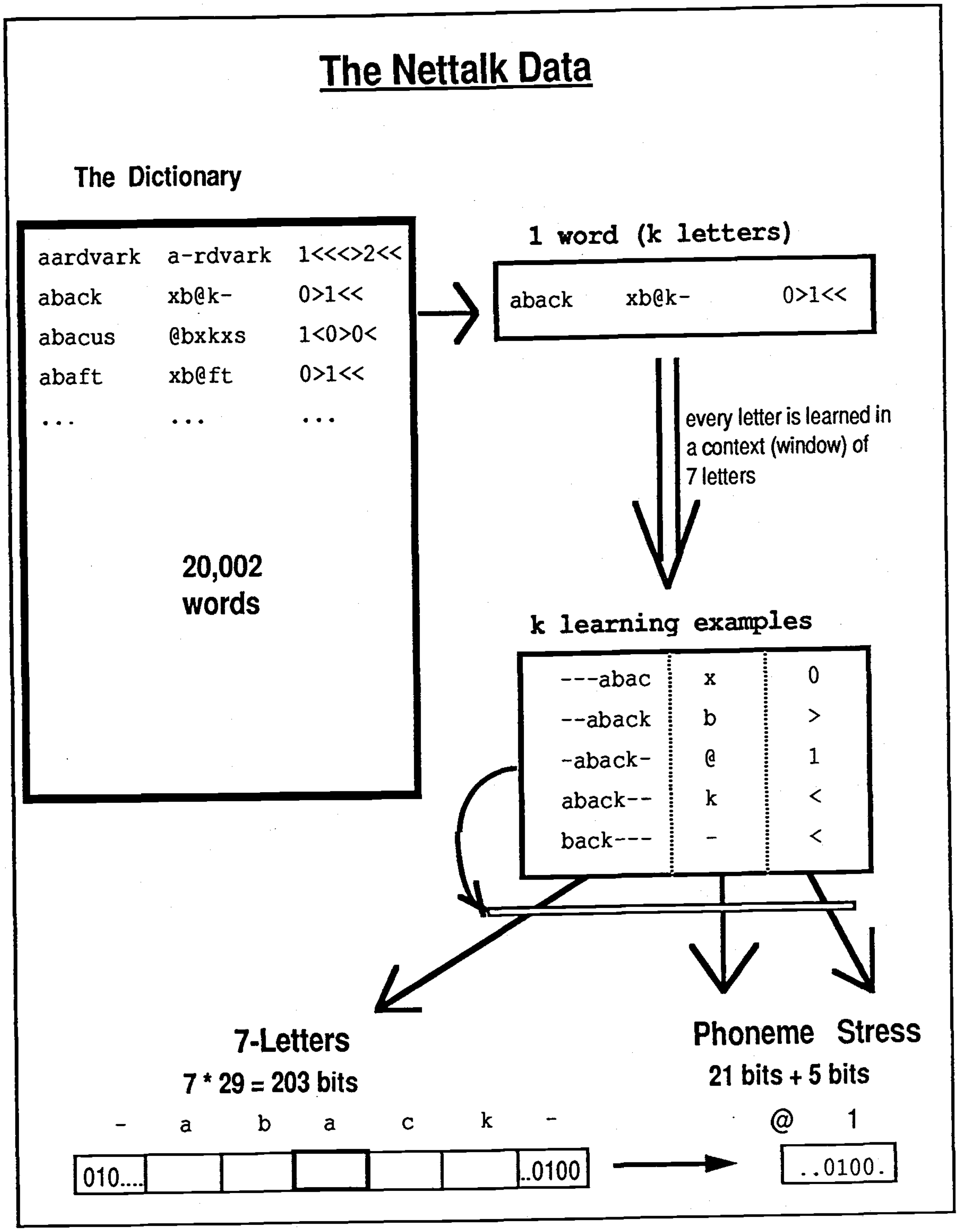

Figure 2. Converting the NETtalk dictionary into learning examples.

in perspective and focus our attention on variations of only certain major aspects of the standard configuration in each subsequent chapter. It will also summarize the "defaults" that can be assumed to hold (in the absence of explicit references to the contrary) throughout the thesis.

The base configuration will be characterized by the following:

Window size: A 7-letter window is employed to represent the context of the current letter: the letter itself (at the center of the window), the 3 letters to its left, and the 3 letters to its right.

Test Set: A test set consisting of 1000 words (drawn randomly and without replacement from the 20,002-word NETtalk dictionary) will be used. We will refer to this as the "standard test set". Unless otherwise stated, all the performance results reported in this thesis will be the performance of the learning system on this 1000-word test set which produces 7,242 windows that need to be classified.

Training Set: A training set consisting of 1000 words (drawn randomly and without replacement from the 19,002-word NETtalk dictionary remaining after removing the "standard test set" from the full dictionary) will be used. We will refer to this as the "standard training set". Unless otherwise stated, all the experiments reported in this thesis will be performed using this "standard training set". These 1000 words produce 7,229 examples, or classified windows.

Input Representation: The input representation scheme introduced by Sejnowski and Rosenberg for the seven-letter windows will be employed. This representation was discussed in more detail in Section 1.4.3.

Output Representation: The standard distributed output code developed by Sejnowski & Rosenberg (described in the previous section) will be employed.

Separate-ps: The above output representation encodes each phoneme and stress separately. This is in contrast to another equally feasible scheme: giving every phoneme/stress *pair* a unique code. This—combined-ps approach introduced in

Section 1.4.4—will also be investigated in subsequent chapters.

Observed Decoding: Unless otherwise stated, observed decoding will be used throughout this thesis. In this scheme, a list is maintained of all the phoneme/stress pairs (ps-pairs) *observed* in the training set, along with their frequency of occurrence. Upon evaluation, the bit-vector output by the learning system is mapped to the ps-pair—from the observed list—whose corresponding bit-vector has the smallest Hamming distance with the output vector. Ties are broken in favour of the most frequent ps-pair.

Algorithm: The algorithm employed is ID3, a simple decision-tree learning algorithm developed by Ross Quinlan [Quinlan83, Quinlan86b]. It constructs a decision tree recursively, starting at the root. At each node, it selects, as the feature to be tested at that node, the feature a_i whose mutual information with the output classification is greatest (this is sometimes called the information gain criterion). The training examples are then partitioned into those examples where $a_i = 0$ and those where $a_i = 1$. The algorithm is then invoked recursively on these two subsets of training examples. The algorithm halts when all examples at a node fall in the same class. At this point, a leaf node is created and labelled with the class in question. The basic operation of ID3 is quite similar to the CART algorithm developed by [Breiman84] and to the tree-growing method developed by [Lucassen83]. Appendix A covers the ID3 algorithm in more detail.

In our implementation of ID3, we did not employ windowing [Quinlan83], CHI-square forward pruning [Quinlan86a], or any kind of reverse pruning [Quinlan87]. Early experiments in [Dietterich90a,b] showed that these pruning methods did not improve performance.

We did apply one simple kind of forward pruning to handle inconsistencies in the training data: If all training examples agreed on the value of the chosen feature, then growth of the tree was terminated in a leaf and the class having more training examples was chosen as the label for that leaf (in case of a tie, the

Table 1. Performance of the base configuration described in this section on the standard training and testing data sets.

Data set used for evaluation	% correct (1000-word data set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
Training set	96.6	99.5	99.8	99.6	100.0	269.9	29.3
Test set	12.5	69.6	81.3	79.2	96.3	269.9	29.3

leaf is assigned to class 0). Note that if all of the examples agree on the value of this feature, then the feature has zero mutual information with the output class (and hence, since this was the *best* feature, every other feature must also have zero mutual information).

The performance of the base configuration is shown in Table 1. This will be the general format we use throughout the thesis to report the performance of our learning systems in this domain. Correctness data are measured by comparing the phonemes and stresses found by the learning system for each window—after mapping to the nearest phoneme/stress pair, if necessary—with the correct answers as given in the dictionary. The meaning of the correctness figures reported for different levels of aggregation are as follows:

Bit: The average correctness of all phoneme and stress bits (after mapping to the nearest phoneme/stress pair, if necessary) .

Stress: Percentage of the windows mapped by the learning system to a correct stress (according to the dictionary).

Phoneme: Percentage of the windows mapped by the learning system to a correct phoneme (according to the dictionary).

Letter: Percentage of the windows mapped by the learning system to a phoneme and a stress that are *both* correct.

Word: Percentage of the words correctly classified by the learning system. A word is correct when all its letters are correct. This implies that for a k-letter word *all* the k windows generated from it must be mapped by the learning system to correct phonemes *and* stresses.

The decision tree statistics shown are:

Leaves: The average number of leaves contained in all the decision trees built by ID3 for the phoneme and stress bits.

Depth: The average maximum depth of all phoneme and stress ID3-trees.

There are several things to note in Table 1. First, the performance of the learning system on the training set is uninteresting. Decision tree building algorithms such as ID3 always perform extremely well on the training set. In fact, had it not been for the presence of some inconsistencies in the training set, all the performance figures shown in the first row of Table 1 would have been 100% . For this reason, we will not report the performance of our learning systems on the training set in the rest of this thesis. Second, due to the aggregation problem mentioned earlier, performance at higher levels of aggregation (e.g. the word level) is disappointingly low when compared with the performance at lower levels of aggregation such as the stress or phoneme level. Another thing to observe is that the bit-level performance of 96.6% is somewhat misleading since only 3 to 5 bits in the 21-bit phoneme vector are set to '1'. Hence, guessing constantly '0' will already achieve a correctness higher than 85%.

1.6 A Reader's Guide to the Thesis

This thesis examines numerous variations to several of the parameters characterising the base configuration discussed in the previous section. The following summary may help readers interested in specific issues.

Chapter 2 reviews some of the previous work relevant to this thesis. Emphasis will be on the more recent work that deals with the application of machine learning techniques to the text-to-speech domain.

Chapter 3 explores several alternatives to the standard *output representation* employed in the base configuration. The three approaches for multiclass learning are compared to a new—superior—technique in which BCH error-correcting codes are employed as a distributed output representation. Random codes are also considered. Experiments are performed to determine which properties of these codes lead to the improved performance. Finally, the impact of employing voting among several sets of decision trees is examined for a number of the multiclass learning methods.

Chapter 4 explores several alternatives to the standard *input representation* employed in the base configuration. We investigate the effects of incorporating the output bits accumulated so far as part of the context for the current letter. We then show the impact of extending the context by including the *phonetic* context of the preceding letters as part of the context for the current letter. We also address the consequences of abandoning the weight-1 input encoding and explore an information theoretic approach[7] to defining a "good" set of binary attributes to represent the extended context. The effects of enlarging the window size and processing the letters of the words in reverse are also explored. Finally, we combine the input techniques developed in this chapter with the error-correcting output technique and the voting method introduced in Chapter 3. The results show that the benefits offered by the output techniques of Chapter 3 are nearly orthogonal to the benefits provided by the improved input techniques developed in this chapter. We conclude by presenting the results of the best performing learning system we have studied in this domain (trained on our standard 1000-word training set).

Chapter 5 evaluates the performance of a recent method introduced by David Wolpert for the NETtalk task [Wolpert90c]. We improve on this method and show

[7]This was an attempt to reproduce the experimental conditions reported in [Lucassen83] so that a fair comparison of their methods to ours could be made.

that—even with our enhancements—it still performs worse on this task than ID3 when combined with our technique of using distributed error-correcting codes for the output.

In all the experiments reported in chapters 1 to 5, learning is accomplished by training on our standard 1000-word training set. However, Chapter 6 is performance oriented. The results of the best performing learning systems trained on 19,002 words are compared with the performance of the DECtalk rule-base on our standard 1000-word test set. Chapter 7 presents the conclusions and discusses several possible directions for future work.

The appendices include descriptions of several algorithms discussed in the thesis, details of some of the codes used as part of the input or output representations, statistics on the NETtalk dictionary and the training set employed, as well as various mutual information data calculated between the elements of the extended context and the desired outputs.

Chapter 2

Previous Work

Several early attempts for the automatic generation of phonemes and/or stresses are reported in the literature. Dennis Klatt and David Shipman [Klatt82], for example, attempted the development of a semi-automatic procedure for discovering letter-to-phoneme rules in 1982. Kenneth Church covered the stress assignment problem for letter-to-sound rules in [Church85]. A host of other attempts are cited in [Klatt87], which contains an extensive bibliograghy of the overall process of text-to-speech conversion. Rather than clutter this survey of previous work with these early attempts, we will focus only on the most recent work that emphasized the machine learning approach to the task of text to speech conversion. With the exception of the work of Lucassen & Mercer presented in Section 2.1.1, these will primarily be concentrated on results published after Sejnowski & Rosenberg's pioneering work on NETtalk [Sejnowski87]. Their work must be credited with popularizing this task and providing a systematic way of applying standard machine learning techniques to tackle it.

In Section 1.3.2, we described three general approaches to the multiclass learning problem. In this chapter, we will present the results of previous studies that have applied these approaches in the English text to speech conversion domain. These will include some of our own results, obtained early in this project. These early results were documented elsewhere[8], and hence, will not be emphasized in this thesis.

Appendix A describes the various algorithms that are discussed in the sections that follow.

2.1 The Direct Multiclass Approach

2.1.1 Lucassen & Mercer's Multiclass Decision Trees

In 1980, a project was initiated at the Speech Processing Group at the IBM Thomas J. Watson Research Center with the objective of finding some way of generating baseforms[9] for new words automatically. Their system was to utilize the spelling of the word along with a few sample utterences of it (possibly by different speakers) to determine the baseform. One part of the system would determine a set of possible baseforms (with likelihood estimates) from the spelling of the word. The second part would then attempt to narrow down the choices by taking advantage of the sample utterences of the word.

Only the first part of their system is relevant to this discussion. They used a 70,000-word phonemic dictionary as their starting point, generating some 500,000 examples as their training set. They employed a 9-letter window and included the phonemes of the four letters to the left of the current letter (letter at the center of the window) in the context for the current letter. These phonemes can be obtained, during execution, from the letters that have already been pronounced during a left-to-right scan. They converted this *extended context* to a binary representation by employing a minimum set of features (or *binary questions* in their words), defined through an elaborate procedure aimed at maximizing the mutual information between these features and the output phonemes (we will reproduce

[8]See [Hild89], [Dietterich90] and [Dietterich91]

[9]The (phonemic) baseform of a word is a string of phonemes that describe the normal pronunciation of the word.

their work on feature set selection in Chapter 4).

The basic operation of the tree-growing method developed by Lucassen and Mercer was quite similar to ID3. The main differences were that a host of ad-hoc mechanisms were employed to limit the growth of the decision trees and that the leaves of the trees were assigned a *collection* of classes (phonemes) with a probability estimate on each class.

The performance of this component of their system (tested on a 500-word test set) was reported as better than 79%.[10]

One weakness of the work reported in [Lucassen83] is that it presents the performance results for one particular choice of input and output representations. In our work, we tried to reproduce their results (as closely as feasible) under standard conditions that can be easily compared with other methods that we investigated. We show, for example, that their elaborate method for selecting "good" binary features is inferior to the simple choice of selecting a local encoding (i.e. weight-1 codes) for the letters, phonemes and stresses used as part of the extended context.[11] Similarly, we present in Chapter 3 several higher performance alternatives to the direct multiclass approach employed in their study. On the other hand, their decision to incorporate the phonetic context of the previous letters in the context for the current letter proved to have mixed results. Including the *left* phonetic context did not help, but incorporating the *right* phonetic context of the previous letters in the context for the current letter proved to be a judicious decision offering a sub-

[10]An error rate of 21% was reported in [Lucassen83], but several reasons were given as to why the actual error rate was probably slightly lower.

[11]In fairness to Lucassen's work, we must point out that a weight-1 code for the elements in the extended context for the current letter results in a substantial number of features—383 for the weight-1 codes compared with 90 for our version of the code developed by their technique, for a window size of 7. This results in a substantial difference in the running times of decision tree building algorithms running on the two alternative coding schemes, and it may have been an important factor in their decision to search for as few input features as possible—given the huge number of training examples and the computing power available in 1983.

stantial improvement in the overall performance in this domain. Similarly, their idea of utilizing the mutual information between each letter, phoneme and stress and the output for the current letter is exploited in Section 5.2 as a justifiable means of computing a "good" set of weights for the Wolpert method [Wolpert90c].

2.1.2 Multiclass ID3

Multiclass ID3 is a generalized version of ID3 that stores integer class numbers at the leaves. [Shavlik89] applied this algorithm to the NETtalk domain and used it to construct a single tree to classify the 115 classes (phoneme/ stress combinations) present in an 808-word training set used in their experiments. In early experiments during the course of this work, we also employed our implementation of Multiclass ID3 to learn two trees: One to predict the 54 phonemes and another (separate) tree to classify the 6 stresses. These, unlike others reported in this thesis, were trained using the 1000 words in the NETtalk dictionary that were most common according to [Kuchera67] as the training set. The performance results of 64.9% obtained by us were nearly indistinguishable from the 64.2% reported in [Shavlik89]. The slight disparity may be due to the fact that [Shavlik89] was using a training set of 808 words instead of 1000. These figures measured correctness at the letter level on the test set.[12]

2.2 The One-per-class Approach

[Shavlik89] assigned a class number to each distinct phoneme/stress pair appearing in the 808 words selected from the NETtalk dictionary to be their training set. The above strategy was then used to learn the resulting 115 classes using the Perceptron and the Backpropagation learning algorithms.

[12]We used a 19000-word test set, while [Shavlik89] was using a 1000-word test set in the experiments being compared.

2.2.1 Perceptron

One perceptron was trained for each of the 115 classes. The collection of perceptrons was then tested on a 1000-word test set. Each example was classified by passing it through all the perceptrons and assigning it to the class whose perceptron's output exceeded its threshold by the largest amount. A classification performance of 49.2% on the test set was reported. The bad performance of the perceptron is understandable in such a complex domain, since perceptrons are only capable of learning concepts that are linearly separable.[13]

2.2.2 Backpropagation

[Shavlik89] also used the Backpropagation algorithm to train a connectionist network with 115 units in the output layer: one for each class. This method of using Backpropagation was found substantially inferior[14] to that discussed in Section 2.3.1, so we will not elaborate on it any further.

2.3 The Distributed Coding Approach

As discussed in Section 1.4.5, we can reformulate the multiclass problem by choosing a suitable boolean encoding for the classes. In the following sub-sections, we present the results obtained by applying various binary learning algorithms to the NETtalk domain using this ditributed coding approach. The particular output encoding employed is the 26-bit distributed code adopted from [Sejnowski87] as detailed in Section 1.4.5.

Several decoding strategies for mapping the 26-bit output vector produced by the classifier to a legal phoneme/stress pair are discussed in Section 2.4. The decoding strategy "best guess (1)" is followed in the results reported below. That

[13]Only 60 out of the 115 perceptrons converged in these experiments [Shavlik89], so 55 of the classes were not (easily) linearly separable.

[14]63.0% vs. 72.3% correctness on the test set.

is, if the output vector does not correspond to any legal phoneme/stress pair, then it is mapped to the phoneme/stress pair having the smallest Hamming distance with the output vector.

2.3.1 NETtalk

Sejnowski and Rosenberg were the first to apply a connectionist learning method to the problem of English text-to-speech conversion. In fact, the data and its format as introduced in Section 1.4 are due to [Sejnowski87].

Sejnowski's NETtalk: The network used in the NETtalk experiment had three layers: an input layer, one hidden layer and an output layer. The input layer was built from 203 processing elements, one for each bit in the input vector (i.e. the binary representation of a 7-letter word given in Section 1.4.3). The hidden layer had 120 processing elements and the output layer 26 (21 of which encoded the phonemes and the remaining 5 encoded the stresses). The network was fully connected, thus requiring 27,809 weights.[15] Figure 3 depicts the architecture of the network.

The Backpropagation algorithm [Rumelhart86] was applied to train the weights. Even though [Sejnowski87] describes the network outputs in terms of both phonemes and stresses, the task is defined as "converting strings of letters to strings of phonemes." Performance figures of 98% (77%) on the training (test) set are reported.[16] These are the percentage of correct phonemes predicted by the network.

Backpropagation training was repeated more recently by [Shavlik89] and [Diet-

[15]There are $203 + 120 + 26 = 329$ processing elements and $203 \times 120 = 24,360$ connections between input and hidden layer and $120 \times 26 = 3,120$ connections between the hidden and the output layer. With one additional weight per node for the "bias term" this sums up to a total of $329 + 24,360 + 3120 = 27,809$ weights.

[16]The test set they employed was the whole dictionary which included the training set. Therefore the above figures are about 5% higher than what they would have been—had the test set been disjoint from the training set.

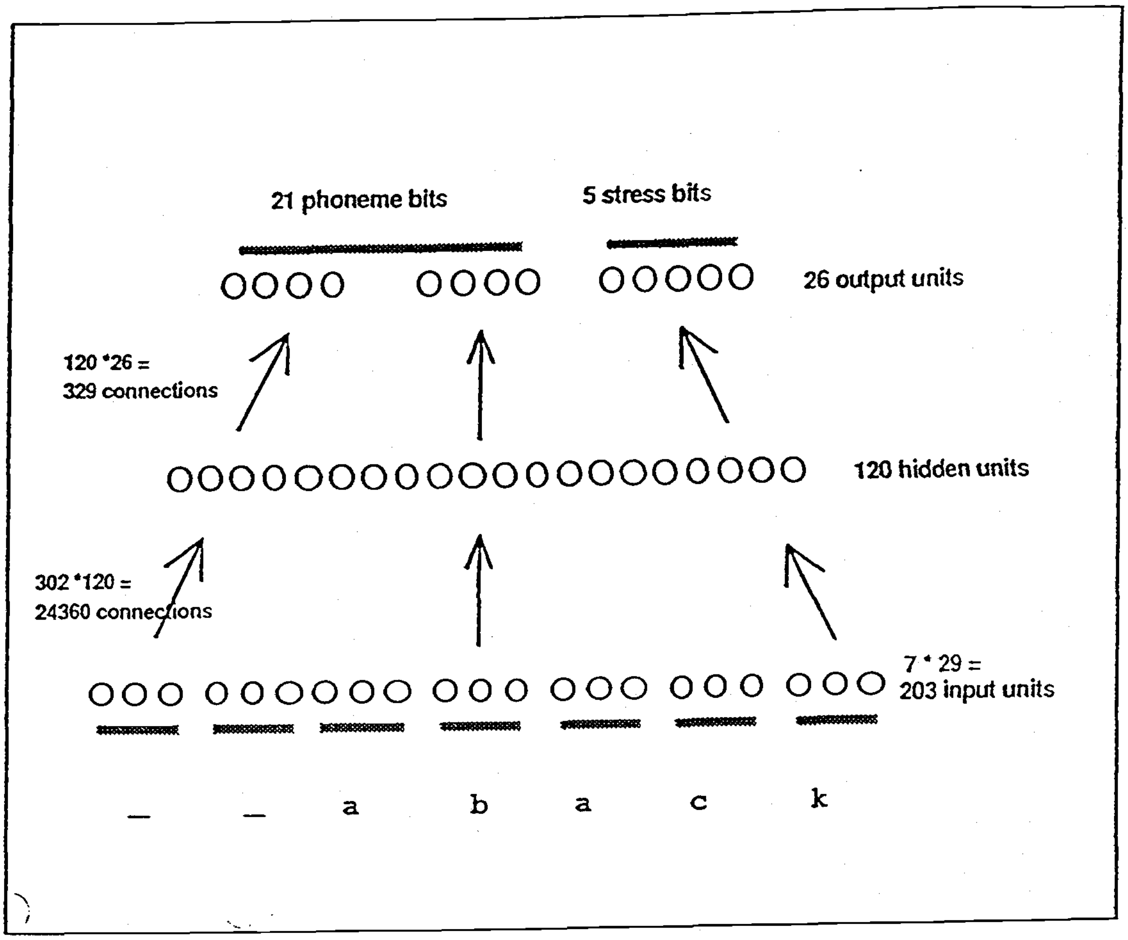

Figure 3. Network architecture used in the NETtalk experiment.

terich89b]. [Shavlik89] reported performance figures of 96% (72%) on the training (test) set. These figures are correctness at the letter level. [Dietterich89b] achieved 99.7% (78.7%) correctness on the training (test) set at the phoneme level and 99.4% (65.6%) correctness on the training (test) set at the letter level. [Sejnowski87] and [Dietterich89b] used as a training set, the 1000 words in the NETtalk dictionary that were most common according to [Kuchera67], while [Shavlik89] used only the 808 most common English words as the training set. The decoding strategies used by [Shavlik89] were also slightly different from the ones used by [Dietterich89b].

2.3.2 ID3 / FRINGE

Using the same output encoding as that of [Sejnowski87], the ID3 algorithm was employed to build 26 trees: one for each of the 26 bits used to code the phonemes and stresses [Shavlik89, Hild89]. As was done for the connectionist network, the outputs of the trees were mapped to the nearest phoneme/stress pairs for performance evaluation. We also applied the FRINGE algorithm to the NETtalk data.

Figure 4 compares the performance of the three algorithms: ID3, FRINGE and Backpropagation, on a test set of 1000 words, at the phoneme, stress, letter and word level as a function of the size of the training set used for learning. The algorithms were trained using the 1000 words in the NETtalk dictionary that were most common according to [Kuchera67].

2.4 Decoding Strategies

[Hild89] investigated the impact of various decoding strategies on the performance of ID3 applied to the NETtalk domain as described in the previous section.

Recall that in the testing phase, the 7-letter window corresponding to the letter we want to classify is presented to the system as a 203-bit vector. The system outputs two bit vectors: a 21-bit vector representing the output phoneme and a 5-bit vector representing the output stress associated with the given input vector. Since there are only 54 phonemes and 6 stress symbols, it follows that not all the $2^{21}(2^5)$ potential answers for a phoneme (stress) bit vector will represent a legal phoneme (stress). Let $\vec{P}$ be the phoneme bit vector found by the algorithm and LP be the set of the 54 legal phoneme vectors. To find a legal phoneme for an arbitrary phoneme bit vector, one of the following "best guess" strategies can be applied:

(0) Do nothing. $\vec{P}$ is considered wrong if $\vec{P} \notin LP$

(1) Find a legal phoneme vector $\vec{P_L} \in LP$ with the smallest Hamming distance [17] to $\vec{P}$. Ties are broken arbitrarily.

(2) As in (1). Additionally, if there is more than one candidate having the smallest Hamming distance, we choose the more likely one. In order to know the likelihood of a particular candidate, the following statistic is maintained. For every letter L in the training set, we count how often the letter is mapped to each phoneme. The relative frequency of the pair (L, P_i) is interpreted as the probability that the letter L maps to the phoneme P_i.

(3) To find the best guess, we consider only the phonemes to which the current letter was observed to map to in the training set. Again the more likely phoneme is used in the case of a tie.

(4) As in (3). Additionally, if the letter is part of a larger *common block*—group of letters such as "ous", "tion"—outputs corresponding to the letters of the entire block are decoded by considering only the group of phonemes to which the current block of letters was observed to map to in the training set.

Similar schemes can be applied to find the closest stress vector.

Table 2 (reproduced from [Hild89]) shows the impact of the above "best guess" strategies on performance at different levels of aggregation. These results clearly show that exploiting statistical information present in the training data can significantly improve the classification performance of ID3 on the NETtalk domain.

It should be noted that the Backpropagation experiments reported in Section 2.3.1 employ best guess (1) above as a decoding strategy. Hence, their results have to be compared with the second line of the above table. Based on this comparison, the Backpropagation algorithm outperforms ID3 by several percentage points.

[17]The Euclidean and Hamming distances are equivalent measures for comparing distances between bit vectors.

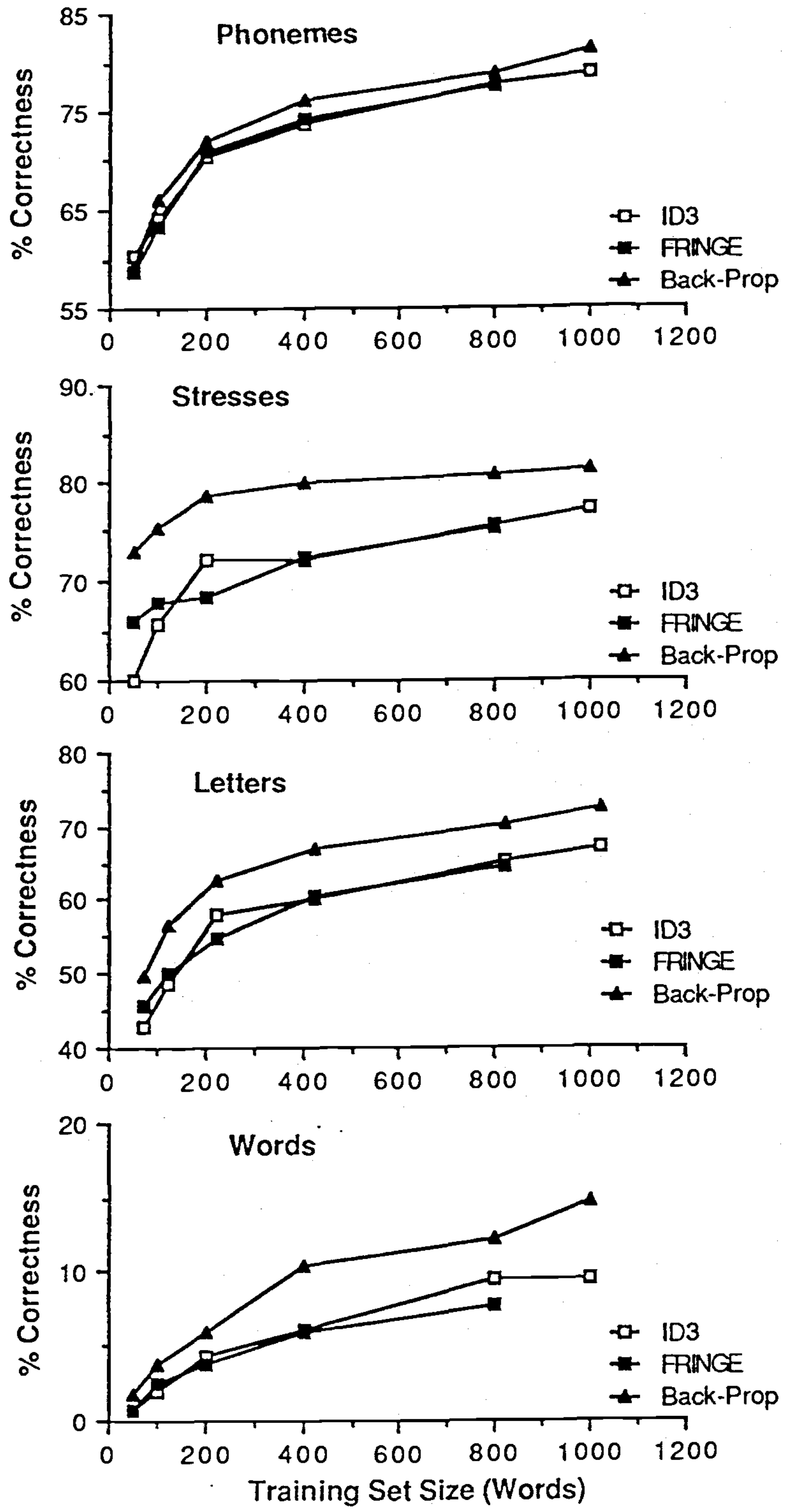

Figure 4. Performance on a 1000-word test set measured as % of: Phonemes, Stresses, Letters and Words, correctly predicted.

Table 2. Impact of different "best guess" (decoding) strategies on performance.

	% correct (19003-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Decoding strategy	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
best guess (0)	6.0	57.0	69.9	73.3	96.0	165.3	23.1
best guess (1)	9.1	60.8	75.4	74.4	95.8	165.3	23.1
best guess (2)	11.9	66.1	78.4	78.0	96.2	165.3	23.1
best guess (3)	12.9	67.2	79.9	78.0	96.2	165.3	23.1
best guess (4)	15.8	69.3	80.3	78.6	96.2	165.3	23.1

The above discussion assumes that the closest phoneme and stress vectors are found independently. An alternative approach is to treat the concatenated phoneme/stress vector $\vec{PS}$ as an entity and try to find the closest phoneme/stress vector $C\vec{PS}$ from a set APS of *allowable* phoneme/stress bit vectors. Depending on the choice of the elements in the set APS, several decoding strategies are possible:

Legal Decoding: All the possible 324 (legal) phoneme/stress vectors[18] are allowed in the set APS. (This is the same as "best guess (1)" discussed earlier.)

Observed Decoding: Only the phoneme/stress vectors observed in the training set are allowed in the set APS. Additionally, if there is more than one candidate for the closest phoneme/stress vector $C\vec{PS}$, we choose the more likely one (more frequent in the training set).

Observed Decoding Given the Current Letter: Only the phoneme/stress vectors to which the current letter was observed to map in the training set are allowed in the set APS. Again we choose the more frequent $C\vec{PS}$ to

[18]The cross product of the 54 phoneme vectors concatentented with the 6 stress vectors.

break ties.

Block Decoding: As above, but if the letter is part of a larger *common block*—group of letters such as "ous", "tion"—outputs correspondng to the letters of the entire block are decoded by considering only the group of phoneme/stress pairs to which the current block of letters was observed to map in the training set.

We will employ observed decoding for *all* the experiments reported in this thesis. The reported results can be made *marginally* better by employing *observed decoding given the current letter*. *Block decoding* was not fully investigated because it was not directly compatible with the *extended context* discussed in Chapter 4.

2.5 Choice of the Training Set

Most of the previous work reported in earlier sections, employed a training set consisting of the 1000 most common English words according to [Kuchera67].[19] On the other hand, our standard training set used throughout this thesis contains 1000 words *randomly* selected (and without replacement) from the 20,003-word NETtalk dictionary. We decided to employ a training set of randomly selected words after initial experiments (see [Hild89]) showed that randomly selected training sets of 1000 words invariably perform better than the most common 1000 English words. This is understandable for two reasons:

1. Most common English words are also the most irregular. A case in point is the pronunciation of the letter "f". Letter "f" maps to phoneme /f/ in all but the word "of", one of the more common English words.

[19]In our case, the most common 1000 English words were extracted from the 20,003 dictionary obtained from [Sejnowski87] by scanning the most common words reported in [Kuchera67] in order, and including in the training set those words that also appeared in the NETtalk dictionary. The scanning stopped once we gathered 1000 words in the training set.

2. Common English words are short in general. Hence the number of training examples obtained from a 1000-word randomly selected training set is significantly more than the number of examples obtained from the 1000 most common English words.[20]

2.6 ID3talk vs NETtalk

Experiments run on the NETtalk domain show that backpropagation performed noticeably better than ID3. This was the case in the experiments reported in the previous sections which used the most common 1000 English words as a training set. A more careful study is reported in [Dietterich90a,b]. In this section we will discuss the main findings of that study. The interested reader is referred to the subject reference for further details.

Table 3 (reproduced from [Dietterich90b]) compares the performance of the two algorithms ID3 and Backpropagation (BP) on the NETtalk task. In these experiments, training was performed using our standard 1000-word training set, and performance was tested on our standard (disjoint) test set. The decoding strategy employed was what we term "legal" decoding—corresponding to best-guess (1) in Section 2.4. It is clear from these figures that BP outperforms ID3 on this task. Furthermore, virtually every difference in the table at the word, letter, phoneme, and stress levels is statistically significant.

To explain the differences between ID3 and BP, we formulated three hypothesis:

Hypothesis 1: Overfitting. ID3 has overfit the training data, because it seeks complete consistency. This results in poor generalization and causes ID3 to make more errors on the test set.

Hypothesis 2: Sharing. The ability of BP to share hidden units among all of the output functions being learned (f_i), allows it to reduce the aggregation problem

[20]7,229 examples from our standard training set vs. 5,521 examples produced from the 1000 most common English words.

Table 3. Performance of ID3 and Backpropagation with "legal" decoding.

	% correct (1000-word test set)				
	Level of Aggregation				
Method	Word	Letter	Phoneme	Stress	Bit (mean)
ID3	9.6	65.6	78.7	77.2	96.1
BP	13.6**	70.6***	80.8***	81.3***	96.7*

Difference in the cell significant at $p < .05^*$, $.005^{**}$, $.001^{***}$

at the bit level and hence perform better.

Hypothesis 3: Statistics. The numerical parameters in the BP network allow it to capture statistical information that is not captured by ID3.

We performed experiments to test these hypotheses as follows:

Tests of Hypothesis 1: Overfitting. We implemented and tested three methods covering three basic strategies reported in the literature for dealing with the overfitting problem in ID3-like algorithms. None of these techniques improved the performance of ID3 on this task. This suggests that Hypothesis 1 is incorrect.

Tests of Hypothesis 2: Sharing. To test this hypothesis, we decided to *remove* sharing from backpropagation, by training 26 independent networks, each having only one output unit, to learn the 26 f_i mappings. If Hypothesis 2 is correct, then, because there is no sharing among these separate networks, we should see a drop in performance compared to the single network with shared hidden units. Furthermore, the decrease in performance should decrease the differences between BP and ID3. This did not turn out to be the case. Removing sharing from BP did not make BP and ID3 more alike. We conclude that sharing does not explain why ID3 and BP are performing differently on this task. Hence Hypothesis 2 is also incorrect.

Tests of Hypothesis 3: Statistics. The ability of the numerical parameters in

Table 4. Performance of ID3 and BP with "observed" and "block" decoding strategies.

Decoding Method	% correct (1000-word test set)				
Learning	Level of Aggregation				
Algorithm	Word	Letter	Phoneme	Stress	Bit (mean)
Observed decoding:					
ID3	13.0	70.1	81.5	79.2	96.4
BP	14.3	71.5	82.0	81.4	96.7
Block decoding:					
ID3	17.5	73.2	83.8	80.4	96.7
BP	19.3	73.7	83.6	81.4	96.7

the BP network to capture statistical information—not captured by ID3—turned out to be the main reason for the observed differences in the performance of BP and ID3 on this task. Several experiments supported this. One experiment showed that thresholding the outputs of the Backpropagation network [21] before mapping to the nearest legal phoneme/stress pair significantly drops the performance of Backpropagation. Other experiments showed that more sophisticated decoding techniques that utilize the statistical information present in the training set (such as "observed" and "block" decodind) significantly improve the performance of ID3 but does not improve the performance of BP by nearly as much. These decoding strategies bring the performance of the two algorithms to nearly the same level, as shown in Table 4.

2.7 Choice of the Algorithm

Even though we will be demonstrating the effectiveness of the techniques we develop in the following chapters mainly through the use of the basic ID3 algorithm, or through some variation of it, it must be stressed that these techniques are to a

[21] Values $> .5$ were mapped to 1, values $\leq .5$ were mapped to 0.

large extent independent of the particular learning algorithm employed. Efficiency considerations are the primary reason for restricting ourselves to ID3 variants. Unlike, Backpropagation, for example, ID3 scales very well with the number of training examples and the number of features selected to represent each example, both of which can become very large in our experiments. Furthermore, the faster runtime of ID3 runs (at least 60 times faster than comparable BP runs) allows us to make a large number of runs under varying experimental conditions in a "reasonable"[22] time period. The comparable results of the performance of ID3 and BP with observed decoding—as shown in Table 4 in the previous section—suggests that similar results would be obtained with backpropagation.

[22]Even with ID3, some of the larger runs described in Chapter 6 take several CPU-weeks to train. For example, building the 127 decision trees for the 127-bit, $d = 63$ BCH output code from a training set of 19,002 words with a window size of 15 takes about 3 CPU-weeks on a SUN 4 workstation. A similar run using backpropagation is not feasible with the current computational facilities under our control.

Chapter 3

Output Techniques

Three general approaches to the multiclass learning problem were discussed in Section 1.3.2. These were (a) the direct application of multiclass algorithms such as the decision-tree algorithms ID3 and CART, (b) application of binary concept learning algorithms to learn individual binary functions for each of the C classes, and (c) application of binary concept learning algorithms with distributed output codes such as those employed in our base configuration introduced in Section 1.5.

This chapter compares these three approaches to a new technique in which BCH error-correcting codes are employed as a distributed output representation. We show that these output representations improve the performance of ID3 on the NETtalk domain with our standard input representation.[23]

Following this, we will evaluate the performance of randomly-generated output codes and show that it is only slightly worse than that of BCH error correcting codes. We perform experiments to show that good error-correcting codes can be designed by generating random binary strings, instead of by using BCH methods.

Finally in Section 3.6 we explore the impact of employing voting among several sets of decision trees on the performance of these methods.

[23]The next chapter generalizes this result and shows that employing error-correcting codes as a distributed output representation improves the performance of ID3 on the NETtalk domain irrespective of the feature set used to represent the input.

The input representation employed throughout this chapter is the binary representation of the standard 7-letter window discussed in Section 1.5. Combining the output techniques developed here with alternative input represention will be covered in Chapter 4.

3.1 Multiclass ID3

In the direct multiclass approach, the Multiclass ID3 algorithm is applied once to produce a decision tree whose leaves are labelled with one of the 126 phoneme/stress classes. This is the combined-ps (combined phoneme/stress) approach. The other alternative is to apply Multiclass ID3 twice to learn two multiclass trees: One labeled with one of 54 phoneme classes and the other labeled with one of 6 stress classes. This is the separate-ps approach.

Table 5 compares the performance of the above two methods of applying multiclass ID3 to the performance of our base configuration. There are a couple of things to observe. First, the direct multiclass and distributed output codes employed in our base configuration performed equally well—given the same standard input encoding. Indeed, the statistical test for the difference of two proportions cannot distinguish them. Another thing to note is the difference in the performance figures for the combined-ps and the separate-ps approaches. The combined-ps approach generally give better performance results at the letter and word levels of aggregation, while the separate-ps approach seems to do slightly better at the individual phoneme and stress levels.

3.2 One-per-class Output Codes

In the one-per-class approach, ID3 is applied 126 times to learn a separate decision tree for each class. When learning class i, all training examples in other classes are considered to be "negative examples" for this class. When the 126 trees are

Table 5. Comparison of the performance of two methods of applying multiclass ID3 to the performance of the base configuration

	Number	% correct (1000-word test set)				Decision Tree	
	of	Level of Aggregation				data (mean)	
Configuration	Trees	Word	Letter	Phoneme	Stress	Leaves	Depth
Multiclass (combined-ps)	1	13.5	70.8	81.1	78.3	1987.0	54.9
Multiclass (separate-ps)	2	13.0	69.7	82.4	79.5	1305.0	48.0
Base Configuration	26	12.5	69.6	81.3	79.2	269.9	29.3

applied to classify examples from the test set, ties are broken in favor of the more-frequently-occurring phoneme/stress pair (as observed in the training set). In particular, if none of the trees classifies a test case as positive, then the most frequently occurring phoneme/stress pair is guessed.

The above discussion assumes the combined-ps approach. Alternatively, we could apply the one-per-class approach on the phonemes and stresses separately. In this case, ID3 is applied 60 times to learn a separate decision tree for each of the 54 phonemes and the 6 stresses. This is the separate-ps approach. When the 60 trees are applied to classify examples from the test set, rather than breaking ties for the phonemes and stresses separately, our implementation *still* breaks ties in favor of the more-frequently-occurring phoneme/stress *pair* (as observed in the training set).[24]

Table 6 compares the performance of the above two methods of applying the one-per-class approach to the performance of our base configuration. The one-per-class, combined-ps method performed markedly worse. In fact, all the differences between this method and the other two (the distributed representation employed

[24]This is conveniently achieved by mapping the 60-bit string output by the decision trees to the nearest 60-bit vector from a set S of vectors corresponding to phoneme/stress pairs observed in the training set. Each vector in S is the concatenation of a 54-bit, weight-1 vector representing a phoneme and a 6-bit, weight-1 vector representing a stress.

Table 6. Comparison of the performance of two methods of applying the one-per-class approach to the performance of the base configuration

Configuration	Number of Trees	% correct (1000-word test set) Level of Aggregation: Word	Letter	Phon.	Stress	Bit	Decision Tree data (mean): Leaves	Depth
One-per-class:								
combined-ps:	126	8.7	66.7	76.4	74.5	99.5	34.9	10.5
separate-ps:	60	11.8	69.5	80.6	78.9	98.7	90.4	14.0
Base Configuration:	26	12.5	69.6	81.3	79.2	96.3	269.9	29.3

in the base configuration and the multiclass method) are statistically significant at or below the .01 level. The separate-ps case of the one-per-class approach fares much better than the combined-ps case. The main reason for this could be that many more training examples can be obtained from the 1000-word training set for each phoneme and stress separately than for each phoneme/stress taken as a pair. Still, the performance of even the separate-ps case lags (slightly) behind that of our configuration. We will cover an effective method of boosting the performance of this approach to multiclass learning in Section 3.6.

3.3 Decision Tree Statistics

Let us consider the relative difficulty of training for each of the multiclass methods, as measured by the sizes of the decision trees produced. Table 7 reproduces the decison tree statistics for the various approaches discussed earlier: the base configuration, multiclass and one-per-class. The smallest decision trees are those learned by the one-per-class approach. On the average, the members of one class can be discriminated from all of the others by a tree having 34.9 leaves for the combined-ps case (90.4 leaves for the separate-ps case). When we move to the distributed output code, the individual decision trees become larger, and hence, more

Table 7. Decision tree statistics for the various multiclass methods.

	Number	Decision Tree Statistics		
	of	Average		Total
Configuration	Trees	Leaves	Depth	Leaves
Multiclass:				
combined-ps:	1	1987.0	54.9	1987
separate-ps:	2	1305.0	48.0	2610
One-per-class:				
combined-ps:	126	34.9	10.5	4397
separate-ps:	60	90.4	14.0	5424
Base Configuration:	26	269.9	29.3	7017

difficult to learn. Each of these trees had an average of 270 leaves. Hence, when we shift from recognizing a single class to recognizing a disjunction of classes, the learning generally task becomes more difficult. Finally, when we must discriminate among all 126 classes simultaneously, the decision tree grows to have 1,987 leaves.

On the other hand, the total complexity of the hypotheses produced by each method—as measured by the total number of leaves in all of the decision trees—shows that the simplest hypothesis is produced by the direct multiclass approach (combined-ps case) with 1,987 leaves. The one-per-class approach produces a total of 4,397 leaves for the combined-ps case (5,424 leaves for the separate-ps case), and the distributed output approach uses 7,017 leaves!

3.4 Error-correcting Codes

Closer examination of the distributed output code approach suggests that a better distributed output code could be designed using error-correcting code methods. Good error-correcting codes choose the individual codewords so that they are well-separated in Hamming distance. The potential benefit of such error correction is that the system could recover from errors made in learning the individual binary functions. However, unlike the distributed code shown in Appendix B, the individual bit positions of such error-correcting codes are not meaningful in the domain. Hence, the individual binary functions to be learned correspond to arbitrary disjunctions of the original C classes. If these functions are difficult to learn, then they may negate the benefit of the error correction.

In the following subsections we investigate this approach using error-correcting codes designed via the BCH method (Bose, Chaudhuri and Hocquenghem [Bose60, Hocquenghem59]). We will first show how to generate and select good BCH error-correcting codes. Following that we present the results of applying codes of varying length designed using these procedures to the text to speech mapping domain. The results will show that while the individual bits (binary functions) of these codes are indeed more difficult to learn, the generalization performance of the system is improved. Furthermore, as the length of the code n is increased, additional performance improvements are obtained. We will also show that these improvements are orthogonal to the size of the sets used during the training phase and to the particular assignment of codewords to individual classes.

Finally we present three limitations of the approach and suggest possible means of overcoming these limitations in Section 3.4.8.

3.4.1 Generating BCH Codes

We used Appendix C of [Lin83] as our starting point for generating BCH codes. This appendix lists for each value of n, k and t the generator polynomial that can be used to generate 2^k, n-bit codewords that are at least a distance $d = 2t+1$ from one another. The BCH method guarantees that *all* the rows of the code (i.e., the codewords) will be separated from each other by the specified minimum Hamming distance d. Consequently, any $t = \lfloor (d-1)/2 \rfloor$ errors will be corrected, because if at most t errors have occurred, the nearest codeword is still the correct codeword. In contrast, the minimum Hamming distance between any two codewords in the

Sejnowski-Rosenberg output code is 1, since many phoneme pairs differ only by whether or not they are voiced. Similarly, in the one-per-class and class-tree approaches, an error in any one of the learned binary functions causes new examples to be misclassified.

To make the code generation procedure concrete, let us examine in some detail the generation of $C = 54$ codewords of distance 15 for our 54 phonemes. First, we find the value of k such that $2^k \geq C$. In this case $k = 6$ is adequate to produce $2^6 = 64$ codewords. Next the value of t is trivially computed from the desired d of 15, since $t = \lfloor (d-1)/2 \rfloor = 7$. The table is then looked up for the closest match for $k \geq 6$ and $t = 7$. In this case an exact match is found as follows:

n	k	t	Generator Polynomial
31	6	7	313365047

If, on the other hand, we were looking for distance $d = 5$ and the same 54 codewords, then the closest match for $k \geq 6$ and $t = 2$ would be

n	k	t	Generator Polynomial
15	7	2	721

which generates $2^7 = 128$ codewords.

The generator polynomial obtained from the appendix is represented in octal. When the octal representation is expanded in binary, the binary digits are the coefficients of the polynomial, with the high order coefficients at the left. For example, the generator polynomial corresponding to the octal representation 721 (or binary 111 010 001) is

$$g(X) = X^8 + X^7 + X^6 + X^4 + 1$$

A simple routine written in C (see Appendix C) [25] is used to generate the 2^k codewords. This routine takes as input the values of n, k and t and the *binary*

[25] We thank Dr. Sulaiman Al-Bassam for providing us with this C routine.

representation of the generator polynomial and prints out the full set of codewords. As an example, the input to the program for the first case ($d = 15$) will be as follows:

```
31 6  7
1 1  0 0 1  0 1 1  0 1 1  1 1 0  1 0 1  0 0 0  1 0 0  1 1 1
```

Note that the left most zero introduced when 313365047 was converted from octal to binary was omitted from the input to the program.

3.4.2 The Code Selection Procedure

The above method of generating BCH codes produces the complete 2^k codewords. This is often more than the desired number of words for the classes, and hence a subset of these must be extracted to encode the C classes. Careful selection of this subset is necessary since arbitrary elimination of some codewords from the set may result in the remaining codeword matrix having identical or complimented columns. This presents a difficulty since the columns of the code, which correspond to the binary functions to be learned, must be substantially different from one another to avoid high correlations between the errors in the individual functions. In particular, as a result of arbitrary elimination of some codewords from the set, it can occur that two columns are complementary. Both ID3 and back propagation have the property that they behave identically when learning a binary function if the class labels on the training examples are interchanged (i.e., all members of class 1 are switched to class 0 and vice versa). The consequence of this is that the errors committed in these two columns will be identical and hence, the code will not be as effective. To eliminate this problem, we wrote several routines that employed a heuristic evaluation function to select codewords one at a time in a greedy fashion that attempts to eliminate the above difficulties. These routines also broke ties[26] in favour of the codeword that had the maximum distance with other

[26]in the score given by the evaluation function to each codeword considered.

Table 8. Performance (% correct on 1000-word test set) of various BCH codes. n is the codeword length and d is the minimum inter-word Hamming distance between all codewords.

BCH Code Data				% correct (1000-wod test set)					Decision Tree data (mean)	
PHONEME		STRESS		Level of Aggregation						
n	d	n	d	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
10	3	9	3	13.3	69.8	80.3	80.6	90.8	677.4	51.9
14	5	11	5	14.4	70.9	82.3	80.3	91.0	684.7	53.1
21	7	13	7	17.2	72.2	83.9	80.4	91.2	681.4	53.9
26	11	13	11	17.5	72.3	84.2	80.4	91.0	700.5	56.4
31	15	30	15	19.9	73.8	84.8	81.5	91.6	667.8	52.7
62	31	30	15	20.6	74.1	85.4	81.6	92.0	669.9	53.3
127	63	30	15	20.8	74.4	85.7	81.6	92.4	661.6	54.8

codewords that had been seleced so far. The procedure was heuristic and hence was not completely successful in every case. For this reason, once a good subset was selected, the column distances were checked to ensure that no complimentary or identical columns were present.

3.4.3 Results

Table 8 shows the results of training ID3 with distributed error-correcting output codes of varying lengths. Phonemes and stresses were encoded separately, although this turns out to be unimportant. Columns headed n show the length of the code, and columns headed d show the minimum Hamming distance between any two codewords.

The first thing to note is that the performance of even the simplest (19-bit) BCH code is superior to the 26-bit Sejnowski-Rosenberg code at the letter and word levels. Better still, performance improves monotonically as the length (and error-correcting power) of the code increases. The long codes perform much better than either the direct multiclass or Sejnowski-Rosenberg approaches at all levels of aggregation (e.g., 74.4% correct at the letter level versus 70.8% for direct multiclass).

Not surprisingly, the individual bits of these error-correcting codes are much more difficult to learn than the bits in the one-per-class approach or the Sejnowski-Rosenberg distributed code. Specifically, the average number of leaves in each tree in the error-correcting codes is roughly 665, whereas the one-per-class trees had only 35 leaves and the Sejnowski-Rosenberg trees had 270 leaves. Clearly distributed output codes do not produce results that are easy to understand!

The fact that performance continues to improve as the code gets longer suggests that we could obtain arbitrarily good performance if we used arbitrarily long codes. Indeed, Shannon's [Shannon48] theorem in information theory proves this under the assumption that the errors in the various bit positions are independent. Unfortunately, this is not the case in practice. In Section 3.4.8, we will discuss these and other limitations of the error-correcting output coding approach and suggest possible means of overcoming some of these limitations.

3.4.4 Error-correcting Codes and Small Training Sets

Given that the individual binary functions require much larger decision trees for the error-correcting codes than for the other methods, it is important to ask whether error-correcting codes can work well with smaller sample[27] sizes. It is well-established that small training samples cannot support very complex hypotheses. The more data that one has available, the more complex the hypotheses that can be statistically supported.

To address this question, Figure 5 shows learning curves for the distributed output code and for the 93-bit error-correcting code (63 phoneme bits with $d = 31$, 30 stress bits with $d = 15$). The data for this figure are shown in Table 9. At

[27]"Sample" and "training set" are used interchangeably by machine learning researchers.

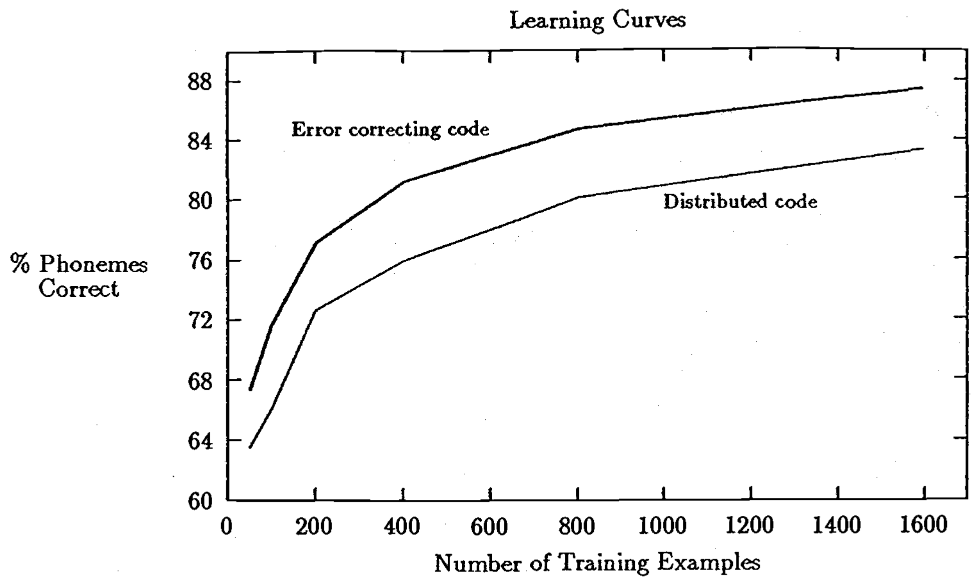

Figure 5. Learning curves for the distributed output code and for the 93-bit error-correcting code (63 phoneme bits with $d = 31$, 30 stress bits with $d = 15$).

all sample sizes, the performance of the error-correcting configuration is better than the Sejnowski-Rosenberg distributed code. Hence, even at small samples, error-correcting codes can be recommended.

3.4.5 Relationship to Other Methods of Improving Performance

In Chapter 4, we will explore several techniques for improving the overall performance of learning in this domain. These methods include extending the context of the current letter and altering the encoding employed for converting the context to binary representation.

An interesting question is whether the benefits provided by error-correcting output codes are independent of the benefits provided by these other improve-

Table 9. Learning curve data for the distributed output code and for the 93-bit error-correcting code (63 phoneme bits with $d = 31$, 30 stress bits with $d = 15$).

Words in training set (m)	Type of output code	% correct (1000-word test set)					Decision Tree data (mean)	
		Level of Aggregation						
		Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
$m = 50$	Std.	1.6	50.7	63.5	69.3	93.6	22.8	8.6
	Ecc.	3.0	54.7	67.4	71.6	84.2	59.7	23.9
$m = 100$	Std.	2.9	54.1	66.1	71.9	94.2	39.4	11.4
	Ecc.	6.7	59.2	71.7	74.6	86.2	103.7	29.5
$m = 200$	Std.	5.6	59.6	72.6	73.4	95.1	65.1	14.1
	Ecc.	11.0	64.7	77.1	76.8	88.4	175.6	34.3
$m = 400$	Std.	8.5	63.8	75.9	75.8	95.8	117.2	18.2
	Ecc.	15.6	69.4	81.2	79.3	90.2	315.1	42.9
$m = 800$	Std.	13.4	68.2	80.1	78.0	96.4	207.2	24.2
	Ecc.	19.8	73.5	84.7	81.3	91.7	554.6	49.4
$m = 1600$	Std.	15.5	71.9	83.3	80.3	96.9	366.0	28.6
	Ecc.	22.5	76.4	87.4	83.1	92.9	1004.2	59.3

ments. The answer, as shown in Table 43 (in the next chapter) is unambiguously "yes." The combination of error-correcting output codes with these additional input features provides the best performing text-to-speech system that we have studied.

3.4.6 Error-correcting Codes and Combined Phoneme/ Stress

In our previous discussion we presented results of applying separate error correcting codes for the phonemes and the stresses. We can of course apply the error-correcting coding technique equally well to the combined-ps case. In this scheme, we generate 127 codewords of length n bits and interword minimum Hamming distance d. The 126 phoneme/stress pairs (classes) observed in the training set are each assigned to one of these 127 codewords. The remaining codeword is assigned to a default or "others" class, in order to take care of other phoneme/stress combinations that were not present in the training set but *may* appear in the test set.[28]

Table 10 shows the results of applying the combined-ps method described above for three, relatively large, values of n and d. The performance figures for the $n = 63, d = 31$ combined-ps case (first line in the table) nearly matches that of the separate phoneme code $pn = 62, pd = 31$ with the stress code $sn = 30, sd = 15$ shown in Table 8—at the letter and word levels of aggregation. Further increase in the length (and interword Hamming distances) of the codes employed in the combined-ps case leads to further performance gains at the letter and word lev-

[28]This is required in our implementation because the evaluation routines are extensive and gather—among other things—bit-correctness statistics. For that, these routines compare the bit-vector computed from the learning algorithm with the *correct* bit-vector obtained by looking up the desired phoneme/stress pair in the codeword assignment table. So, when a phoneme/stress pair is given which is not in the table, it is assigned to this extra (unique) codeword for "others" and the bit-correctness evaluations can then proceed as normal.

Table 10. Performance of error-correcting codes applied to the combined phoneme/stress case.

Code Length n	Interword Distance d (min.)	% correct (1000-word test set) Level of Aggregation: Word	Letter	Phoneme	Stress	Bit (mean)	Decision Tree data (mean): Leaves	Depth
63	31	20.3	74.3	83.8	80.3	87.4	1000.0	68.7
127	63	22.3	75.5	85.2	81.5	87.8	1002.9	68.3
255	127	22.4	75.5	85.2	81.6	87.8	1014.4	68.8

els. The letter level performance goes up from 74.4% to 75.5%, and the word level performance jumps from 20.8% to 22.4%. This is not the case, however, for performance at the phoneme level. The best phoneme performance of 85.2% in the combined-ps case still does not match the best phoneme performance of 85.7% shown in Table 8 for the separate-ps case. The best stress level performance remains the same in both cases.

There are several other reasons for employing the combined-ps scheme instead of the separate-ps scheme—besides their good performance at higher levels of aggregation. One advantage is that we deal with one set of codewords instead of two, which allows us to characterize all the codewords with the same parameters (e.g. code length, interword Hamming distances, etc.). This proves to be useful when setting up experiments to determine the effects of varying these parameters. Another advantage is that we can overcome the limitations on the maximum "useful" length for the error-correcting codes that can be employed for the 6 stress symbols. (This and other limitations are discussed in Section 3.4.8).

3.4.7 Effect of Codeword Assignment

An interesting question that arises in the error-correcting coding approach is whether any thought should go into the process of assigning codewords to individual classes. On the one hand, distance considerations suggest that all such assignments should be equally good if the interword Hamming distance d is sufficiently high, since each BCH codeword is guaranteed to be at least a distance d from *all* others. On the other hand, each assignment of codewords to classes results in variations in the *columns* of the codeword matrix which correspond to the individual binary functions that must be learned by the learning algorithm. Hence it is possible that certain assignments might lead to functions that are easier to learn than other assignments.

To address the above question, we made 11 experimental runs on the 127 bit, $d = 63$ code employed in the previous section for the combined-ps case. In each run, the individual codewords were *randomly* assigned to corresponding phoneme/stress pairs (classes). The results, shown in Table 11, indicate that while some assignments are *slightly* better than others, the differences are not significant enough to warrant an elaborate method for the process of assigning codewords to individual classes.

3.4.8 Limitations of the Error-correcting Code Approach

In the error-correcting coding approach, the fact that performance continues to improve as the code gets longer may suggest to some that we could obtain arbitrarily good performance if we used arbitrarily long codes. This, however, is not the case. In practice, there are limitations to the error-correcting coding approach for the multiclass learning task. The discussion that follows covers three of these limitations and suggests possible means of overcoming them.

(1) Limitations due to bit-error correlations: Shannon's [Shannon48] theorem in information theory which shows that transmission errors can be arbitrarily reduced by employing longer and longer codes assumes that errors in the various bit positions are independent. However, because each of the bits is learned using the same body of training examples, it is clear that the errors are not independent. Table 12 shows the correlations between the errors committed at each bit position for the 21-bit, $d = 7$ *phoneme* code employed in the third row of Table 8. All of

Table 11. Effect of 11 random assignments of the 126 codewords to classes on the performance of the $d = 63$ error-correcting code (combined-ps).

Random Assignment Number	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
1	22.3	75.5	85.2	81.5	87.8	1002.9	68.3
2	22.2	75.8	85.1	81.7	88.1	998.2	68.8
3	21.8	75.5	85.0	81.8	87.9	995.7	66.9
4	22.2	75.7	85.0	81.5	88.0	996.6	66.9
5	22.2	75.5	85.2	81.6	87.9	994.7	68.1
6	21.4	75.5	85.3	81.4	88.0	996.5	69.4
7	21.0	75.6	85.3	81.4	88.0	998.6	68.8
8	22.4	75.6	85.3	81.5	88.0	999.3	67.7
9	23.1	75.7	85.6	81.7	88.0	999.8	68.9
10	23.1	75.8	85.4	81.6	88.1	991.1	67.9
11	21.5	75.5	85.4	81.5	88.0	998.3	69.7

the correlation coefficients are positive, and many of them are larger than 0.30. Hence, there must come a point of diminishing returns where further increases in code length do not improve performance.

One way of overcoming the limitations due to bit-error correlations—in domains where there is an abundance of training examples—is to learn each bit of the codeword from a different set of training examples. To see that this in fact works, we trained each of the 127 bits of the $d = 63$ code employed in Section 3.4.6 on a different subset of 1000 words selected randomly from a pool of 19,002 words (the complete NETtalk dictionary less the standard test set). Table 13 shows the dramatic effect of this reduction in bit-error correlation on the performance of error-correcting codes. Word-level performance jumped from 22.3% for the normal case (of all bits being trained on the same set of training examples) to 36.5% when each bit was trained on a different set of 1000 randomly selected words! Performance at all other levels of aggregation is significantly improved. For comparison purposes, we applied the same technique—of training each bit on a different set of 1000 randomly selected words—to two other coding schemes: The Sejnowski & Rosenburg distributed code employed in our base configuration and the 126-bit, weight-1 code employed in the combined-ps case for the one-per-class approach. The results, also shown in table 13, do not indicate a similar benifit for these other coding schemes. For both the Sejnowski & Rosenburg distributed code, and the one-per-class case, performance at all but the word levels of aggregation *dropped* as a result of training the individual bits on different training sets.

(2) Limitations due to the small number of classes: Since the number of bits in any *column* of the codeword matrix is C, there are at most 2^C distinct columns that can be used in the construction of the C-word code matrix. Furthermore, half of these are compliments of the other half, so must be eliminated from consideration leaving us with only 2^{C-1} possible distinct columns. Also the all 1 (or all 0) column must be removed. Hence the maximum length code that we can construct for C classes is of length $n = 2^{C-1} - 1$ bits. Requirements for having some minimum

Table 12. Pairwise bit error correlations for a 21-bit, $d = 7$ phoneme code. Values are multiplied by 100 then rounded for clarity.

	Correlation (×100) between errors in bit positions i (rows) and j (columns).																				
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	.	19	35	16	25	29	23	25	24	20	25	25	24	25	31	20	27	24	24	29	13
2	19	.	32	25	11	17	32	22	21	29	14	28	35	21	25	27	22	19	14	30	30
3	35	32	.	21	19	36	16	28	32	19	27	32	28	26	25	23	27	25	27	28	17
4	16	25	21	.	29	24	18	36	19	31	27	32	27	25	30	30	29	24	24	27	29
5	25	11	19	29	.	24	16	29	20	20	29	27	14	14	30	17	15	24	17	17	23
6	29	17	36	24	24	.	12	31	25	26	26	31	27	24	26	17	28	27	27	22	17
7	23	32	16	18	16	12	.	19	13	32	8	36	31	32	26	27	24	14	12	29	28
8	25	22	28	36	29	31	19	.	26	27	29	29	19	24	28	24	35	30	27	23	26
9	24	21	32	19	20	25	13	26	.	19	32	31	22	25	22	21	30	25	29	26	12
10	20	29	19	31	20	26	32	27	19	.	22	31	34	28	30	37	28	31	30	31	26
11	25	14	27	27	29	26	8	29	32	22	.	42	18	30	29	15	27	31	33	21	19
12	25	28	32	32	27	31	36	29	31	31	42	.	32	40	34	27	27	32	31	32	26
13	24	35	28	27	14	27	31	19	22	34	18	32	.	30	36	21	22	17	18	39	24
14	25	21	26	25	14	24	32	24	25	28	30	40	30	.	25	23	25	24	21	33	24
15	31	25	25	30	30	26	26	28	22	30	29	34	36	25	.	20	21	22	24	39	29
16	20	27	23	30	17	17	27	24	21	37	15	27	21	23	20	.	25	18	22	25	21
17	27	22	27	29	15	28	24	35	30	28	27	27	22	25	21	25	.	36	26	28	17
18	24	19	25	24	24	27	14	30	25	31	31	32	17	24	22	18	36	.	26	19	19
19	24	14	27	24	17	27	12	27	29	30	33	31	18	21	24	22	26	26	.	23	19
20	29	30	28	27	17	22	29	23	26	31	21	32	39	33	39	25	28	19	23	.	26
21	13	30	17	29	23	17	28	26	12	26	19	26	24	24	29	21	17	19	19	26	.

Table 13. Effect of training each bit of the output code on a different set of 1000 randomly selected words.

Output Coding Method	Number of trees	% correct (1000-word test set)				
		Level of Aggregation				
Training set		Word	Letter	Phoneme	Stress	Bit
(1) $d = 63$ Error-correcting code						
Separate for each bit	127	36.5	78.8	87.2	83.8	89.3
Common for all bits	127	22.3	75.5	85.2	81.5	87.6
(2) Std (Sejnowski) distributed code						
Separate for each bit	26	15.9	65.9	78.8	76.5	96.0
Common for all bits	26	12.5	69.6	81.3	79.2	96.3
(3) One-per-class (combined-ps)						
Separate for each bit	126	10.7	62.5	71.7	72.7	99.4
Common for all bits	126	8.7	66.7	76.4	74.5	99.5

Table 14. A set of 6 codewords generated by concatenating *all* possible 6-bit, weight-2 columns.

	Bit position														
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
3	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0
4	0	0	1	0	0	0	1	0	0	1	0	0	1	1	0
5	0	0	0	1	0	0	0	1	0	0	1	0	1	0	1
6	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1

distance between the codewords will further restrict the possible length for the number of bits in each codeword.

In our domain, the above does not impose any practical limitation on the length of error-correcting codes we can construct for the 54 phonemes. This is not the case, however, for the 6 stresses. Constructing a good[29] set of BCH error-correcting codes with a large number of bits and only 6 words in the set proved difficult, so other methods for designing such codes were investigated. These methods start from the columns of the codeword matrix. One such method is to concatenate *all* possible columns of the code having only two out of the six bits on (i.e. weight-2 columns). There are 15 such columns, (6 choose 2), and the code (shown in Table 14) has a uniform distance of 8 between every codeword and the other.

Trying the above procedure with weight-3 columns gives rise to longer codewords, but they are not useful for learning since half of the columns turn out to be compliments of the other half.

Another method for designing *the least* number of the *longest* codewords possible for a *good* error-correcting code is to list *all* possible columns of length $C - 1$, (where C is the number of codewords required) and then add one more codeword

[29]We consider an error-correcting codeword set to be "good" if the minimum distance between the codewords is nearly equal to half the number of bits in the code.

Table 15. Generating 6 error-correcting codewords of maximal length

i	The codeword set	Method of generation
1	01010101010101010101010101010101	repeat 01 (16 times)
2	00110011001100110011001100110011	repeat 0011 (8 times)
3	00001111000011110000111100001111	repeat 00001111 (4 times)
4	00000000111111110000000011111111	repeat 0000000011111111 (twice)
5	00000000000000001111111111111111	...
6	11111111111111111111111111111111	add all 1's codeword

Note that the last column of the code is useless and must be removed.

with all its bits set to 1 (or all its bits set to 0 if you so desire). For our 6 stresses, we have $2^5 = 32$ possible columns, creating a 32-bit code (shown in Table 15) with a uniform interword Hamming distance of 16. Note that the last column of the code (the all 1 column) must be removed resulting in a 31-bit code.

Still, the above methods can take you only so far. Another possible means of overcoming the limitations due to the small number of classes is to learn at higher levels of aggregation. Assigning a unique class number to each phoneme/stress combination (i.e. the combined-ps approach discussed in section 3.4.6) is an example of this technique. One can also go further and map *groups* of letters to *groups* of phoneme/stress pairs. This will certainly drive up the number of classes, but it may not be very useful in this domain due to the much smaller number of training examples that are available when learning at these larger grain sizes.

(3) Limitations due to bit accuracy: Even ignoring the problem of bit-error correlations, there is a minimum value for the bit accuracy needed for error correction to take over. To see this, let us assume that we can learn the individual bits in the code to have probability q of being correct. This of course is an over-simplification since different bits have different such probabilities in practice, but let us set aside this issue and consider q to be the average of these probabilities. If we have a t-error correcting code, then the probability that we will get the right answer after decoding using the BCH error-correcting code is the probability of making t or fewer errors out of n bits. This probability can be calculated (from the binomial distribution, assuming our errors are independent) as follows:

$$p(t \text{ or fewer errors}) = \sum_{i=0}^{t} \begin{pmatrix} n \\ i \end{pmatrix} (1-q)^i q^{n-i}$$

Table 16 shows the overall success rate calculated from the above formula for q ranging from 0.5 to 0.95 and values of n and t typical for BCH codes. There are several things to note here. First, the table shows that error-correcting codes do not become useful until q, the probability of individual bits being correct, approahes 0.8. Even then, it takes a 63-bit code that corrects up to 15 errors to raise the overall performance to a level beyond the value of q of 0.8. Once q crosses .8, error correction starts to take over, and once we get $q > .85$, it appears that we can drive the correctness arbitrarily high by using longer and longer codes. A second thing we can observe from the data shown is that there are short error-correcting codes that are more efficient than longer ones. Consider the columns of the table for values of q between 0.8 and 0.9. It is clear that the $n = 31, t = 7$ code is more efficient in terms of error-correcting capability than the next code: $n = 45, t = 8$. This can be explained by the fact that for a substantial increase in the length of the code from 31 to 45, we did not gain a comparable improvement in the error-correcting capacity of the code: t only increased from 7 to 8. In general, the most efficient BCH codes are the ones for which t is roughly equal to one fourth of the length n. Hence the codes $n = 31, t = 7$ and $n = 63, t = 15$ are considered "good" error-correcting codes.

There is no simple way of overcoming the limitations due to the low accuracy of learning the individual bits. Hence, the solution must be to beef up the bit-correctness to acceptable levels by including more and more training examples in the learning phase.

Table 16. Overall (global) success rate for error-correcting codes calculated assuming the binomial model

		$q =$ Average probability of bit correctness						
n	t	$q = 0.5$	$q = 0.6$	$q = 0.7$	$q = 0.8$	$q = 0.85$	$q = 0.9$	$q = 0.95$
6	0	0.0156	0.0467	0.118	0.262	0.377	0.531	0.735
15	2	0.0037	0.027	0.127	0.398	0.604	0.816	0.964
21	3	0.00074	0.011	0.086	0.370	0.611	0.848	0.981
31	7	0.0017	0.033	0.245	0.730	0.918	0.990	0.9999
45	8	0.00001	0.001	0.047	0.441	0.775	0.968	0.9997
51	11	0.00003	0.004	0.121	0.685	0.929	0.996	0.99999
63	15	0.00002	0.005	0.175	0.821	0.979	0.9996	1.00000

3.5 Random Codes

In addition to testing the performance of BCH codes, we also evaluated the performance of randomly-generated output codes. There were two reasons for considering random codes. First, randomly-generated codes are much easier to design than BCH codes. Second, it is interesting to ask whether biological systems might employ error-correcting codes as distributed representations. If so, some random way of generating such codes would be much more biologically plausible than using the highly specialized BCH method.

Consequently, we generated random codewords with lengths matching the BCH codes employed in Section 3.4.3. Table 17 shows the results when ID3 was trained using these random codewords (on our standard 1000-word training set and tested on our standard 1000-word test set).

The results show that randomly generated codes perform only slightly worse than the BCH codes of equal length. This raises an interesting question: What do these two kinds of codes have in common and why do they improve the performance of learning systems?

Table 17. Performance of random codes of equal length to the BCH codes employed in Table 8.

Code Length		% correct (1000-word test set)					Decision Tree data (mean)	
PHONEME	STRESS	Level of Aggregation						
pn	*sn*	Word	Letter	Phoneme	Stress	Bit (maen)	Leaves	Depth
10	9	12.3	69.0	79.9	79.7	91.5	598.9	48.5
14	11	14.2	70.8	82.0	80.5	92.1	597.8	49.8
21	13	16.5	71.8	83.2	80.6	92.0	624.9	48.9
26	13	18.2	72.4	84.3	80.8	92.0	619.3	45.9
31	30	19.1	72.7	84.2	80.9	92.6	587.9	47.4
62	30	20.5	73.9	85.3	81.5	91.8	681.8	56.3
127	30	20.1	73.4	85.4	81.1	92.3	656.1	55.6

We have four hypotheses to explain the reasons for the improved performance of these codes:

Hypothesis 1: Error-correcting Capability. Random codes do as well as BCH error-correcting codes, because generating bit strings randomly produces codewords with inter-word Hamming distances comparable with those of equal length BCH codes. Hence these random codes will have a comparable error-correcting capability.

Hypothesis 2: Code Length. Longer codes perform better than shorter codes. Hence random codes do as well as BCH codes of equal length.

Hypothesis 3: Balanced Rows. Random codes do as well as BCH error-correcting codes, because both methods produce codeword matrices whose rows (i.e. the codewords) are almost balanced; having nearly as many zeros as they have ones.

Hypothesis 4: Balanced Columns. Random codes do as well as BCH error-correcting codes because both methods produce codeword matrices whose *columns* are nearly balanced. It is commonly believed that binary learning algorithms work best when the frequencies of positive and negative examples are balanced.

These hypotheses are neither mutually exclusive nor exhaustive.

The following three subsections present the results of experiments we performed to test the validity of these hypotheses.

3.5.1 Test of Hypothesis 1 (Error-correcting Capability)

To test our first hypothesis, we measured the inter-word Hamming distances for all random codes used in Table 17 and compared them to the distances of the corresponding BCH codes. Tables 18 and 19 summarize these statistics. The average distances (for the BCH codes and the random codes of equal length) track each other very well as can be seen from the first table. This makes sense, since the Hamming distance between any two randomly-generated n-bit strings is binomially distributed with a mean of $n/2$. However the average distance between all codewords is not necessarilly a good indicator of the error correction capability of any code.[30] To achieve good error correction, the minimum inter-word Hamming distances among codewords is what counts. Based on this measure, Table 19 shows that the random codes we used did in fact have error correction capability even though it was not nearly as much as their corresponding BCH codes. This suggests that Hypothesis 1 is correct. Furthermore, we claim that the slight inferior performance of random codes to their BCH counter-parts could be attributed to the disparity between the error correction capability of these two sets of codes. Evidence to support this claim will come in the next section.

3.5.2 Test of Hypothesis 2 (Code Length)

One obvious way to test our second hypothesis would be to develop codes of different lengths, but having the same minimum inter-word Hamming distance, and test

[30]This will be apparent from the experiments of the next section.

Table 18. Average distances between the codes employed in Sections 3.4 and 3.5.

				Average Distances			
PHONEME		STRESS		PHONEME		STRESS	
n	*d*	*n*	*d*	BCH	Random	BCH	Random
10	3	9	3	5.1	5.0	4.5	5.3
14	5	11	5	7.1	7.0	5.8	5.2
21	7	13	7	10.7	10.4	7.5	5.7
26	11	13	11	13.2	13.1	12.8	11.5
31	15	30	15	15.7	15.4	15.7	13.7
62	31	30	15	31.5	31.0	15.7	15.9
127	63	30	15	63.5	63.5	15.7	15.5

Distances shown in columns 2 and 4 are the minimum for BCH codes only.

Table 19. Minimum distances between the codes employed in Sections 3.4 and 3.5

		Overall Mininum				Average of Minimums			
		PHONEME		Stresses		PHONEME		Stresses	
pn	*sn*	BCH	Random	BCH	Random	BCH	Random	BCH	Random
10	9	3	0	4	3	3.0	1.6	4.0	3.7
14	11	5	1	5	3	5.0	2.7	5.0	3.7
21	13	7	2	7	2	7.0	5.2	7.0	3.2
26	23	11	6	11	9	11.0	7.5	11.0	9.7
31	30	15	8	15	11	15.0	9.6	15.0	11.2
62	30	31	18	16	10	31.0	21.7	16.0	12.8
127	30	63	44	16	13	63.0	50.3	16.0	13.8

pn (*sn*) is the code length for phonemes (stresses).

their performance to see the effect of code length. Generating such codes turned out to be impractical for two reasons:

1. BCH codes: To generate BCH codes of various length and the same minimum inter-word Hamming distance, one must choose a reasonably short distance (e.g. 5) so that codes with length as small as 15 bits can be constructed. The problem then becomes generating the longer codes. Our routines when employed for generating longer codewords with that distance would produce an astronomical number of codewords. For example, for distance 5, they will generate 2^{51} codewords of length 63 and 2^{113} codewords of length 127. Attempts to modify the routine to produce only a subset of those codewords produced codes with undesirable properties for learning.

2. Random codes: Our attempts to generate random codes of various lengths and the same minimum inter-word Hamming distance failed. We found that the distances between the randomly generated codewords increase consistently with the length of the code.

As an alternative, we decided to generate several sets of codes of the same length but with varying inter-word Hamming distances. Again the same difficulties mentioned above were faced. So we decided to generate "biased" random codes, where a certain portion of the bits are randomly set to zero or one (flip a coin), while the rest of the bits are always set to zero. Which bits are to be randomized and which are to be biased toward zero were again determined by a random process determined by a parameter, *random-portion* in the range 0.0 to 1.0 (1.0 meaning completely random).

We employed the algorithm *biased-random* shown in Table 20 to generate 127 codewords (for ps-pairs in the training set—the combined-ps case), each of length 127 bits long. We varied *random-portion* from 0.01 to 1.00 in steps of 0.01 and generated 5 sets of codewords for each value of *random-portion*, a total of 500 sets. Each codeword set generated was passed through a routine that calculates

Table 20. Algorithm *biased-random* employed to generate "biased" random codes, where a certain portion of the bits are randomly set to zero or one (flip a coin), while the rest of the bits are always set to zero.

```
algorithm: biased-random
input: num-words, num-bits, random-portion
output: num-words codewords each of length num-bits
begin { biased-random }
  Avg-num-bits-to-randomize := num-bits × random-portion
  for $J = 1$ to num-words do
    begin
      {generate a codeword of length num-bits as follows: }
      for $i = 1$ to num-bits do
        begin
          set random-number = generate a random number
                              betweeen 0 and num-bits less 1
          if random-number < Avg-num-bits-to-randomize then
            set $bit_i$ randomly to zero or 1;
          else
            set $bit_i$ to zero.
        end;
      output the codeword;
    end;
end {biased-random}
```

the average, average of the minimums and the overall minimum distances. The results were a general steady increase of the distances from close to zero to 63.5 (average), 45 (overall minimum), as *random-portion* was increased towards 1.00.

From the 500 sets of codewords, we selected 6 good sets that have the following minimum distances: 2, 5, 9, 15, 31, and 45. The criterion for "goodness" was to have the difference between the minimum and average distances calculated for the

Table 21. Performance of "biased random codes" of equal length (127 bits) and various inter-word Hamming distances.

Inter-word distances			% correct (1000-word test set)					Decision Tree data (mean)	
			Level of Aggregation						
Min.	Avg-min	Avg	Word	Letter	Phoneme	Stress	Bit (maen)	Leaves	Depth
2	4.2	8.2	9.2	67.2	76.0	75.6	98.2	136.1	19.6
5	9.6	16.0	12.8	69.8	79.0	77.0	96.7	247.9	27.0
9	13.8	21.5	14.4	71.0	80.3	78.0	95.6	341.4	31.9
15	20.3	29.8	15.2	72.6	82.2	79.2	93.8	480.1	39.4
31	36.0	48.3	20.1	74.8	84.7	80.8	90.4	778.5	55.1
45	48.9	63.5	22.2	75.7	85.3	81.5	88.0	998.6	68.5

codeword set as low as possible, so that the codeword set was the best representative of the distance-label it was assigned to. It should be observed that these codewords are still random in every sense of the word, and hence they are expected to be good codes for the learning algorithm in the sense that no two columns are likely to be identical, compliments or excessively correlated.

The results shown in Table 21 unambiguously confirm our first hypothesis—that distances between the codewords (error-correcting capability) and not the codeword length—is the factor responsible for the performance improvement when using error-correcting or random codes. Hence Hypothesis 2 is incorrect.

The above results also suggest that a generate and test approach similar to the one used in this experiment is an attractive method for generating variable distance codewords sets, when the number of bits in each word is large, i.e. when the space of possible codewords is huge compared to the number of codewords required.

3.5.3 Tests of Hypothesis 3 and 4 (Balanced Rows or Columns)

To test these hypothesis, we ran experiments with two sets of codewords of length 127 bits each, and minimum Hamming distance 2, constructed in such away that:

(A) One had nearly as many zeros as there are ones in each row of the codeword matrix.

(B) The other had nearly as many zeros as there are ones in each column of the codeword matrix.

The first set was easy to construct. Starting with one nearly balanced 127-bit codeword, CW, we generated the 126 additional codewords needed to encode the classes for the combined-ps case. To generate the i^{th} codeword, we copied CW, then complimented the i^{th} bit.

To construct the second set of codewords, we started with one nearly balanced *column*, CC, of the codeword matrix and generated the i^{th} additional cloumn by copying CC, then complimenting the i^{th} bit or in some cases complimenting the i and the $i + 1$ bits (of the column vector). In this manner all the 127 required columns were generated. This construction produced codewords that had the following properties:

- Half of the codewords in the set had a very large distance (124 bits) with codewords in the other half.
- Codewords within each half were a distance 2 or 4 from one another.
- The overall average distance between all codewords was 64, but,
- The overall minimum distance was 2, as was the mean of the minimum distances.

Table 22. Performance of balanced output codes of length 127 and inter-word Hamming distance 2 compared to the one-per-class approach.

Type of output code employed	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (maen)	Leaves	Depth
(A) Balanced Rows	8.7	66.7	76.4	74.5	99.5	34.4	10.3
(B) Balanced Columns	8.6	66.7	75.6	75.1	94.2	410.4	34.5
(C) One-per-class	8.7	66.7	76.4	74.5	99.5	34.4	10.3

Table 22 compares the results of using these two output codes with the one-per-class approach discussed in Section 3.2. There are several points to notice here. First, the performance of (A) and (C) are exactly identical. A little scrutiny of the codewords used in both cases convinced us that this should be the case since the columns in both sets of codewords were identical, and ID3 does not care what the number of zeros (ones) are in the codeword. It cares about the number of zeros (ones) in the columns of the codeword matrix, since they determine which concepts are being learned. However, the concepts in the balanced columns case (B) were entirely different. They were much more difficult to learn as evidenced by the huge size of the trees built compared with the one-per-class case. Nevertheless, they did not perform any better.

We therefore conclude that both Hypothesis 3 and 4 are incorrect. Balanced codes do not help and balancing the class frequencies has no effect.

This once again strengthens our belief in Hypothesis 1, that error correction capability is the key to the improved performance of both BCH and random codes.

3.5.4 Random Codes: An Alternative to BCH?

The results of our experiments with random codes suggest that a reasonably good startegy for generating good error-correcting codes is to randomly generate and

Table 23. An "optimal" set of 6 codewords for the stresses and their inter-word Hamming distance statistics.

i	The codeword set	Inter-word distance between codewords *i* & *j* (below)						Row Summaries		
		1	2	3	4	5	6	Min.	Avg.	Max.
1	0101010101010101010101010101010	0	16	16	16	16	16	16	16.0	16
2	0011001100110011001100110011001	16	0	16	16	16	16	16	16.0	16
3	0000111100001111000011110000111	16	16	0	16	16	16	16	16.0	16
4	0000000011111111000000001111111	16	16	16	0	16	16	16	16.0	16
5	0000000000000000111111111111111	16	16	16	16	0	16	16	16.0	16
6	1111111111111111111111111111111	16	16	16	16	16	0	16	16.0	16

Overall Average Distance: 16.0

test—keeping the best sets. We can argue for this case by comparing the distance matrices for an "optimal" maximal length code of distance 16 (for the 6 words we need for the stresses) with corresponding codes generated randomly. The reason for choosing the codes for the stresses to perform this experiment is that the codewords are small and manageable (i.e. fit on a page). Experiments with larger sets of codewords and larger number of bits in each word gave similar results.

The "optimal" maximal length code is generated by concatenating *all* possible (distinct) columns of length 5, creating 5 codewords of length $2^5 = 32$, then adding a sixth codeword of all ones to prevent the resulting 6-bit columns from being compliments of each other, and finally, removing the all 1 column. This results in 6 sets of 31-bit codewords with a uniform interword Hamming distance of 16 as shown in Table 23.

Randomly generated codewords of length 31 bits gave an overall average distance ranging from 15.1 to 16.2 (in 10 runs). The best run had the distance statistics shown in Table 24 which is very close to the optimal!

Table 24. Inter-word Hamming distance statistics for the best of 10 randomly generated sets of 6 codewords each and length 31 bits.

	Inter-word distance between codewords i & j (below)						Row Summaries		
i	1	2	3	4	5	6	Min.	Avg.	Max.
1	0	15	19	13	17	13	13	15.4	19
2	15	0	18	14	16	14	14	15.4	18
3	19	18	0	20	20	14	14	18.2	20
4	13	14	20	0	16	16	13	15.8	20
5	17	16	20	16	0	18	16	17.4	20
6	13	14	14	16	18	0	13	15.0	18

Overall Average Distance: 16.2

Hence, we can conclude this section by asserting that random codes do almost as well as BCH codes, because they produce error-correcting codes. They are an attractive easy alternative to BCH codes.

3.6 Voting: Multiple Trees per Bit

One of the shortcomings of the one-per-class approach—as employed in our experiments—is that the decision trees built for the classes return—upon evaluation—binary decisions. $Tree_i$ decides whether or not a given example is a member of $class_i$ without giving any indication of the strength of the belief in its decision. One way to overcome this is to modify the tree building algorithm to store in each leaf of the tree a probability estimate reflecting the conditional probability that the example belongs to $class_i$—given the features tested along the path to that leaf. [Buntine90] and [Lucassen84] each give a possible method of estimating such conditional probabilities.[31] Another approach that we explored is to retain

[31]The straight forward approach of assuming that the required conditional probability is simply the fraction of examples in the training set that reach the leaf is not a good idea, since these

the binary nature of the decision trees but build x trees for each class. We can then resolve any conflicts between the x trees by taking the majority vote. This by itself, will increase the confidence in the individual class membership decisions, since a majority vote among the x trees is more likely to be correct than any single tree—other things being equal. We can further enhance the voting procedure by returning a real number v, reflecting the strength of our belief in the class membership decision—where v is simply the fraction of the x trees that voted "yes". This will allow us to resolve ties in favour of the class with the highest confidence level v.

In the previous discussion we argued the case for employing voting among several sets of trees by focussing on the one-per-class approach. However, there is no reason why voting should not be employed in general regardless of the particular scheme followed in coding the output. In particular, voting can be combined with distributed codes just as well, by building several trees for each bit of the code. The decoding phase that maps the output bits to the nearest legal (or observed) codeword remains the same if the results of the voting are returned as binary decisions reflecting the majority vote. However, should we decide to return the results of the voting as real numbers between 0 and 1 instead of binary digits, then the distance metric employed in the decoding phase must be slightly altered. In this case we used the following distance measure: $d(\vec{x},\vec{y}) = \sum_i |x_i - y_i|$. This reduces to the Hamming distance when $\vec{x}$ and $\vec{y}$ are Boolean vectors.

3.6.1 Constructing x Sets of Trees

ID3, unlike Backpropagation, is a deterministic algorithm. Given the same set of examples, ID3 always constructs the same decision tree for each output function. Hence, to force ID3 to build a different tree we must somehow give it a different set of examples every time. However, in order to have an objective evaluation of

fractions tend to be very small and the estimates too biased to the particular training set used to build the tree.

the effect of voting independent of the number of examples present in the training set, we have to constrain our experiments so that all the examples are drawn from the same pool: our standard 1000-word training set. There are a number of ways of meeting these constraints. We tested two:

1. Subdivide the training set, S_{all} into x subsets: S_1 to S_x. Build x sets of trees T_1 to T_x, where the training set used to build T_i is S_{all} concatenated with S_i. This method of including some of the examples twice in the training set will alter the distribution of the positive and negative examples for the output functions, and hence the ID3 selection criteria is likely to favour different features thereby producing different sets of trees. This method has the advantage that all the sets of trees see all the examples in the training set.

2. Subdivide the training set, S_{all} into x subsets: S_1 to S_x as above. Build x sets of trees T_1 to T_x, where the training set used to build T_i is:

$$\bigcup_{j=1, j\neq i}^{x} S_j$$

   That is tree set T_i is built using all the subsets but S_i as a training set. This method of building the trees is slightly faster since each tree sees a smaller set of examples. However, because of this very reason, the individual performance of each set of trees may suffer.

All our experiments for voting were done with the value of x, the number of trees built for each bit, being equal to or greater than 5. For these values of x, we found no noticeable difference between the performance of the collection of trees generated by either of the above methods. Hence we abandoned the first method and opted for the second more efficient alternative. Thus all the results reported in this section were obtained by using the second method described above for generating the x collection of trees.

3.6.2 Voting: Performance Results

Table 25 shows the results of applying the techniques described above for employing voting among x=5 trees (per bit) for three kinds of output encoding: the one-per-class approach, the standard distributed code of Sejnowski and Rosenburg, and a 93-bit BCH error-correcting code (a 63 bit, $d = 31$ code for phonemes, and a 30 bit, $d = 15$ code for stresses). Each set of trees was constructed from a different subset of 800 words out of the 1000-word training set. Table 26 shows similar results but with $x = 9$ trees voting (each built from 888-word subsets of the training set). For each kind of output code, these tables show three lines:

(A) The first row (marked 1x1000) shows the results for a single tree built using the full 1000-word training set (no voting).

(B) The next row (marked 5x800 or 9x888) shows the results for 5 (respectively 9) trees voting. The results of the vote are thresholded and returned as a binary decision reflecting the consensus among the majority of the trees.

(C) The third row (marked 5x800 float or 9x888 float) shows the results for 5 (respectively 9) trees voting as in (B), but with the results of the voting returned as a real number (the fraction of the trees that voted "yes" as explained in section 3.6).

There are several things to note in both tables. First, voting improves the performance in general regardless of the method of output encoding employed. Second, returning the results of voting as real (floating point) numbers is always better than thresholding it prematurely to binary. Third, the one-per-class approach benefits the most from voting. Fourth, the best performance of the one-per-class approach with voting does not quite match the performance of the error-correcting output code employed in these experiments—even if voting is not employed for the latter. Sixth, the peformance figures in Table 26 (9 trees voting) are generally superior to those shown in Table 25 (5 trees voting).

Table 25. Effect of employing voting among 5 sets of trees for different methods of encoding the output

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Configuration	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
(1) One-per-class							
(A1) – 1x1000	11.8	69.5	80.6	78.9	98.7	90.4	14.0
(B1) – 5x800	14.4	71.6	82.2	80.1	98.8	78.1	13.4
(C1) – 5x800 float	16.7	72.9	84.2	81.1	98.9	78.1	13.4
(2) (S & R) distributed code							
(A2) – 1x1000	12.5	69.6	81.3	79.2	96.3	269.9	29.3
(B2) – 5x800	14.3	71.2	82.5	80.3	96.7	232.3	26.6
(C2) – 5x800 float	14.7	71.7	83.3	80.8	96.8	232.3	26.6
(3) Ecc code: $pd = 31, sd = 15$							
(A3) – 1x1000	20.6	74.1	85.4	81.6	91.8	684.3	54.4
(B3) – 5x800	21.6	75.1	85.9	82.5	92.1	591.6	51.3
(C3) – 5x800 float	22.1	75.2	86.1	82.7	92.2	591.6	51.3

Table 26. Effect of employing voting among 9 sets of trees for different methods of encoding the output

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Configuration	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
(1) One-per-class							
(A1) – 1x1000	11.8	69.5	80.6	78.9	98.7	90.4	14.0
(B1) – 9x888	14.6	71.7	81.8	80.2	98.8	82.6	13.6
(C1) – 9x888 float	19.2	73.3	84.7	81.5	98.9	82.6	13.6
(2) (S & R) distributed code							
(A2) – 1x1000	12.5	69.6	81.3	79.2	96.3	269.9	29.3
(B2) – 9x888	14.6	71.7	82.6	80.6	96.7	246.4	27.8
(C2) – 9x888 float	16.1	72.3	83.7	81.0	96.8	246.4	27.8
(3) Ecc code: $pd = 31, sd = 15$							
(A3) – 1x1000	20.6	74.1	85.4	81.6	91.8	684.3	54.4
(B3) – 9x888	22.2	75.1	86.0	82.3	92.1	627.1	52.3
(C3) – 9x888 float	23.6	75.0	86.2	82.3	92.2	627.1	52.3

Let us now take a closer look at the difference between rows (B1 , C1) and (B3 , C3) in Table 26. Whether or not we prematurely threshold the result of the voting has a dramatic impact on the performance of the one-per-class approach. This is understandable, since thresholding the result of voting for each bit makes a final commitment based on information local to that bit, whereas delaying such final commitments until the results for all the bits are in and it is time to map to the nearest codeword is bound to be a wiser decision. Another way to look at these results is that they suggest that breaking ties between classes based on the strength of the vote is a much better strategy than breaking ties based on the frequency of the classes (or prior probability of each class). The effect of thresholding is much less pronounced when we employ error-correcting output codes. This shows the power of error correction, since even if premature decisions lead to errors in certain bits, the code will be able to correct for these errors if they are within its error correction capability.

There are two possible reasons for the superior performance shown when the number of trees voting is increased from 5 to 9:

1. Each tree in the 5-tree case is built from 800 words providing examples for the training set, while in the 9-tree case each tree is built from 888 words providing examples for the training set. This may account for the better performance of the latter since the individual trees perform better. If this is true, we should expect that 5-trees voting each built from 888 instead of 800 words should perform equally well to the 9-tree case.

2. As the number of trees voting is increased, performance generally improves. Performance is not sensitive to the number of examples used to build each tree.

To test the validity of the first explanation, we repeated our voting experiments with 5 trees voting but with each tree built from 888 words instead of 800 words.

Table 27. Effect of employing voting among 5 sets of trees each built with 888 words in the training set. These decision trees were ***different*** from the ones employed in the previous table.

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Configuration	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
(1) One-per-class							
(A1) – 1x1000	11.8	69.5	80.6	78.9	98.7	90.4	14.0
(B1) – 5x888	14.9	71.4	81.4	80.4	98.8	80.2	13.4
(C1) – 5x888 float	18.2	73.3	84.1	81.6	98.9	80.2	13.4
(2) (S & R) distributed code							
(A2) – 1x1000	12.5	69.6	81.3	79.2	96.3	269.9	29.3
(B2) – 5x888	13.4	71.5	82.3	80.8	96.6	239.6	27.4
(C2) – 5x888 float	15.8	72.6	83.3	81.5	96.8	239.6	27.4
(3) Ecc code: $pd = 31, sd = 15$							
(A3) – 1x1000	20.6	74.1	85.4	81.6	91.8	684.3	54.4
(B3) – 5x888	21.8	75.1	86.0	82.4	92.1	609.2	51.6
(C3) – 5x888 float	22.2	75.0	86.2	82.4	92.2	609.2	51.6

The results—shown in Table 27—are generally better than those in Table 25, which indicates that our first explanation is at least partially correct.

To test the validity of our second explanation—that the number of trees voting is the major factor for the improved performance, we combined the trees used in Tables 25 through 27 (19 trees total)[32] and measured the performance of those when the number of trees voting is gradually reduced from 19 down to 3. The results corresponding to row (C)—the float case (no thresholding)—are shown in Table 28. There seems to be little gain in having a larger number of trees contribute in the voting process. The best performance is seen when about 9 to 11 trees vote.

[32]The 5 sets of trees shown in Table 27 were numbered 1 to 5, the 5 sets of trees shown in Table 25 were numbered 6 to 10, and the 9 sets of trees shown in Table 26 were numbered 11 to 19. "x trees voting" in Table 28 means that trees 1 through x participated in the voting.

Beyond that there can be a slight degradation of performance in some cases.

3.7 Summary

In this chapter, we introduced a new approach to multiclass learning problems; one in which BCH error-correcting codes are employed as a distributed output representation. By comparing the performance of several previous approaches to the multiclass learning task to this new technique, we showed that error-correcting output codes provide an excellent method for applying binary learning algorithms to multiclass learning problems. In particular, we demonstrated that the standard ID3 algorithm coupled with error-correcting output codes outperforms the direct multiclass method, the one-per-class method, and a domain-specific distributed output code (the Sejnowski-Rosenberg code) in the NETtalk domain. We also discussed several limitations of the error-correcting code approach and suggested possible means of overcoming some of these limitations.

Following that, we evaluated the performance of randomly-generated output codes and showed that it is only slightly worse than that of BCH error correcting codes. We performed experiments to identify which properties of these codes lead to the improved performance. We demonstrated that the error-correcting capability is the key factor and that good error-correcting codes can be designed by generating random binary strings, instead of by using BCH methods. The results suggest that random codes provide an attractive alternative to BCH methods in practical applications.

Finally, we showed that the error-correcting code method is superior to the approach of generating multiple hypotheses and employing some form of voting among them.

Table 28. Effect of the number of trees voting on performance. Results of the voting are returned as a real number (fraction) without thresholding.

Output Coding Method	% correct (1000-word test set)				
	Level of Aggregation				
Number of trees	Word	Letter	Phoneme	Stress	Bit (mean)
(1) One-per-class					
19 trees voting	18.6	73.2	84.5	81.8	98.9
17 trees voting	18.7	73.2	84.6	81.7	98.9
15 trees voting	19.0	73.2	84.6	81.6	98.9
13 trees voting	18.9	73.3	84.6	81.7	98.9
11 trees voting	18.9	73.4	84.6	81.8	98.9
9 trees voting	18.8	73.5	84.6	81.8	98.9
7 trees voting	18.8	73.5	84.4	81.9	98.9
5 trees voting	18.2	73.3	84.1	81.6	98.9
3 trees voting	17.1	73.1	83.6	81.6	98.9
(2) Std (Sejnowski) distributed code					
19 trees voting	16.0	72.4	83.6	81.3	97.1
17 trees voting	16.2	72.5	83.7	81.3	97.1
15 trees voting	15.8	72.4	83.6	81.3	97.0
13 trees voting	15.9	72.4	83.7	81.4	97.0
11 trees voting	15.6	72.6	83.5	81.6	97.1
9 trees voting	15.7	72.7	83.6	81.7	97.1
7 trees voting	15.6	72.5	83.4	81.6	97.1
5 trees voting	15.8	72.6	83.3	81.5	97.0
3 trees voting	15.6	72.0	83.0	81.1	96.9
(3) Ecc code: $pd = 31, sd = 15$					
19 trees voting	22.9	75.0	86.2	82.4	92.4
17 trees voting	22.6	75.1	86.2	82.5	92.4
15 trees voting	22.6	75.1	86.2	82.6	92.4
13 trees voting	22.2	75.2	86.2	82.6	92.4
11 trees voting	22.1	75.1	86.2	82.6	92.4
9 trees voting	22.3	75.2	86.1	82.8	92.4
7 trees voting	22.2	75.2	86.2	82.7	92.4
5 trees voting	22.2	75.0	86.2	82.4	92.4
3 trees voting	22.7	75.1	86.1	82.7	92.4

Chapter 4

Input Techniques

In our base configuration (Section 1.5), we chose to represent the context for each letter to be pronounced by a 7-letter window: the letter itself, the 3 letters to its left and the 3 letters to its right. Following Sejnowski & Rosenberg, we also encoded each letter by a weight-1 (local), 29-bit code. In this chapter, we will explore several alternatives to both of these decisions.

Section 4.1 investigates the effects of incorporating the output bits accumulated so far as part of the context for the current letter. Following that, Section 4.1.4 shows that this input technique is not compatible with the error-correcting output coding method introduced in the previous chapter.

Section 4.2 explores the effect of including the *phonetic* outputs of the preceding letters as part of the context for the current letter. We refer to that as the *extended context.* In that section, we will also address the consequences of abandoning the weight-1 input encoding and explore an information theoretic approach to defining a "good" set of binary attributes to represent the extended context. After examining several binary representations of the extended context in Section 4.2, we conclude that for *all* levels of aggregation except the word level, utilizing the *left* phonetic context substantially *degrades* the generalization performance of the learning system. All is not lost however, since in Section 4.2.4 we establish that certain binary representations of the extended context coupled with processing the letters of the words in a right-to-left order (i.e. utilizing the *right* phonetic context) substantially improves the performance at *all* levels of aggregation in this domain.

The final sections in this chapter explore (briefly) the effects of combining the extended context (with the *right* phonetic context) with several other techniques previously introduced. Section 4.3 combines extended context with the sequential evaluation method (of Section 4.1.1) for incorporating the output bits of other functions as input features. Sections 4.4 and 4.5 combine extended context with the error-correcting output technique and the voting method introduced in the previous chapter. The results show that the benefits offered by the output techniques of the previous chapter are nearly orthogonal to the benefits provided by the improved input techniques developed here.

Finally, Section 4.6 explores the effect of enlarging the standard 7-letter window that we have been employing—so far—in our experiments. We conclude the chapter by showing the best performing learning system we have studied in this domain (trained on our standard 1000-word training set).

4.1 Output Bits of Other Functions as Input Features

One of the three hypotheses formulated in Section 2.6 to explain the differences between ID3 and Backpropagation was the sharing hypothesis. There, we suggested that the ability of BP to share hidden units among all of the f_i being learned *may* allow it to reduce the aggregation problem at the bit level and hence perform better. We therefore thought of several possible methods for the introduction of sharing among the concepts learned by ID3. One such method is learning each output function based not only on the primitive given input features, but also on the values of the rest of the output functions. In the Sejnowski & Rosenburg distributed output representation employed for the text-to-speech domain, to learn f_9, for example, we would use as input features values of f_1 through f_8 and f_{10} through f_{26} in addition to the original attributes: 1 through 203. A problem arises when this method is used in classifying novel instances, since for these, only the original attributes are available. In particular a situation may arise in which evaluating function i requires the knowledge of the output of function j and vice versa. Hence, some output values may not be computable directly. The next three sections present the results of two methods for utilizing the output bits of other functions in a manner that avoids this race condition during evaluation.

4.1.1 Sequential Evaluation

This method of sharing the output values (bits) of the other functions by ID3, avoids the race condition described above by building the trees in a certain order, say $tree_1$ to $tree_{26}$. We use only the input features to build $tree_1$, the original input features and the output of $tree_1$ to build $tree_2$, and in general the input features and the outputs of $tree_1$ through $tree_{i-1}$ to build the i^{th} tree. Evaluation of output values for new examples is then possible in the same order: $tree_1$ to $tree_{26}$, since the output of $tree_1$ to $tree_{i-1}$ will become available by the time we are ready to compute the output of the i^{th} tree.

The problem with this is two-fold :

1. Performance results are sensitive to the order in which the trees are built. In particular, if trees for some of the bits that are more difficult to learn than others happen to be the first ones to be built, then introducing their noisy output to the learning system when building trees for subsequent bits may not help the performance at all. In fact, there is a possibility that it may hurt the learning process.

2. Sharing is not fully present. The first tree built, for example, cannot use knowledge of any of the output functions in determining the value of the function it is computing.

The next two sections cover two methods for overcoming the above problems.

4.1.2 Optimal Sequential Evaluation

The solution to the first problem mentioned in the previous section is to measure the individual bit-error rates—for trees built without any sharing—on an alternate test set. Then, an "optimal" sequence for building the trees would be to build them in the order of the observed bit-error rates on the alternate test set (with the trees corresponding to the least bit-errors built first, of course). In our case, we measured these bit error rates by testing the performance of the trees built without sharing on a second test set of 1000-words randomly selected from the 18,002 words remaining in the dictionary after removing the standard training and test sets. The bit-error measurements were "raw", i.e. done without any kind of mapping to the nearest phoneme/stress vector. Table 29 shows these error (or rather the correctness) rates[33] and the corresponding "optimal" tree building sequence for the standard (Sejnowski) distributed code. Table 30 compares the performance results of employing this optimal sequence of building the trees to a straight sequence of building them: 1 to 26. The table also includes the performance of our base configuration, for ease of comparison.

There are several things to note. First, sharing of previously processed output bits results in a slight *reduction* in the accuracy at the bit level. This is understandable since now an error in an early bit in the sequence may lead to an error in a subsequent bit which may have selected that bit to test during the learning process. Second, there is some small benefit at the word and letter level even when using a straight (arbitrary) sequence for building the trees. The effects, however, are more pronounced when an optimal sequence for building the trees is employed. Third, the performance at the phoneme and stress levels move in opposite directions. To understand this, we need to recognize that bits 21 to 26 are the stress bits. In both sequences employed, these are among the very last trees to be built

[33]The 3 bits with a 100 % correctness in the table correspond to word boundary information needed for continuous text and do not change in our data set. These were kept for consistency with the codes used by other researchers.

Table 29. Bit correctness rates and tree building sequence for the standard (Sejnowski) distributed output code.

% Correct is the bit correctness on a 1000-word test set.					
% Correct	Bit Number	% Correct	Bit Number	% Correct	Bit Number
100.0	26	98.5	14	93.8	13
100.0	20	98.0	3	93.1	2
100.0	19	97.5	6	93.1	23
99.9	11	96.3	1	92.8	25
99.9	10	95.9	16	92.1	22
99.5	9	95.2	15	91.3	5
99.5	8	95.1	4	91.1	17
99.5	7	94.5	21	90.1	24
99.3	12	94.5	18		

Table 30. Effect of including the previously processed output bits, bit_1 to bit_{i-1}, as additional input features when learning the i^{th} bit.

Tree building sequence	% correct (1000-word data set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
Straight (1 to 26)	13.1	70.4	81.7	77.8	96.1	113.6	20.0
Optimal sequence	14.4	71.1	82.3	78.5	96.2	114.2	18.6
Base configuration	12.5	69.6	81.3	79.2	96.3	269.9	29.3

and hence are more susceptable to the propagation of previous bit-errors. Finally, the dramatic reduction in the average size of the decision trees indicates that the output bits included as part of the input feature-set proved to be very relevant to the learning process.

It should be pointed out that even though we determined an optimal sequence for building the trees in our experiments by testing the "raw" bit-level performance on an alternate test set, the availability of this additional test set is not a *necessary* requirement. An alternative method that may be employed in the absence of this alternate test set—in domains in which training examples are scarce—is *cross validation.* To carry out cross validation runs, the standard training set is split into two: a "subtraining" set and a "cross-validation" set. Learning is done from the subtraining set, and bit error (or correctness) rates are measured from evaluating the performance of the resulting trees on the cross-validation set.

4.1.3 Relaxation Methods

One drawback of the above sequential method of sharing the output bits of other functions is that sharing is not *fully* present. An alternative, more complicated approach, is to build the i^{th} tree using the original input attributes plus values of *all* the output functions except the i^{th} one. Classification can then be made through an iterative procedure described by the algorithm *relaxation* (shown in Table 31). This algorithm basically cycles through any race conditions that may arise (due to output bit interdependencies) until an "equilibrium state" is reached in which all output bits are consistent with one another. We applied this relaxation technique with *max-allowed-iterations* limited[34] to 10. Four initialization methods are mentioned in Table 31. We did not try initialization method (a)—treating $f_i(0)$ as missing—but performance with the other two initialization methods (b) and (c) (which are based on frequency information) were totally disappointing, even on the training set. Table 32 presents the performance results with initialization method (d): computing $f_i(0)$ from a set of trees learned from the original input features only *without* any sharing of output bits. The first row in the table presents the results of applying a modified relaxation procedure that uses the values of $f_i(k-1)$ in each iteration *after* mapping to the nearest phoneme/stress using the *observed* decoding strategy. The second row shows the results when no such decoding is

[34]In most cases the outputs either converged after 3 or 4 iterations or continued until the 10^{th} iteration without convergence.

Table 31. The relaxation algorithm for including all output bits (except bit_i) as additional input features when learning the i^{th} bit.

```
algorithm: Relaxation
Note:  $f_i(k)$ means the value of $f_i$ in the $k^{th}$ iteration.
       $f_i(0)$ means the initial value of $f_i$.
begin { Relaxation }
  Initialize: Set values of $f_i(0)$ as detailed further below.
  $k := 1$
  repeat
    outputs-converge = true
    {Optional Step: Adjust the values of $f_i(k-1)$ by mapping to the nearest phoneme/
         stress vector.}
    for $i := 1$ to max-output-function-number do
      begin
        compute $f_i(k)$ using the values of $f_i(k-1)$
        if $f_i(k) \neq f_i(k-1)$ then
          outputs-converge := false
      end
    $k := k + 1$
  until outputs-converge or $k$ = max-allowed-iterations
end {Relaxation}
```

Initialization Methods:

The initialization mentioned above can be performed in many ways e.g.

(a) Treat $f_i(0)$ as missing and use the techniques reported in the literature for handling missing values during classification using decision trees [QUINLAN86].

(b) Use the most common value for each f_i as $f_i(0)$. (This is 0 for all the 26 functions in the standard distributed output representation in the text-to-speech domain).

(c) In the text-to-speech domain, use the values of f_i corresponding to the most frequent phoneme and the most frequent stress symbols.

(d) Compute $f_i(0)$ from a set of trees learned from the original input features only *without* any sharing of output bits.

Table 32. Performance of the relaxation method for including all output bits (except bit_i) as additional input features when learning the i^{th} bit.

Decoding strategy (after each iteration)	% correct (1000-word data set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
Observed decoding	14.9	69.6	81.5	78.7	96.2	31.6	8.5
No decoding (except last stage)	13.6	68.9	80.0	78.6	96.0	31.6	8.5
Base configuration	12.5	69.6	81.3	79.2	96.3	269.9	29.3

done except at the last stage. These results indicate that—considering their added complexity, relaxation methods do not seem to do any better than the optimal sequential evaluation method discussed earlier. The only noticeable difference is that the size of the trees produced is significantly reduced. We must remember however, that a standard set of trees (obtained from our base configuration) was used for initialization purposes.

4.1.4 Shared Output Bits & Error-correcting Codes

In previous sections, we have shown that including the output bits of other functions as input features may lead to some (marginal) improvement in performance. This performance improvement is sensitive, however, to the order of building the trees—which in turn reflects the disparity in the accuracy of the individual output bits being shared. With this in mind, one may wonder about the wisdom of employing this technique with error-correcting output codes. The individual bits of these codes are generally more difficult to learn; hence, these bits have higher bit-error rates than bits in other output coding techniques. To answer this question, Table 33 compares the performance—with and without sharing of output bits—of BCH error-correcting codes of various lengths (and inter-word Hamming distances). Sharing of output bits was accomplished by the optimal sequential

Table 33. Performance of various BCH error-correcting codes with (first rows) and without (second rows) sharing of output bits. For the second rows, optimal sequential evaluation is employed as described in Section 4.1.2.

BCH Code Data				Output	% correct (1000-wod test set)					Decision Tree	
Phoneme		Stress		Bit	Level of Aggregation					data (mean)	
n	*d*	*n*	*d*	Sharing	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
10	3	9	3	YES	15.9	71.4	82.4	78.9	90.4	150.0	19.4
				NO	13.3	69.8	80.3	80.6	90.8	677.4	51.9
14	5	11	5	YES	14.7	70.1	81.6	78.8	90.1	121.6	16.3
				NO	14.4	70.9	82.3	80.3	91.0	684.7	53.1
21	7	13	7	YES	14.7	70.5	81.1	79.2	90.0	97.8	13.3
				NO	17.2	72.2	83.9	80.4	91.2	681.4	53.9
26	11	13	11	YES	15.8	70.5	81.4	78.7	89.7	64.4	8.9
				NO	17.5	72.3	84.2	80.4	91.0	700.5	56.4
31	15	30	15	YES	15.5	71.1	82.4	78.4	90.2	54.3	9.0
				NO	19.9	73.8	84.8	81.5	91.6	667.8	52.7

evaluation method described in Section 4.1.2. The results indicate that while including the output bits of other functions as input features may improve the performance of short error-correcting codes, it is generally *not recommended* to use this technique—even with *optimal* sequential evaluation—with error-correcting output codes.

4.2 Previous Commitments as Input Features: The Extended Context

This section explores the effects of including the *phonetic* context of the letters previously processed as part of the context for the current letter. With this *extended context*, the function that we must learn (assuming a 7-letter window) has the following form:

$$f(L_{-3}, L_{-2}, L_{-1}, L_0, L_{+1}, L_{+2}, L_{+3}, P_{-3}, S_{-3}, P_{-2}, S_{-2}, P_{-1}, S_{-1}) = P_0S_0$$

where

L_i is the letter at position i,
(0 being the position of the current letter,
negative indices indicating letters i positions to the left and
positive indices indicating letters i positions to the right),

P_i is the phoneme to which L_i is mapped and,

S_i is the stress to which L_i is mapped.

Including the phonetic context in the manner described above at learning time will not present problems during performance evaluation. Because the letters of the word are processed in sequence, when it comes time to pronounce the letter at position i in the word, all letters to the left of it have already been pronounced, so their phonemes and stresses are available as inputs to the classifier. However, the extended context *does* introduce several additional degrees of freedom into the learning problem. For one thing, the phonemes and stresses—which are now part of the context—must be converted to binary by choosing a suitable representation. This binary representation for the phonemes and stresses when used as input features may not *necessarily* be the same as the representation used for the output phonemes and stresses. Another degree of freedom is the possibility of processing the letters of the word starting at the end of the word and working towards the first letter. In this case, the *right* phonetic context is included as part of the extended context. In Section 4.2.4 we will show that this reverse processing of the letters in the word offers some improvement in the overall performance.

The following sub-sections will address the binary representation of the extended context. We will first cover the method followed by Mercer & Lucassen to define "good" (dense) binary representations for the letters, phonemes, and stresses that are included in the extended context. Section 4.2.3 will then compare the performance of the input representations developed by this method to that of other binary representations.

4.2.1 An Information Theoretic Approach for Defining Input Features

As discussed in Section 2.1.1, Lucassen & Mercer employed a 9-letter window and included the phonemes of the four letters to the left of the current letter in the context for the current letter. To convert this *extended context* to binary representation, Lucassen & Mercer employed a minimum set of features (or *binary questions*—in their words), defined through a procedure aimed at maximizing the mutual information between these features and the output phonemes. We have replicated their feature set selection procedure and employed it to define a set of binary features for each letter, phoneme and stress which is part of the extended context. In the following discussion of *our* implementation of this procedure, we will skip over much of the reasoning and motivation for the various steps involved. The interested reader is referred to [Lucassen83] for a more thorough discussion.

Questions About the Current Letter

Let us first examine the *output* of the feature set selection procedure when it is employed for defining a set of binary features for the current letter L. L can be regarded as a discrete random variable taking one of 26 values from the set $\mathcal{A}$ containing the 26 English letters[35]: A through Z. Let the probability mass function of L be $p(L)$. Let Y denote the pronunciation of the current letter. Y can also be regarded as a discrete random variable taking values from the set $\mathcal{Y}$ containing the 126 phoneme/stress pairs observed in the training set with an associated joint probability mass function $p(L, Y(L))$.

We want the feature selection procedure to define the single most informative binary question $Q(L)$ about a letter L: that which maximizes

[35]We removed the other three symbols "–", "_", and "." when defining features in this way, since these do not appear in our data sets.

$$MI(\mathrm{Q}(L)\ ;Y),$$

where MI stands for mutual information. Following [Lucassen83], we will call this question QL_1 (Question about the current letter, No. 1). By definition, this choice of QL_1 minimizes the conditional entropy, or the remaining uncertainty, of the pronunciation of a letter L given $\mathrm{QL}_1(L)$. The question QL_1 obtained by this method is shown below:

QL_1 is "true" for (A E G H I J L O U W Y)
QL_1 is "false" for (B C D F K M N P Q R S T V X Z)

After QL_1 is found, we determine the most informative *second* question about the letter L — QL_2: that which maximizes

$$MI(\mathrm{QL}_2(L)\ ;Y \mid \mathrm{QL}_1(L)).$$

The question QL_2 obtained by this method is shown below:

QL_2 is "true" for (A B C D E I K O Q S T Y Z)
QL_2 is "false" for (F G H J L M N P R U V W X)

Together, $\mathrm{QL}_1(L)$ and $\mathrm{QL}_2(L)$ divide the alphabet $\mathcal{A}$ into four groups, or four *partitions*:

1: (G H J L U W)
2: (A E I O Y)
3: (F M N P R V X)
4: (B C D K Q S T Z)

We continue to generate new questions, $\mathrm{QL}_i(L)$, while maximizing

$$MI(\mathrm{QL}_i(L)\ ;Y \mid \mathrm{QL}_1(L), \ldots, \mathrm{QL}_{i-1}(L))$$

This process is continued until every letter L is completely specified by the answers to each of the questions about it, i.e. each of the *partitions* defined by the combination of answers to the questions

$$\mathrm{QL}_1(L),\ \mathrm{QL}_2(L),\ \ldots,\ \mathrm{QL}_n(L)$$

contains a single letter. The full set of questions (features) defined in this manner are shown in Table 83 of Appendix D.

The procedure

We will now consider the algorithm followed to define new questions, $\mathrm{QL}_i(L)$, while maximizing

$$MI(\mathrm{QL}_i(L)\ ;\ Y \mid \mathrm{QL}_1(L),\ \ldots,\ \mathrm{QL}_{i-1}(L))$$

Recall that $\mathrm{QL}_1(L)$, ... , $\mathrm{QL}_{i-1}(L)$ taken together divide the alphabet $\mathcal{A}$ into a set of partitions which we will call *partitions-so-far*. Our goal now is to come up with a new clustering of the alphabet into two sets representing a new binary question $\mathrm{QL}_i(L)$. This is achieved by calling the recursive ***define-feature*** algorithm shown in Table 34 with ***clusters-list*** initially having as many clusters as there are letters in the alphabet (i.e. each letter in a separate cluster). The algorithm merges clusters in a greedy fashion with the objective of maximizing the conditional mutual information between the resulting clusters and Y given *partitions-so-far*. The procedure terminates when there are only two clusters remaining representing answers to the binary question $\mathrm{QL}_i(L)$.

The objective function

Instead of maximizing the ***additional MI*** between the clusters, X, and the output, Y, after knowing *partitions-so-far*[36], we chose to maximize the ***total MI*** after knowing both X ***and*** *partitions-so-far* taken jointly:

$$MI(Y; X, \textit{partitions-so-far})$$

These differ from one another by a constant because of the additive property of MI, so the results of maximizing either objective function should be identical.

[36]This was done in [Lucassen83].

The formula we used was[37]

$$MI_{total} = \sum_{z \in partitions} \sum_{x \in X} \sum_{y \in Y} P(y,x,z) \log \frac{P(y,x,z)}{P(y)P(x,z)},$$

where the various probabilities are calculated from *joint-prob-table*—also input to the *define-feature* algorithm—which gives estimates for the joint probability between each letter and output phoneme/stress as calculated from the 1000-word training set.

Swapping and Moving Elements

In an attempt to further improve the binary partition obtained by the *define-feature* algorithm, the clustering program is followed by a procedure that considers all possible ways of moving a single letter from one cluster to another, as well as all possible ways of swapping two letters that are in opposite clusters. The swap or move that improves the value of the objective function by the greatest amount is performed. This process of swapping and/or moving continues until no single swap or move can further increase the value of the objective function.

Questions About the Letter to the Left

In a manner similar to the one described above, a set of binary features is defined to encode the letter to the left of the current letter (LL). The objective function in this case is slightly more complicated:

$$MI_{total} = \sum_{l \in L} \sum_{z \in partitions} \sum_{x \in X} \sum_{y \in Y} P(l,y,x,z) \log \frac{P(l,y,x,z)}{P(y)P(l,x,z)}$$

and requires the joint probability distribution

$$p(L, LL, Y(L))$$

where L is the current letter, LL is the letter to its left, and $Y(L)$ is its pronunciation. The resulting binary features—shown in Table 83 of Appendix D—are used to represent *all* letters to the left of the current letter.

[37]See page 126 of [Abramson63] for additional details.

Table 34. Algorithm: *define-feature.* **Input:** *clusters-list, partitions-so-far, joint-prob-table.* **Output:** *clusters-list* after reduction to only two clusters.

```
begin { define-feature }
  if clusters-list contains only two clusters then
    Return clusters-list
  else
    begin
      Initialize candidates-to-merge to null
      Initialize best-MI to some negative number
      {Consider merging every possible pair in clusters-list }
      for $i = 1$ to number of clusters in clusters-list less 1 do
        for $j = i + 1$ to number of clusters in clusters-list do
          begin
            Let $MI =$ The conditional mutual information between the clusters and $Y$ given
                      partitions-so-far when the number of clusters in clusters-list
                      is reduced by 1 as a result of merging $clusters_{ij}$.
            if $MI >$ best-MI then
              begin
                set best-MI to MI
                set candidates-to-merge to $clusters_{ij}$
              end
            else if $MI =$ best-MI then
                add $clusters_{ij}$ to candidates-to-merge
          end;
      if there is more than one pair of clusters in candidates-to-merge then
        resolve ties in favour of merging the cluster pair whose merging results in
          maximizing the unconditional mutual information between $X$ and $Y$.
          Hence reduce candidates-to-merge to one pair of clusters.
      Let new-clusters-list be the clusters remaining after merging the pair of clusters in
          candidates-to-merge
      {Call the algorithm recursively with the new-clusters-list. }
      define-feature (new-clusters-list, partitions-so-far, joint-prob-table)
    end
end { define-feature }
```

Questions About the Letter to the Right

Similarly, a set of binary features is defined to encode the letter to the right of the current letter (LR). The joint probability distribution used in this case is:

$$p(L, LR, Y(L))$$

where L is the current letter, LR is the letter to its right, and $Y(L)$ is its pronunciation. The resulting binary features—shown in Table 83 of Appendix D—is used to represent *all* the letters to the right of the current letter.

Questions About the Phonemes to the Left

The same clustering algorithm described above was also applied to the 54 phonemes constituting the phoneme alphabet. This time, the objective was to identify good binary questions about the phoneme, LP, corresponding to the letter that immediately precedes the current letter. The objective function, the mutual information between LP and the pronunciation of the current letter, $Y(L)$, was computed from the joint probability distribution:

$$p(LP, Y(L))$$

The resulting binary features—shown in Table 84 of Appendix D—are used to represent *all* of the phonemes corresponding to the letters preceding the current letter.

Questions About the Stresses to the Left

Finally, the clustering algorithm described above was applied to the 6 stress symbols with the objective of identifying good binary questions about the stress, LS, corresponding to the letter that immediately precedes the current letter. The objective function, the mutual information between LS and the pronunciation of the current letter, $Y(L)$, was computed from the joint probability distribution

$$p(LS, Y(L)).$$

The resulting binary features—shown in Table 85 of Appendix D—are used to represent *all* of the stresses corresponding to the letters preceding the current letter.

4.2.2 Performance Results

Table 35 shows the performance results when the codes developed by the information theoretic approach described in the previous sections are employed with three output techniques: The distributed code method with the standard (Sejnowski & Rosenburg) output code, the multiclass approach, and the one-per-class approach. Phonemes and stresses are considered separately in these experiments (the separate-ps approach). To separate the effects of utilizing the left phonetic context from the effect of the *different* representations employed to convert the context to binary, we have evaluated the performance of each method with both the standard 7-letter context, "StdContext", and with the *extended context*, "ExtContext", which includes the phonemes and stresses corresponding to the 3 letters to the left, in addition to the standard 7-letter context. These results show that, for *all* levels of aggregation except the word level, utilizing the left phonetic context substantially *degrades* the generalization performance of the learning system. More importantly, the performance at the stress, phoneme and letter levels—with "StdContext" or "ExtContext"—is worse than that of our base configuration. These observations are valid for all three output techniques compared. Only the performance at the word level shows an improvement with ExtContext over StdContext. Even then, the improvement is barely enough to match the word level performance of our base configuration. Despite the performance results, features of the phonetic context seem to be significant for learning as evidenced by the reduced size of the trees built with "ExtContext" compared to the ones built with "StdContext".

Let us now focus our attention *only* on the impact of the representation employed for converting the *standard* context to binary. Table 36 reproduces the lines marked with "StdContext" from Table 35 together with the performance of

Table 35. Impact of the input encoding developed by Mercer & Lucassen's information theoretic approach for defining input features.

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Context employed	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
Std (S & R) distributed code:							
StdContext	10.9	67.5	79.0	78.3	96.0	315.1	16.7
ExtContext	12.7	61.5	74.4	73.7	95.2	245.0	15.6
Multiclass (Separate-ps):							
StdContext	10.0	66.4	80.0	77.1	N/A	1446.0	20.4
ExtContext	11.8	61.9	75.8	75.2	N/A	1118.4	23.0
One-per-class (Separate-ps):							
StdContext	11.6	68.2	78.9	78.5	98.6	106.1	13.1
ExtContext	12.6	61.9	74.1	74.4	98.2	79.4	12.3
Base configuration	12.5	69.6	81.3	79.2	96.3	269.9	29.3

StdContext means the standard 7-letter window context.

ExtContext = StdContext + the phonetic context of the previous 3-letters.

Table 36. Comparing the input encoding developed by Mercer & Lucassen's approach for the 7-letter window *only* (without the phonetic context) with the weight-1 code employed in our base representation.

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Context employed	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
Std (S & R) distributed code:							
Dense (M & L) code	10.9	67.5	79.0	78.3	96.0	315.1	16.7
Std (weight-1) code	12.5	69.6	81.3	79.2	96.3	269.9	29.3
Multiclass (Separate-ps):							
Dense (M & L) code	10.0	66.4	80.0	77.1	N/A	1446.0	20.4
Std (weight-1) code	13.0	69.7	82.4	79.5	N/A	1305.0	48.0
One-per-class (Separate-ps):							
Dense (M & L) code	11.6	68.2	78.9	78.5	98.6	106.1	13.1
Std (weight-1) code	11.8	69.5	80.6	78.9	98.7	90.4	14.0

the same three output methods when the standard 29-bit, weight-1 (local) code —employed in our base configuration—is used to convert the 7-letter window to binary. The results in this case are even more conclusive. At *all* levels of aggregation and for all of the three output methods considered, the binary codes obtained by the information theoretic approach of Mercer & Lucassen (M & L) for the letters perform worse than the simple 29-bit local encoding employed in our base configuration. The only *favourable* effect of the (M & L) codes is that they tend to produce trees that are "shallower" on the average. They require fewer tests to classify a new example. The trees, however, generally contain a larger number of leaves.

4.2.3 Other Binary Representations for the Extended Context

The superior performance of the local encoding for converting the letters of the context to binary—over the (M & L) code—suggested that a search for better representations of the extended context might be fruitful. We decided to embark on this search for one output technique: The distributed (Sejnowski & Rosenburg) output code employed in our base configuration.

Table 37 shows the performance figures when several schemes for converting the extended context to binary representation are employed. An explanation of each of the schemes shown in the table follows:

ExtContext(M & L): This scheme employs the codes (or the features) developed by the Mercer & Lucassen method described earlier. Recall that each part of the extended context is converted separately to binary as follows:

- The current letter is converted using the feature set shown in the second column of Table 83 of Appendix D.
- The three letters to its left are converted using the feature set shown in first column of Table 83 of Appendix D.
- The three letters to its right are converted using the feature set shown in third column of Table 83 of Appendix D.
- The three phonemes corresponding to the letters to the left of the current letter are converted using the feature set shown in Table 84 of Appendix D.
- The three stresses corresponding to the letters to the left of the current letter are converted using the feature set shown in Table 85 of Appendix D.

ExtContext(1): In this scheme, all the letters in the context are converted to binary using a 29-bit weight-1 (local) code as employed in our base configuration.

The left phonemes and left stresses which are part of the extended context are converted to binary using *the same* code as that used for the output phonemes and stresses. (In this case 21-bits for each phoneme and 5 bits for each stress).

ExtContext(2): Here, weight-1 codes are employed throughout for *all* the elements of the extended context:

- All the letters in the context are converted to binary using the standard 29-bit weight-1 (local) code as usual.
- The three left phonemes are converted to binary by employing a 54-bit wieght-1 code corresponding to the 54 phonemes. The codeword for $phoneme_i$ will have all, but the i^{th} bit, set to **0**.
- The three left stresses are converted to binary by employing a 6-bit wieght-1 code corresponding to the 6 stresses. The codeword for $stress_i$ will have all, but the i^{th} bit, set to **0**.

ExtContext(3): Combines the representation of ExtContext(L & M) with ExtContext(2). In this scheme, each element of the extended context is converted to binary using a code obtained by concatenating the codes employed in ExtContext(L & M) with those employed in ExtContext(2) for that element of the context.

ExtContext(4): As in ExtContext(2). Additionally, a **30**th bit is added to the code for each letter or input symbol:

Bit$_{30}$ is "false" for (A E I O U Y - _ .)
Bit$_{30}$ is "true" for all other letters.

This bit should make it easier for the learning system to distinguish consonants from vowels or semi-vowels.

ExtContext(5): As in ExtContext(2). Additionally, a 7^{th} bit is added to the code for each stress symbol in the extended context:

Bit_7 is "false" for (< > -)

Bit_7 is "true" for (1 2 0)

ExtContext(6): Combines ExtContext(2) with the additional bits employed in both ExtContext(4) & ExtContext(5).

ExtContext(7): As in ExtContext(6). Additionally:

- One bit is added to the code for each phoneme in the extended context, to produce a 55-bit code:

  Bit_{55} is "true" for phonemes (a e i o u A O W Y)

  Bit_{55} is "false" for all other phonemes.

  This bit encodes whether the vowel (phoneme) is "tense" or "lax" ("non-tense"). Linguists frequently make this distinction in their rules for English stress.

- A 31st bit is added to the code for each letter or input symbol:

  Bit_{31} is "true" for (E I O U)

  Bit_{31} is "false" for all other letters (or input symbols).

  This bit distinguishes the "non-low vowels" from all others.

- An 8th bit is added to the code for each stress symbol in the extended context:

  Bit_8 is "true" for (1 2)

  Bit_8 is "false" for (< > - 0) This bit indicates whether the letter is stressed or not.

The results shown in Table 37 reveal several interesting points (besides the obvious fact that input representation has a profound effect on the performance of learning systems):

Table 37. Impact of alternative schemes for converting the extended context to binary represe-nation. See text for explanation.

Input	% correct (1000-word test set)					Decision Tree	
	Level of Aggregation					data (mean)	
Representation	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
ExtContext(M & L)	12.7	61.5	74.4	73.7	95.1	245.0	15.6
ExtContext(1)	15.1	66.5	79.5	76.3	96.0	213.7	27.6
ExtContext(2)	15.2	66.2	78.5	76.3	95.8	225.0	27.1
ExtContext(3)	16.6	66.8	79.1	77.3	96.0	192.6	19.3
ExtContext(4)	17.0	67.5	80.5	77.4	96.2	209.4	22.5
ExtContext(5)	14.5	66.0	78.7	76.2	95.8	222.8	26.9
ExtContext(6)	17.1	68.1	80.8	77.7	96.2	209.8	23.6
ExtContext(7)	17.0	67.9	81.2	77.1	96.2	207.2	24.2
Base Configuration	12.5	69.6	81.3	79.2	96.3	269.9	29.3

1. Including the *left* phonetic context *degrades* the performance at the bit, stress, phoneme and letter levels for all the schemes employed to convert the extended context to binary. However, it consistently improves the word level performance.

2. Including the left phonetic context results invariably in smaller trees compared to the base configuration.

3. ExtContext(7) nearly compensates for the negative effect of the left phonetic context as far as the phoneme performance is concerned. The 81.2% phoneme correctness achieved nearly matches that of the base configuration. This is encouraging since it indicates that by a suitable scheme for the binary representation of the extended context, we can have an improved word-level performance without sacrificing the performance at the phoneme level.

4. ExtContext(6) gives the best stress performance of 77.7%. This, however,

still falls short of the 79.2% correctness achieved without an extended context.

4.2.4 Forward vs. Backward Processing

Another degree of freedom introduced by employing an extended context is whether to process the letters of the words first-to-last (forward) and hence to include the pronunciations of the letters to the *left* of the current letter or vice versa: process the letters of the words last-to-first (backward) and hence include the pronunciations of the letters to the *right* of the current letter in the extended context.

This point was (briefly) investigated in the work of Mercer & Lucassen. The result of their investigation can best be described by the following quote from [Lucassen83], page 11:

> The [system] operates on a word from left to right, predicting phones in left-to-right order. This decision was made after preliminary testing failed to indicate any advantage to either direction. The left-to-right direction was chosen to simplify the interface to the linguistic decoder, as well as because of its intuitive appeal.

Our findings were entirely different. Table 38 reproduces Table 37 and, in addition, shows the effects of processing the letters of the words in a last-to-first order for each of the input representations investigated. The results unambiguously indicate that processing the letters of the word backward—in a right-to-left manner—is far superior to the "normal" left-to-right processing. Performance improvement is observed at *all* levels of aggregation and for *all* the schemes employed to convert the extended context to binary. Furthermore, the decision trees produced as a result of this backward processing are smaller than their counterparts produced with normal processing. This indicates that the *right* phonetic context is more relevant than the left phonetic context if we want to learn rules for the pronun-

Table 38. Impact of processing the letters of the words *backward* (in right-to-left order) on the performance of the input represenations employed in Table 37

Input Representation	Order of letter processing	% correct (1000-word test set) Level of Aggregation: Word	Letter	Phoneme	Stress	Bit	Decision Tree data (mean): Leaves	Depth
ExtContext (M & L)	Forward	12.7	61.5	74.4	73.7	95.1	245.0	15.6
	Backward	15.3	69.7	80.1	80.4	96.2	226.5	15.4
ExtContext(1)	Forward	15.1	66.5	79.5	76.3	96.0	213.7	27.6
	Backward	18.4	70.5	81.8	79.7	96.4	200.3	20.9
ExtContext(2)	Forward	15.2	66.2	78.5	76.3	95.8	225.0	27.1
	Backward	19.7	71.5	82.2	80.7	96.4	198.1	23.8
ExtContext(3)	Forward	16.6	66.8	79.1	77.3	96.0	192.6	19.3
	Backward	19.7	73.1	82.7	82.4	96.6	171.2	18.8
ExtContext(4)	Forward	17.0	67.5	80.5	77.4	96.2	209.4	22.5
	Backward	23.8	73.9	83.7	82.3	96.7	183.2	22.5
ExtContext(5)	Forward	14.5	66.0	78.7	76.2	95.8	222.8	26.9
	Backward	20.3	71.7	82.7	80.6	96.5	213.0	23.6
ExtContext(6)	Forward	17.1	68.1	80.8	77.7	96.2	209.8	23.6
	Backward	23.9	73.9	83.7	82.2	96.8	182.0	22.3
ExtContext(7)	Forward	17.0	67.9	81.2	77.1	96.2	207.2	24.2
	Backward	24.4	74.2	83.9	82.6	96.9	180.1	22.0
Base Configuration		12.5	69.6	81.3	79.2	96.3	269.9	29.3

ciation of the current letter. This finding is consistent with the fact that several researchers working on predicting English word-stress have come up with rules that refer to certain properties of the syllables of the words taken in a right-to-left order. Examples of such rules are (see [Halle71] for a complete list):

> If the last vowel is nontense, primary stress goes on the antepenultimate vowel when the penultimate vowel is nontense and followed by no more than one consonant.
>
> If the last vowel is nontense, primary stress goes on the penult when the penult is itself tense or when it is followed by two (or more) consonants.
>
> If the last vowel is tense, it bears primary stress.

It should be noted that the combination of extended context, backward processing of the letters of the words and improved binary represenation for the extended context have now raised the performance at *all* levels of aggregation when compared to the base configuration. ExtContext(7), with words processed ***backwards***, (or "ExtContext(7-BW)" for short) emerged as a winner[38] of this "contest", and hence, will be adopted in reporting the results of most of our experiments with the extended context throughout the remainder of this thesis. However, we must first justify this decision by showing that "ExtContext(7-BW)" is also a good choice when other output techniques are employed. This is done in Table 39 for the multiclass (separate-ps) approach and Table 40 for the one-per-class (separate-ps) approach. The same general trends seen for the distribited code case are also apparent in these two tables. In particular:

[38]The natural choices are "ExtContext(2-BW)" or "ExtContext(7-BW)". "ExtContext(2-BW)" is to be prefered as a domain independent method for input encoding while "ExtContext(7-BW)" is a performance booster for this domain. There are various reasons for excluding some of the others. "ExtContext(1-BW)", for example, is not general enough—being dependent on the particular *output* codes employed. It will not be applicable, for instance, to the multiclass approach. "ExtContext(3-BW)" requires the feature set employed in ExtContext(M & L) which requires an elaborate procedure to compute.

1. ExtContext(M & L) lags way behind the other input representations.
2. Processing the letters of the words backwards in a right-to-left fashion offers a substantial improvement over normal left-to-right processing.
3. ExtContext(7-BW) results in the best stress, letter, and word level performance in all cases.
4. ExtContext(2-BW) seems to result in the best phoneme level performance of 83.7% for the multiclass (separate-ps) case, compared with 82.8% for ExtContext(7-BW). However, the improved stress level performance of ExtContext(7-BW)—83.0% vs. 80.1%—more than compensates for that at the letter and word levels.

Hence our decision to adopt "ExtContext(7-BW)" as *the* input representation for subsequent experiments is ***experimentally*** justified. Nevertheless, we will show wherever feasible performance figures for the extended context with both "ExtContext(2-BW)" and "ExtContext(7-BW)" representations.

4.3 Sharing of Output Bits & Extended Context

In Section 4.1, we showed that including the output bits of other functions as input features may lead to some (marginal) improvement in performance when our standard input representation is employed. To evaluate the effectiveness of this technique when combined with extended context, better binary representations and right-to-left (BW) processing of the letters of the word, we combined the sequential evaluation method described in Section 4.1.1 with "ExtContext(2-BW)" and "ExtContext(7-BW)". Tables 41 and 42 show the results when:

- A straight sequence of building the decision trees is employed,
- The decision trees are built in an optimal sequence obtained by the method described in Section 4.1.2, and

Table 39. Impact of several binary representations of the extended context on the performance of the multiclass (separate-ps) approach.

Input Represenation	% correct (1000-word test set)				Decision Tree data (mean)	
	Level of Aggregation					
Processing Direction*	Word	Letter	Phoneme	Stress	Leaves	Depth
ExtContext(M & L):						
Forward	11.8	61.9	75.8	75.2	1118.4	23.0
Backward	13.3	67.7	79.9	78.6	1016.4	21.0
ExtContext(2)						
Forward	18.5	66.3	81.1	75.9	1039.4	48.0
Backward	21.2	71.9	83.7	80.1	935.4	44.4
ExtContext(7)						
Forward	17.1	68.1	81.5	77.1	950.0	35.0
Backward	22.1	73.0	82.8	83.0	884.0	31.4
Base configuration	12.5	69.6	81.3	79.2	269.9	29.3

*Sequence of processing the letters of the words

Forward = left-to-right, Backward = right-to-left sequence.

Table 40. Impact of several binary representations of the extended context on the performance of the one-per-class (separate-ps) approach.

Input Represenation	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Processing Direction*	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
ExtContext(M & L):							
Forward	12.5	62.1	74.1	74.6	98.2	79.4	12.3
Backward	16.9	70.5	80.2	79.9	98.7	75.8	11.9
ExtContext(2)							
Forward	15.3	67.2	79.2	77.1	98.6	71.7	14.0
Backward	19.0	70.8	81.1	80.2	98.8	65.2	11.4
ExtContext(7)							
Forward	19.7	68.2	80.7	77.3	98.6	66.9	12.1
Backward	24.3	74.7	83.9	83.2	98.9	59.3	11.7
Base configuration	12.5	69.6	81.3	79.2	96.3	269.9	29.3

*Sequence of processing the letters of the words

Forward = left-to-right, Backward = right-to-left sequence.

Table 41. Effect of sharing the previously processed output bits (by the sequential evaluation method) combined with "ExtContext(2-BW)" input representation.

Tree building sequence	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
Straight (1 to 26)	22.1	71.4	82.3	78.6	96.1	86.3	15.1
Optimal sequence	23.0	71.7	83.2	78.2	96.2	86.0	15.9
Base (No Sharing)	19.7	71.5	82.2	80.7	96.1	213.3	25.6

Table 42. Effect of sharing the previously processed output bits (by the sequential evaluation method) combined with "ExtContext(7-BW)" input representation.

Tree building sequence	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit (mean)	Leaves	Depth
Straight (1 to 26)	23.9	74.0	83.5	81.4	96.6	81.6	15.0
Optimal sequence	23.5	74.0	84.0	81.1	96.7	79.6	15.0
Base (No Sharing)	24.4	74.2	83.9	82.6	96.7	193.9	23.7

- No sharing of output bits is employed (for comparison purposes).

Our standard (S & R) distributed output code was employed in these experiments. Results similar to those obtained for the standard input representation are generally observed. These include a substantial reduction in the size of the decision trees and (for the optimal sequential method) some improvement in the performance at the phoneme level at the expense of stress level performance. The letter and word level performance also improve for "ExtContext(2-BW)" (Table 41) but remain essentially unchanged for "ExtContext(7-BW)" (Table 42). This suggests that a good set of input features may make the improvements offered by the sharing of output bits redundant.

4.4 Error Correcting-codes & Extended Context

In Chapter 3, we introduced a new technique for improving the overall performance of learning systems by employing BCH error-correcting codes as a distributed output representation. We showed that these output representations improve the performance of ID3 on the text-to-speech domain with our standard input representation.

In the previous sections, we explored several alternative methods to boost the performance in this domain. These include extending the context of the current letter, selecting a *favourable* encoding for converting the context to binary representation, and processing the letters of the words backwards (right-to-left). We coined the terms "ExtContext(2-BW)" and "ExtContext(7-BW)" to refer to this new combination of input representation and processing sequence. (See Section 4.2.3).

An interesting question is whether the benefits provided by error-correcting output codes are independent of the benefits provided by "ExtContext(2-BW)" or "ExtContext(7-BW)". The answer, as shown in Table 43 (for the separate-ps case) and Table 44 (for the combined-ps case), is unambiguously "yes." The combination of error-correcting output codes with these improved input techniques provides the best performing text-to-speech system that we have studied.

4.5 Voting & Extended Context

In the previous chapter (Section 3.6), we showed that the performance of decision tree building algorithms could be substantially improved by employing *voting* among several *sets* of decision trees. This was shown—for our standard input representation—regardless of the output coding technique employed.

With the substantial improvements obtained by the input techniques presented in this chapter—namely an extended context, better binary representations and

Table 43. Performance of various BCH error-correcting codes with and without extended context and right-to-left (BW) processing of the letters of the words. See text for a precise meaning of "ExtContext(2-BW)" and "ExtContext(7-BW)". Phonemes and stresses are coded separately (the separate-ps case).

BCH Code Data				Input	% correct (1000-wod test set)				Decision Tree	
Phoneme		Stress			Level of Aggregation				data (mean)	
n	*d*	*n*	*d*	Representation	Word	Letter	Phon.	Stress	Leaves	Depth
10	3	9	3	StdContext	13.3	69.8	80.3	80.6	677.4	51.9
				ExtContext(2-BW)	24.3	72.9	82.0	82.3	476.4	40.1
				ExtContext(7-BW)	26.6	74.8	83.4	83.7	431.0	32.0
14	5	11	5	StdContext	14.4	70.9	82.3	80.3	684.7	53.1
				ExtContext(2-BW)	24.3	73.2	83.1	81.6	503.8	42.8
				ExtContext(7-BW)	27.2	75.2	84.6	83.3	453.1	34.0
21	7	13	7	StdContext	17.2	72.2	83.9	80.4	681.4	53.9
				ExtContext(2-BW)	26.1	74.8	84.9	82.4	494.4	39.7
				ExtContext(7-BW)	28.9	76.6	86.0	83.9	453.3	37.1
26	11	13	11	StdContext	17.5	72.3	84.2	80.4	700.5	56.4
				ExtContext(2-BW)	27.0	74.5	85.1	82.0	504.9	41.7
				ExtContext(7-BW)	29.3	76.4	86.2	83.7	456.3	36.8
31	15	30	15	StdContext	19.9	73.8	84.8	81.5	667.8	52.7
				ExtContext(2-BW)	29.7	76.9	86.2	83.8	483.8	40.1
				ExtContext(7-BW)	31.5	77.8	86.7	85.0	441.5	35.9
62	31	30	15	StdContext	20.6	74.1	85.4	81.6	669.9	53.3
				ExtContext(2-BW)	29.7	77.0	86.6	84.0	495.6	41.5
				ExtContext(7-BW)	32.6	78.1	87.3	84.9	444.5	35.7
127	63	30	15	StdContext	20.8	74.4	85.7	81.6	661.6	54.8
				ExtContext(2-BW)	30.2	77.2	86.6	84.1	495.7	41.9
				ExtContext(7-BW)	32.2	78.1	87.3	84.9	458.9	36.9

Table 44. Performance of BCH error-correcting codes with and without extended context and right-to-left (BW) processing of the letters of the words for the combined phoneme/stress case.

Code Length n	Interword Distance d (min.)	Input Representation	% correct (1000-word test set) Level of Aggregation: Word	Letter	Phon.	Stress	Decision Tree data (mean): Leaves	Depth
63	31	StdContext	20.3	74.3	83.8	80.3	1000.0	68.7
		ExtContext(2-BW)	29.3	76.4	85.3	82.3	774.3	53.5
		ExtContext(7-BW)	31.3	77.7	85.8	83.3	733.0	46.3
127	63	StdContext	22.3	75.5	85.2	81.5	1002.9	68.3
		ExtContext(2-BW)	29.0	77.1	86.0	83.2	782.9	53.9
		ExtContext(7-BW)	32.2	78.1	86.8	83.7	739.8	46.5
255	127	StdContext	22.4	75.5	85.2	81.6	1014.4	68.8
		ExtContext(2-BW)	30.6	77.3	86.3	83.1	792.9	55.3
		ExtContext(7-BW)	33.5	78.5	87.1	83.8	745.8	46.9

right-to-left (BW) processing of the letters of the words—one may wonder whether the benefits provided by *voting* are independent of those provided by these other input techniques.

To answer this question, we repeated two of the voting experiments reported in Section 3.6, but with "ExtContext(7-BW)"—our best performing input representation combined with the superior right-to-left (BW) processing of the letters of the words. Tables 45 and 46 present the results of these experiments for 5 (respectively 9) sets of trees voting. The figures shown in these tables unambiguously suggest that the benefits provided by *voting* are orthogonal to those provided by the improved input techniques. Hence, we can generalize the results of Section 3.6 even further: Voting can be employed to improve the performance of decision tree building algorithms regardless of the *input* or *output* coding technique employed.

Table 45. Effect of employing voting among 5 sets of trees each built with 800 words in the training set and "ExtContext(7-BW)" input representation.

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Configuration	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
(1) One-per-class							
(A1) – 1x1000	24.3	74.7	83.9	83.2	98.9	61.2	12.1
(B1) – 5x800	26.4	76.0	84.9	83.9	99.0	52.1	11.1
(C1) – 5x800 float	31.0	78.1	86.6	85.3	99.1	52.1	11.1
(2) (S & R) distributed code							
(A2) – 1x1000	24.4	74.2	83.9	82.6	96.7	193.9	23.7
(B2) – 5x800	27.1	75.8	85.0	83.8	97.1	159.5	21.1
(C2) – 5x800 float	28.3	76.6	86.0	84.0	97.2	159.5	21.1
(3) Ecc code: $pd = 31, sd = 15$							
(A3) – 1x1000	32.6	78.1	87.3	84.9	93.2	454.0	36.5
(B3) – 5x800	32.2	78.7	87.6	85.2	93.2	407.4	36.4
(C3) – 5x800 float	33.4	78.7	87.7	85.3	93.3	407.4	36.4

Table 46. Effect of employing voting among 9 sets of trees each built with 888 words in the training set and "ExtContext(7-BW)" input represenation.

Output Coding Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
Configuration	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
(1) One-per-class							
(A1) – 1x1000	24.3	74.7	83.9	83.2	98.9	61.2	12.1
(B1) – 9x888	26.1	75.8	84.9	83.3	99.0	54.6	11.3
(C1) – 9x888 float	30.1	78.0	86.6	85.1	99.1	54.6	11.3
(2) (S & R) distributed code							
(A2) – 1x1000	24.4	74.2	83.9	82.6	96.7	193.9	23.7
(B2) – 9x888	27.4	75.4	84.5	83.3	97.0	167.2	21.7
(C2) – 9x888 float	29.0	76.8	86.1	84.3	97.2	167.2	21.7
(3) Ecc code: $pd = 31, sd = 15$							
(A3) – 1x1000	32.6	78.1	87.3	84.9	93.2	454.0	36.5
(B3) – 9x888	32.7	78.6	87.5	85.1	93.3	428.6	36.5
(C3) – 9x888 float	33.2	78.6	87.6	85.2	93.3	428.6	36.5

4.6 Effect of Window Size

In all the experiments reported so far, we have been employing a 7-letter window. In this section, we will explore the effect of enlarging the window size on the performance of several configurations that employ distributed output codes.

Table 47 shows the impact of enlarging the window when the standard (Sejnowski & Rosenberg) output code is employed—both with "StdContext" and "ExtContext(7-BW)". Table 48 presents similar data but for a 92-bit BCH error-correcting output code (a 62 bit, $d = 31$ code for phonemes, and a 30 bit, $d = 15$ code for stresses). Note that with the *extended context*, the phonetic context of i letters to the right of the current letter is included, where $i = \lfloor w/2 \rfloor$, w being the size of the window.

There is no uniform (general) trend that can be tracked regarding the impact of enlarging the size of the window on performance. Depending on the particular input/output configuration employed, "optimal" performance for each level of aggregation may occur with a window size ranging anywhere from 7 to 15. Hence, in general, cross validation techniques should be used to arrive at the "optimal" window size for each method and configuration.[39] Having said that, let us focus on one particular input/output configuration: that with error-correcting output codes and "ExtContext(7-BW)" (bottom half of Table 48). There, it seems that a larger window size w slightly benefits stress performance while marginally degrading phoneme performance. Letter and word level performance, however, gradually improve and reach their peak performance at $w = 13$.

We will conclude this chapter by considering the effect of a larger window size for the *combined* phoneme/stress case with several *long* BCH error-correcting codes and "ExtContext(7-BW)". The performance figures for these configurations and two window sizes (7 and 13) are compared in Table 49. Once again, these results

[39] We will follow this path in Chapter 6 where the best configurations for several learning methods are sought to be compared with the performance of DECtalk's rule-base.

Table 47. Impact of enlarging the window when the standard (Sejnowski & Rosenberg) output code is employed—both with "StdContext" and "ExtContext(7-BW)".

Context employed	Window Size w	% correct (1000-word test set) Level of Aggregation: Word	Letter	Phoneme	Stress	Bit	Decision Tree data (mean): Leaves	Depth
StdContext								
	7	12.5	69.6	81.3	79.2	96.3	269.9	29.3
	9	12.9	69.0	80.9	79.1	96.6	236.0	25.2
	11	14.5	68.7	81.0	78.6	96.6	228.7	24.1
	13	14.0	68.5	80.4	79.0	96.5	225.5	23.5
	15	13.9	68.7	80.3	79.1	96.5	223.4	24.1
ExtContext(7)								
letters	7	24.4	74.2	83.9	82.6	96.9	180.1	22.0
processed	9	25.1	72.9	82.8	82.2	96.9	158.5	20.1
right	11	25.4	73.4	82.9	82.5	97.0	155.3	19.9
to	13	26.2	73.4	82.8	82.6	97.0	152.7	19.5
left	15	25.9	73.4	82.7	82.7	96.9	151.9	19.1

StdContext means the standard w-letter window context.

ExtContext = StdContext + the phonetic context of the *right* $\lfloor w/2 \rfloor$ letters.

Table 48. Impact of enlarging the window when a 92-bit BCH error-correcting output code is employed (a 62 bit, $d = 31$ code for phonemes, and a 30 bit, $d = 15$ code for stresses)—both with "StdContext" and "ExtContext(7-BW)".

Context	Window Size	% correct (1000-word test set)					Decision Tree data (mean)	
		Level of Aggregation						
employed	w	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
StdContext								
	7	20.6	74.1	85.4	81.6	92.0	669.9	53.3
	9	21.8	74.4	85.9	82.1	92.2	633.1	50.7
	11	22.3	74.1	85.6	82.2	92.2	613.0	48.9
	13	22.6	74.5	85.7	82.6	92.3	602.6	47.8
	15	22.8	74.5	85.5	83.0	92.3	597.1	48.5
ExtContext(7)								
letters	7	32.6	78.1	87.3	84.9	93.3	444.5	35.7
processed	9	34.8	78.1	87.3	85.1	93.3	418.5	33.1
right	11	34.5	78.3	87.2	85.5	93.4	407.1	33.3
to	13	35.6	78.6	87.2	85.9	93.5	399.7	33.0
left	15	35.4	78.3	87.0	85.7	93.4	398.1	33.2

StdContext means the standard w-letter window context.

ExtContext = StdContext + the phonetic context of the *right* $\lfloor w/2 \rfloor$ letters.

Table 49. Impact of enlarging the window for the *combined* phoneme/stress case when several *long* BCH error-correcting codes and "ExtContext(7-BW)" are employed.

Code Length n	Interword Distance d (min.)	Window Size w	% correct (1000-word test set) Level of Aggregation				Decision Tree data (mean)	
			Word	Letter	Phon.	Stress	Leaves	Depth
63	31	$w = 7$	31.3	77.7	85.8	83.3	733.0	46.3
		$w = 13$	35.4	78.1	85.4	84.3	667.8	44.8
127	63	$w = 7$	32.2	78.1	86.8	83.7	739.8	46.5
		$w = 13$	36.0	78.5	86.3	84.5	673.5	45.7
255	127	$w = 7$	33.5	78.5	87.1	83.8	745.8	46.9
		$w = 13$	36.4	79.1	86.5	85.4	680.7	46.1

indicate that unlike the phoneme performance, the stress performance benefits from the larger window size. Moreover, the improved stress performance is more than enough to offset the reduction in phoneme performance, resulting in an overall improvement at the letter and word levels for the larger window size of 13.

It is instructive to compare the best performance figures shown in the above table with those of our base configuration (Table 1 of Chapter 1). The word level correctness of 36.4% (achieved by the $n = 255, d = 127$ code and $w = 13$) is nearly three times the 12.5% correctness figure that we started out with. Performance figures at other levels of aggregation (phoneme, stress, and letter levels) have also dramatically increased. Indeed, we have come a long way!

Equally impressive is the fact that the machine learning techniques we have developed reached this level of performance (when tested on 1000 *unseen* words) after training on *only* 1000 words. If the reader at this point becomes curious to learn the best performance when larger training sets are employed, a peek at Table 72 in Chapter 6 should satisfy that curiosity.

Chapter 5

The Wolpert Method

In the past few years, several researchers have suggested that learning and generalization are closely related to classical approximation theory. [Poggio89], for example, argued that networks currently used for learning an input-output mapping from a set of examples can be considered as specific methods of approximation and hence, the problem of learning should be approached from the point of view of classical approximation theory. In their work, Poggio and Girosi addressed the generalization issue within the framework of regularization theory which views generalization as surface-fitting. Based on these notions, they developed a technique called GRBFs (Generalized Radial Basis Functions) which map to a class of 3-layer neural networks.[40]

More recently, David Wolpert—in a series of papers [Wolpert89a, Wolpert90a, Wolpert90b]—presented what he called a "partial vision of a broad mathematical theory of generalization". The primary objection that Wolpert voiced against viewing regularization theory as *the* theory of generalization was that regularization theory is *too* wedded to viewing generalization as surface fitting, and that—in his words—it is not clear why the regularizers commonly used in the theory (e.g., fit the points of the training set with a surface of minimal integrated curvature) should be related to good generalization in the real world.

[40]For details on the performance of GRBFs on the NETtalk task see [Wettschereck90].

In his theory of generalization, Wolpert coined the term HERBIE (HEuRistic BInary Engine) to refer to a class of generalizers whose sets of mapping functions $g\{i\}$—from inputs and i training examples to outputs—possess certain desirable properties for improved generalization. In a more recent article that appeared in Neural Networks (1990) [Wolpert90c], Wolpert also introduced a form of a nearest neighbour algorithm for the text-to-speech mapping task based on the notion of HERBIEs. He referred to the algorithm as "a generalizer for the task of reading aloud" or "a self-guessing, metric-based HERBIE". He also reported that his simple method outperformed Sejnowski and Rosenberg's NETtalk on this task, and cited error rates of 18% compared with 22% for NETtalk.[41]

We have implemented Wolpert's generalizer for the text-to-speech domain and experimented with several variations of it in this domain. In the following section, we will describe our implementation (henceforth refered to as "Wolpert's HERBIE" or simply "Wolpert"). Following that, we will discuss the procedure that we followed to set the required parameters for the Wolpert method. Finally, in Section 5.3, we present the performance results obtained by applying the standard method—and several enhancements that we investigated—to our standard 1000-word training and test sets. Results of applying the Wolpert method to the 19,002 words (full dictionary less the test set) will be presented in Chapter 6.

5.1 Wolpert's HERBIEs

In his theory of generalization [Wolpert89a, Wolpert90a, Wolpert90b], Wolpert defines a HERBIE (HEuRistic BInary Engine) as any generalizer whose sets of mapping functions $g\{i\}$—from inputs and i training examples to outputs—satisfy the following properties:

[41]These results correspond to performance figures of 82% (Wolpert) and 78% (NETtalk) which must be correctness at the phoneme level. The training and test data were taken from the transcribed speech of a 6-year-old child [Carterette74].

- All $g\{i\}$ are invariant under rotation, translation, scaling, or parity inversion of the input space and/or the output space.
- All $g\{i\}$ are invariant under reordering of the elements of the training set.
- Taken together, the $g\{i\}$ always reproduce the training set.

Based on these general notions, Wolpert argued that the Backpropagation algorithm has poor generalization power when compared to HERBIEs, and he presented several comparative studies to support his claim. The most recent such study covered the text-to-speech conversion task. In [Wolpert90c], Wolpert introduced a form of a nearest neighbour algorithm for this mapping task that can be described by the following general statement:

Given a *question* $\vec{q} = \{q_1 \ldots q_7\}$ (i.e. a 7-letter window to classify), Wolpert's HERBIEs would guess the class (phoneme/stress)[42] to be assigned to the current letter as the "center of mass" of the four elements of the training set whose input (i.e. 7-letter window) lies *nearest* to the *question* $\vec{q}$.

There are several terms that need to be defined in order to make the above general statement more precise. First, a distance measure must be selected so that the four *nearest* neighbours could be determined based on this metric. The distance measure employed is a *weighted* Hamming distance in the seven dimensional input space defined by the 7-letter windows. More precisely, the distance between two 7-letter windows $\vec{x}$ and $\vec{q}$ is defined by

$$d(\vec{x},\vec{q}) \equiv \sum_{j=1}^{7} \rho_j[1 - \delta(x_j, q_j)]$$

where $\delta(.,.)$ is the Kronecker delta function ($\delta(x_j, q_j) = 1$ if $x_j = q_j$, $\delta(x_j, q_j) = 0$ otherwise), and $\{\rho_1 \ldots \rho_7\}$ is a set of real numbers (*weights*). We will discuss several methods for determining these weights in Section 5.2.

[42]Wolpert's work covered only phoneme classification. We, of course, employed both phonemes and stresses in our implementation.

Next, we need to define a "mass" and a "position" for each element of the training set so that the "center of mass" can be calculated from the four nearest neighbours to the question $\vec{q}$. The "mass" of each element of the training set was taken as the reciprocal of its input-space distance to $\vec{q}$ as defined above. To define a "position" for each element of the training set, we need to decide on a *binary* representation of the ouput classes. For the purposes of this discussion, let us assume that the standard (Sejnowski & Rosenberg), 26-bit distributed code is employed. Hence, each example in the training set will have the form $(\vec{x}, \vec{u})$, where

> $\vec{x}$ is a 7-letter window as described earlier, and
> $\vec{u} = \{u_1 \ldots u_k \ldots u_{26}\}$ is the binary representation of the phoneme/stress that the 7-letter window $\vec{x}$ maps to.

We now define the "position" of each element of the training set—when computing the "center of mass" for output bit k—as u_k.

With the necessary definitions now in place, we can describe the Wolpert method for classifying a question $\vec{q}$ as follows:

1. Scan the training set, calculating the distance between each element $(\vec{x}, \vec{u})$ and $\vec{q}$ by the metric described above. Keep track of the nearest four neighbours to the question $\vec{q}$. Call these $(\vec{x_1}, \vec{u_1})$ through $(\vec{x_4}, \vec{u_4})$.

2. Compute the value of the k^{th} bit of the ouput code to be assigned as the *answer* to $\vec{q}$ by the formula

$$f_k = \frac{\sum_{i=1}^{4} \{u_{ki}/d(\vec{x}_i, \vec{q})\}}{\sum_{i=1}^{4} \{1/d(\vec{x}_i, \vec{q})\}}$$

   where

   > $d(\vec{x}_i, \vec{q})$ is the distance metric defined earlier,
   > i runs over the four "nearest neighbours", and
   > k runs over the 26 bits defining each output vector.

The computed values, $f_1 \ldots f_{26}$, will be real numbers between 0 and 1. They can each be rounded to an integer and then the resulting binary vector mapped to the nearest *observed*[43] phoneme/stress pair. We will identify this scheme in our table entries by the label "thresholded". Alternatively, we can treat $f_1 \ldots f_{26}$ as a vector of real numbers and map it to the nearest *observed* phoneme/stress pair by employing a modified distance measure: $d(\vec{x}, \vec{y}) = \sum_i |x_i - y_i|$. The latter method was found to be superior. Hence, in all the experiments reported in this thesis for the Wolpert method, it can be assumed that "thresholding" is not employed—unless explicitly stated.

5.2 Determining the parameters ρ_j

A key factor that greatly influences the performance of the Wolpert method is the value assigned to each of the weights ρ_1 through ρ_7. Wolpert employed knowledge of the domain to come up with a number of potentially "good" set of weights. He then chose among these sets through cross validation techniques.[44] The "best" set of weights that Wolpert arrived at with this method is shown in Table 86 of Appendix E. We refer to this set of weights as *Wolpert weights*.

We decided to adopt a more formal approach to come up with a good set of values for the weights ρ_1 through ρ_7. For each letter position j in the 7-letter window,[45] we set ρ_j equal to the *mutual information* between the letter at position j

[43]We employed observed decoding throughout the experiments reported in this thesis. Other decoding strategies (such as *legal* or even *block* decoding) work just as well. These were tested but will not be reported to avoid introducing yet another dimension to an already crowded set of degrees of freedom.

[44]To carry out cross validation runs, the standard training set is split into two: a "subtraining" set and a "cross-validation" set. Learning is done from the subtraining set, and the best set of weights is chosen as the set that provides the best performance when evaluated on the cross-validation set.

[45]In these discussions, we assume that a standard 7-letter window is employed. However, in our work, we have computed weights for windows containing up to 15-letters with these methods.

in the window and the output. All the required frequency (probability) information was gathered from our standard 1000-word training set. We repeated the *mutual information* computations three times for each letter position; once considering the outputs to be phoneme/stress pairs to get what we will refer to as *combined-ps weights*, a second time considering the outputs to consist of phonemes only (*phoneme-weights*) and a third time considering the outputs to consist only of the stresses (*stress-weights*). These weights are documented in Table 87 of Appendix E.

5.2.1 Base Configuration: Wolpert

Table 50 shows the results of applying the Wolpert method with a standard 7-letter window context and the Sejnowski & Rosenberg 26-bit distributed output code. Results of employing several sets of weights are shown, each *with* and *without* thresholding of the resulting output bits before mapping to the nearest *observed* phoneme/stress pair. An explanation of each set of weights[46] employed follows:

1. Wolpert weights are the weights employed by Wolpert in his original reported study.

2. (M & L) MI weights are the mutual information values between each letter position and the output *phonemes* as reported in [Lucassen83].

3. Combined-ps weights are the mutual information values between each letter position and the output *phoneme/stress pairs.* These were computed from our standard 1000-word training as indicated in Section 5.2.

4. Separate-ps weights indicates that the *phoneme-weights* are employed when determining the phoneme bits and the *stress-weights* are employed when determining the stress bits. Recall from Section 5.2 that these weights are mutual information values between each letter position and the phonemes (respectively stresses) taken as being the outputs. These were computed

[46]See Appendix E for the actual values of the various set of weights employed.

Table 50. Performance of the Wolpert method with a standard 7-letter window context and the Sejnowski & Rosenberg 26-bit distributed output code. See text for explanation of the various set of weights employed.

Weights employed	Threshold output bits?	% correct (1000-word test set)				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
Wolpert weights	YES	13.0	71.1	80.6	80.2	96.6
	NO	13.9	72.0	81.8	80.7	96.6
(M & L) MI weights	YES	14.9	72.2	82.3	80.3	96.7
	NO	14.9	72.5	82.9	80.4	96.7
Combined-ps weights	YES	15.4	72.6	82.2	80.8	96.7
	NO	15.6	72.9	82.8	81.1	96.7
Separate-ps weights	YES	15.1	72.3	82.0	80.8	96.7
	NO	15.6	72.7	82.8	81.1	96.7

from our standard 1000-word training set. Note that unlike the other cases, employing separate weights for the phoneme and stress bits requires that the search for the four nearest neighbours be performed twice, once for the phonemes and a second time for the stresses.

There are several things to note in Table 50. First, the performance of the Wolpert method with *all* the weights that are based on the mutual information computations is superior to the performance of the method with the "best" weights obtained by Wolpert through cross validation runs. This is exciting, since it indicates that the mutual information values constitute the best known set of weights, which might make a comprehensive search of the weight space redundant!

Next, let us look at the impact of "thresholding"—i.e. rounding the output values computed for each bit to binary—before mapping to the nearest *observed* phoneme/stress pair. At *all* levels of aggregation and, for *all* the sets of weights considered, "thresholding" degrades the performance of the Wolpert method. This

Table 51. Comparing the performance of the "base configuration" for the Wolpert method with that for the ID3 and the Backpropagation algorithms.

Base configuration for	% correct (1000-word test set)				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit
Wolpert	15.6	72.9	82.8	81.1	96.4
ID3	12.5	69.6	81.3	79.2	96.3
BP	14.3	71.5	82.0	81.4	96.7

was also found to be the case in all the variations of the Wolpert method that we experimented with. Hence, in reporting the results of these variations (Section 5.3), we will only show the results when "thresholding" is not employed.

Finally, we note that the best stress, letter and word level performance shown in Table 50 is that with the Combined-ps weights (without "thresholding"). The more expensive case of employing separate-ps weights for phonemes and stresses did not offer any improvement in performance. Hence, we will adopt this set of weights in our "base configuration" for the Wolpert method.

Now that we have defined what we mean by a "base configuration" for the Wolpert method, let us focus our attention on the performance of this configuration relative to our base configuration for the ID3 algorithm defined in Chapter 1, and to that of the Backpropagation algorithm (evaluated under comparable conditions). These performance figures are shown in Table 51. It is striking that the Wolpert method *significantly* outperforms the basic ID3 method at *all* levels of aggregation in the text-to-speech domain. The performance figures, however, are not so striking when compared to that of BP. BP slightly outperforms the Wolpert method at the stress level. The latter comes out in the lead (but not by wide margins) at the phoneme, letter and word levels of aggregation.

Table 52. Comparing the performance of the Wolpert method with Wolpert's original weights with that for the Backpropagation algorithm.

Method	% correct (1000-word test set)				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit
Wolpert	13.9	72.0	81.8	80.7	96.6
BP	14.3	71.5	82.0	81.4	96.7

5.2.2 Wolpert vs. Backpropagation

The previous comparison of the Backpropagation (BP) algorithm and the Wolpert method compared *our* base configuration of Wolpert with BP. However, in order to properly determine the validity of Wolpert's claims regarding the superiority of his method to BP, one must compare the performance figures of the Wolpert method when the *original Wolpert weights* are employed and not when our—superior—set of weights are used. Table 52 shows the performance of the Wolpert method with Wolpert weights together with the performance of the Backpropagation algorithm when both methods are trained on our standard 1000-word training-set and tested on the standard 1000-word test set. The same input encoding, output encoding, and, decoding strategy were employed for both methods. Based on this comparison, we believe that Wolpert's claims of having a generalizer "superior" to NETtalk were greatly exaggerated. The performance of the two methods is statistically indistinguishable, specially at the *phoneme* level which Wolpert considered in his study.

We attribute the discrepancies between our findings and Wolpert's pusblished results to the following:

1. Wolpert chose a slightly different domain for his study than the NETtalk domain under our consideration. The training sets and test sets employed were entirely different than ours.

2. Wolpert processed the training set he employed to remove any inconsistencies present [Dietterich90c]. He alluded to this in his paper [Wolpert90c] as follows:

   > Only the first of the two pairs of learning sets that Sejnowski and Rosenberg indicate they used (once suitably modified for use by the center of mass HERBIE) was used to teach the center of mass HERBIE system.

This processing of the training set (we believe) gave the Wolpert method an unfair advantage over Backpropagation.

5.3 Variations

In Chapters 3 and 4, we developed several techniques that considerably enhance the performance of decision tree building algorithms on the text-to-speech task. In this section, we will explore—among other things—the impact of these techniques when applied to the Wolpert method. First, we will investigate the effect of varying some of the parameters that were "fixed" in our dicussion of the Wolpert's method (Section 5.1) in order to simplify the exposition. In the following two sections, we will explore the impact on performance of varying two of these parameters: the number of nearest neighbours considered in the "center of mass" computations (previously *four*), and the size of the window (previously *seven*). Following that (Section 5.3.3), we will abandon the standard 26-bit distributed output code and consider instead distributed BCH error-correcting output codes. Section 5.3.4 then considers extending the 7-letter context to include the phonetic context of the 3-letters to the *left* (*right*) of the current letter, corresponding to processing the letters of the words in a left-to-right (right-to-left) sequence. Finally, in Section 5.3.5, we consider the effect of extending the input space even further by including the additional features that we made available to the ID3 algorithm

Table 53. Impact of varying the number of nearest neighbours considered on the performance of the Wolpert method.

Number of neighbours considered (nn)	% correct (1000-word test set)				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit
3	15.4	72.4	82.6	80.6	96.7
4	15.6	72.9	82.8	81.1	96.7
5	15.7	72.7	83.0	81.0	96.8
6	14.9	72.7	83.0	81.2	96.8
7	14.1	71.9	82.3	81.0	96.7

with "ExtContext(7)".

5.3.1 Number of Nearest Neighbours

Table 53 shows the impact of varying the number of nearest neighbours considered in the "center of mass" computations on the performance of the Wolpert method. In the experiments shown in this table, values for other parameters are kept the same as for the "base configuration" identified in the previous section. These include a standard 7-letter window, the Sejnowski & Rosenberg 26-bit distributed output code and employing the combined-ps weights described in earlier sections. The results show that while considering as many as six nearest neighbours ($nn = 6$) *may* offer a *slight* improvement in phoneme and stress level correctness, the differences are insignificant when compared with the $nn = 4$ case. $nn = 4$ (5) seems to offer the best letter (word) level performance. The results suggest that any small value for $nn \in \{4, 5, 6\}$ will do just as well.

5.3.2 Window Size

Table 54 shows the impact of varying the window size w on the performance of the Wolpert method. As always, values for other parameters are kept the same as

Table 54. Impact of varying the window size on the performance of the Wolpert method.

Window size w	% correct (1000-word test set)				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit
7	15.6	72.9	82.8	81.1	96.7
9	16.7	73.2	83.1	81.3	96.8
11	16.7	73.1	83.0	81.4	96.8
13	16.5	72.9	82.6	81.7	96.8
15	16.6	72.9	82.6	81.8	96.8

for the "base configuration" identified earlier. The results show that enlarging the window offers a slight—but consistent—increase in the stress level performance. The phoneme and letter level performance peaks at $w = 9$, then reverses direction. Word level performance remains essentially unchanged for $w \geq 9$. We can therefore conclude that the Wolpert method— with weights derived by the mutual information method—is not very sensitive to enlarging the size of the window.

5.3.3 The Wolpert Method & Error-correcting Codes

In Section 5.1, we described the Wolpert method assuming that the output code employed is our standard (Sejnowski & Rosenberg) 26-bit distributed output code. The discussion, however, generalizes readily to the case where any *arbitrary binary code* is employed for the outputs. An interesting question is whether employing BCH error-correcting output codes will offer improvements in the performance of the Wolpert method similar to those observed for ID3 in Chapter 3. To answer this question, Table 55 compares the performance of the Wolpert method when several BCH error-correcting output codes—of various lengths and inter-word Hamming distances—are employed with that when the standard (Sejnowski & Rosenberg) 26-bit distributed output code is employed as an output representation. The results show that employing error-correcting codes degrades the performance at the

Table 55. Comparing the performance of the Wolpert method with and without BCH error-correcting output codes.

Code Length n	Interword Distance d (min.)	% correct (1000-word test set) Level of Aggregation: Word	Letter	Phoneme	Stress	Bit
63	31	16.2	73.1	82.9	80.3	86.8
127	63	16.4	73.1	82.9	80.4	86.6
255	127	16.5	73.2	83.0	80.4	86.6
(S & R) 26-bit code		15.6	72.9	82.8	81.1	96.7

stress level while offering some (slight) improvement at the phoneme, letter and word levels. These improvements, however, are minor when compared with those obtained by the ID3 method in Chapter 3.

The relative ineffectiveness of error-correcting output codes with the Wolpert method is discouraging, but understandable. Because all of the bits are computed in this method using the *same* four nearest neighbours, the individual bit errors are expected to be *extremely* highly correlated. We found in Section 3.4.8, however, that for error-correcting codes to be effective the individual bit errors must be independent from one another. Table 56 shows that the situation is not any better if separate weights are employed for the phonemes and stresses. Despite the "potential" for choosing a different set of four neighbours when computing the phoneme bits than when computing the stress bits in this case, bit error correlations do not seem to be any lower. We have measured these correlations and found that they are indeed very high (in the 0.4 to 0.5 range in general). Comparing these with the bit error correlation data reported in Table 12 for ID3, shows that the Wolpert figures are consistently higher.

Table 56. Comparing the performance of the Wolpert method with and without BCH error-correcting output codes when separate weights are employed for phonemes and stresses.

Code Length n	Interword Distance d (min.)	% correct (1000-word test set)				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
63	31	15.0	72.8	82.7	80.0	86.6
127	63	15.2	72.8	82.7	80.1	86.5
255	127	15.2	72.8	82.8	80.0	86.4
(S & R) 26-bit code		15.6	72.7	82.8	81.1	96.7

5.3.4 The Wolpert Method & Extended Context

In Chapter 4, we explored several *input* techniques to boost the performance of decision tree building algorithms in the text-to-speech domain. These included extending the context of the current letter, selecting a *favourable* encoding for converting the context to binary representation, and processing the letters of the words backwards (right-to-left). A natural question that comes to mind is whether or not similar techniques could be employed to improve the performance of the Wolpert method in this domain.

Extending the context for the Wolpert method to include the *phonetic* context (phonemes and stresses) of the letters to the left (or to the right) of the current letter is straight forward, since the method as described in Section 5.1, does not care whether the input space consists of letters only, or of a combination of letters, phonemes and stresses. All we need to do is to assign the proper weight ρ_j to the j^{th} element of the context and let j run over *all* the elements of the context when computing the distance measures $d(\vec{x}, \vec{q})$.

As was done for the letters (See Section 5.2), we decided to set the weights for the phonemes (stresses) when they are part of the input space equal to the *mutual information* between these phonemes (stresses) and the output. For this purpose, we computed the mutual information data between *phonemes* (*stresses*) corresponding to letters j positions away from the current letter (in a 15-letter window) and the output class of the current letter. The computations were repeated three times for each phoneme (stress) position; once considering the outputs to be phoneme/stress pairs to get the *combined-ps weights*, a second time considering the outputs to consist of phonemes only (*phoneme-weights*) and a third time considering the outputs to consist only of the stresses (*stress-weights*). These weights are given in Appendix E. Table 88 shows the data that applies to phonemes in the input space while Table 89 shows similar data—but for stresses that are part of the input space.

Table 57 shows the performance of the Wolpert method when *extended context* is employed. Results for both "forward" (left-to-right) processing and "backward" (right-to-left) processing of the letters of the words are shown; once with the standard (Sejnowski & Rosenberg) 26-bit distributed output code, and again with an $n = 63, d = 31$ BCH error-correcting output code. In these experiments, a standard 7-letter window was employed. This implies that the input space consisted of 13 elements: 7 letters, 3 phonemes and 3 stresses corresponding to the letters to the left (right) of the current letter. Common weights (the *combined-ps* weights) were employed when determining *both* the phoneme *and* the stress bits. Finally, the number of nearest neighbours (*nn*) considered was 4, and the output bits were *not* thresholded before mapping to the nearest *observed* phoneme/stress pair.

The results shown in Table 57 unambiguously indicate that—as was the case for ID3—including the *left* phonetic context *degrades* the performance of the Wolpert method in this domain at *all* levels of aggregation. The results also suggest that the Wolpert method *may* not be able to take advantage even of the *right* phonetic context in the same way that ID3 does.

Before making a final judgement, however, we decided to examine the effect of varying the relative magnitude of the weights assigned to the phonemes and stresses (in the extended context) to the weights assigned to the letters in the

Table 57. Performance of the Wolpert method when *extended context* is employed. Effects of including the *phonetic context* of the letters to the *left* (processing forward) as well as the *phonetic context* of the letters to the *right* (processing backwards) are shown.

Output code employed	Direction of processing	% correct (1000-word test set)				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
Standard (S & R)	FORWARD	11.5	65.9	76.2	78.2	95.8
26-bit code	BACKWARD	15.6	70.5	80.2	80.3	96.3
Error-correcting	FORWARD	12.9	65.8	76.0	77.4	83.1
code, $(n = 63, d = 31)$	BACKWARD	16.0	70.7	80.2	79.6	85.5

context. In the experiments reported in Table 58, we fixed the magnitude of the weights assigned to the *letters* in the context to be the combined-ps weights as before. The weights used for the phonemes and stresses (in the extended context) were *reduced*[47] by multiplying them by a fraction p. p was varied from 0.5 down to 0.1 in increments of 0.1. This procedure resulted in an improved performance for all values of p between 0.5 and 0.1 when compared to the performance with the original weights computed for the phonemes and stresses ($p = 1.0$ in the table). This suggests that the weights assigned to these other elements of the context should be based on the *additional* mutual information contributed by these elements *given* the information provided by their corresponding letters.

With the modified weights for some of the elements of the extended context, we can now see some advantage in utilizing the *right* phonetic context and error-correcting codes with the Wolpert method. However, the best performance of 18.9% (74.0%) at the word (letter) levels shown in Table 58 should be compared with the 29.3% (76.4%) corresponding figures for ID3 (Table 44 for the combined-

[47]Upon examination of the weights assigned to the phonemes and stresses by the mutual information computations, we decided that the magnitudes of these weights were too high compared to the magnitudes of the weights assigned to the letters.

Table 58. Performance of the Wolpert method when ***extended context*** is employed, and the weights used for the phonemes and stresses (in the extended context) are ***reduced*** by multiplying them by a parameter p. Letters of the words are processed backward, i.e. in a right-to-left sequence.

Output code employed	p	% correct (1000-word test set)				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
Standard (S & R) 26-bit code	1.0	15.6	70.5	80.2	80.3	96.3
	0.5	17.9	73.3	83.5	81.8	96.8
	0.4	17.6	73.4	83.5	81.8	96.8
	0.3	17.8	73.5	83.6	81.5	96.9
	0.2	17.1	73.4	83.7	81.4	96.9
	0.1	16.9	73.5	83.5	81.5	96.8
Error-correcting code, $(n = 63, d = 31)$	1.0	16.0	70.7	80.2	79.6	85.5
	0.5	18.8	73.8	83.6	81.3	87.1
	0.4	18.6	73.9	83.7	81.5	87.2
	0.3	18.9	74.0	83.8	81.3	87.2
	0.2	17.8	74.0	83.8	81.1	87.2
	0.1	18.2	73.9	83.6	81.1	87.2

ps case and the *same* error-correcting code—$n = 63, d = 31$—employed above and, ExtContext(2-BW)). Based on this comparison, the Wolpert method lags far behind ID3 in this domain, when *both* are augmented with our techniques of error-correcting output codes and an extended context.

5.3.5 The Wolpert Method & ExtContext(7-BW)

In the previous section, we saw that—for certain combinations of weights and output coding techniques—the Wolpert method may benefit (slightly) from an extended context. In this section, we consider the effect of extending the input space even further by including the additional features that we made available to the

ID3 algorithm with the "ExtContext(7)" representation defined in Section 4.2.3.

For the Wolpert method, ExtContext(7-BW) implies that the same information conveyed by the *extra* bits in ExtContext(7) is made available in the representation of the windows, and that the letters of the words were processed in a right-to-left order. Hence, a 7-letter *extended context* for this method means that each window consists of a 36-character string as follows:

L. 7 characters (each $\in \mathcal{A}$, where $\mathcal{A}$ is the set containing the 26 letters and the three symbols "-", "_", and ".") representing the 7 letters in the window.

L_{f1}. 7 characters (each $\in \{0,1\}$) indicating whether each of the 7 letters in the window is a consonant, (0), or not, (1).

L_{f2}. 7 characters (each $\in \{0,1\}$) indicating whether each of the 7 letters in the window is a non-low vowel (1), or not, (0).

P. 3 characters (each $\in \mathcal{P}$, where $\mathcal{P}$ is the set of 54 phonemes defined earlier) representing the 3 *right* phonemes which are part of the *extended context.*

P_{f1}. 3 characters (each $\in \{0,1\}$) indicating whether each of the above *right* phonemes is tense, (1) or not (i.e. lax), (0).

S. 3 characters (each $\in \mathcal{S}$, where $\mathcal{S} = \{\texttt{0,1,2,>,<,-}\}$ is the set of 6 stress symbols employed in the NETtalk dictionary) representing the 3 *right* stresses which are part of the *extended context.*

S_{f_1}. 3 characters (each $\in \{0,1\}$) indicating whether each of the above *right* stresses is $\in \{\texttt{0,1,2}\}$, (1) or not, (0).

S_{f_2}. 3 characters (each $\in \{0,1\}$) indicating whether each of the above *right* stresses is $\in \{\texttt{1,2}\}$, (1) or not, (0).

Weights to be assigned to the "extra" elements—$L_{f1}, L_{f2}, P_{f1}, S_{f_1}$, and S_{f_2}—introduced as part the input space were set (as usual) to the value of the mutual

Table 59. Performance of the Wolpert method when *extended context* is employed, the weights used for the phonemes and stresses (in the extended context) are *reduced* by multiplying them by a parameter p, and the extra features defined by ExtContext(7) are included in the input space. Letters of the words are processed backward, i.e. in a right-to-left sequence.

Output code employed	p	% correct (1000-word test set)				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
Standard (S & R) 26-bit code	1.0	18.4	72.6	81.6	82.4	96.6
	0.5	19.7	74.2	83.7	82.7	96.9
	0.4	19.8	74.7	84.2	82.9	97.0
	0.3	19.9	75.1	84.6	82.8	97.1
	0.2	19.4	75.0	84.6	82.8	97.1
	0.1	19.1	74.9	84.4	82.8	97.1
Error-correcting code, ($n = 63, d = 31$)	1.0	18.5	72.6	81.8	81.5	86.5
	0.5	21.4	74.8	84.0	82.6	87.6
	0.4	20.9	75.3	84.3	82.7	87.9
	0.3	20.8	75.7	84.8	83.0	88.1
	0.2	21.1	75.7	84.8	82.8	88.1
	0.1	20.3	75.5	84.6	82.6	88.0

information between that element and the output phoneme/stress pairs. These mutual information values for letters i positions to the left or to the right of the current letter ($i \in \{0, 7\}$) are shown in Tables 90 through 92 of Appendix E.

Table 59 shows the performance of the Wolpert method when this input space—equivalent to "ExtContext(7-BW)"—is employed. The results generally show a marked improvement over those of Table 58. However, we see once again that the best configuration for the Wolpert method lags behind the best *comparable* configuration for ID3 in the NETtalk domain. Table 60 shows this direct comparison.

Table 60. A comparison of some of the "best" configurations for the Wolpert method with *comparable* configurations for ID3 in the NETtalk domain. For both, an $n = 63, d = 31$ BCH Error-correcting code and extended context is employed. Letters of the words are processed backward, i.e. in a right-to-left sequence. Combined-ps.

Context employed	Learning Method	% correct (1000-word test set)				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
ExtContext(2-BW)	Wolpert	18.9	74.0	83.8	81.3	87.2
	ID3	29.3	76.4	85.3	82.3	88.4
ExtContext(7-BW)	Wolpert	21.1	75.7	84.8	82.8	88.1
	ID3	31.3	77.7	85.8	83.3	89.0

Chapter 6

Comparison with DECtalk

In all the experiments reported in chapters 1 to 5, learning is accomplished by training on our standard 1000-word training set. This chapter, however, is performance oriented. The results of the best performing learning systems trained on 19,002 words are compared with the performance of the human-developed letter-to-sound rules incorporated in DECtalk; a commercially available device, marketed by Digital Equipment Corporation (DEC) for the English text-to-speech conversion task.

To carry out this comparison, we had to obtain the pronunciations output by DECtalk's rule-base on our standard 1000-word test set.[48] However, before a valid comparison was possible, a number of obstacles had to be overcome. For one thing, the format of the outputs from DECtalk's rule-base was entirely different from the outputs of our learning systems. All the "silent" phonemes and "no-stress" symbols do not appear in the former, and the primary and secondary stress symbols are intermixed with the remaining phonemes. This means that the phoneme/stress string is not aligned with the letters of the words, which makes the performance evaluation at the phoneme, stress and letter levels nearly impossible. Other, less

[48]We thank Tony Vitali of Digital Equipment Corporation (DEC), for running the entire NETtalk dictionary through DECtalk and providing us with the outputs from DECtalk's rule-base on *all* the words in the dictionary.

serious problems, included the disparity between some of the phonemes and stresses employed by the two systems. The solution we adopted was to painstakingly *re-align* the outputs of DECtalk's rule-base for all the 1000 words in our standard test set. Section 6.1 details the original format for the outputs of DECtalk's rule-base and the procedure we employed to convert these to the standard *aligned* format used in the NETtalk dictionary.

Another difference that had to be eliminated before a valid comparison of our learning systems and DECtalk's rule-base could be made was the differences in stress information. DECtalk's output—once re-aligned—contained only *three* stress symbols corresponding to primary, secondary and no stress. NETtalk's dictionary, on the other hand, used three *additional* symbols which encoded syllable boundary information. These additional symbols were eliminated from the NETtalk dictionary for all the experiments reported in this chapter. We refer to this reduction of stress symbols to only three (primary, secondary and no stress) as *simplified stresses.* Section 6.2 presents the performance results (on our standard test set) of some of our earlier configurations when these *simplified stresses* are employed.

Section 6.3 then embarks on a hill climbing search to find the "best" configuration for three learning methods that we chose to compare with DECtalk: multiclass ID3, Wolpert and the error-correcting output code method. In this search, the learning systems were trained on our standard 1000-word training set but the testing was done on an *alternate test set*—disjoint from both the training set and our standard test set. Section 6.4 then compares the performance of the best configurations obtained as a result of this search—but trained on 19,002 words—to the performance of DECtalk's rule-base.

All the comaprative studies considered thus far employed a *strict* correctness criteria which required phonemes and stresses to match *exactly.* Section 6.5, however, presents the results of a different kind of comparative study: one in which human subjects were asked to judge the quality of the pronunciations output by each of the methods considered.

We conclude this chapter by presenting results of "brain damage" experiments similar to those reported in the connectionist literature. We show that ID3 trees—built when error-correcting codes are employed to encode the classes—exhibit the same resistance to "brain damage" and the same gradual degradation in performance with increased noise levels as that observed in Backpropagation networks.

6.1 Outputs from DECtalk's Rule-base

As mentioned earlier, we decided to evaluate our learning methods by comparing their performance with that of the human-developed letter-to-sound rules incorporated in DECtalk. For that, we obtained—through the valued assistance of Mr. Tony Vitali of Digital Equipment Corporation, (DEC)—the outputs of DECtalk's rule-base on our standard 1000-word test set. The original format of the outputs obtained can be seen from a sample shown in Table 61. Notice that this format is entirely different from that of the NETtalk dictionary also shown in that table for comparison purposes. In DECtalk's outputs, all the "silent" phonemes and "no-stress" symbols do not appear, and the primary and secondary stress symbols (the symbols ' and ‘ respectively) are intermixed with the remaining phonemes. Unlike the outputs in the NETtalk dictionary, there is no longer a one-to-one correspondence between the individual characters in the DECtalk's output strings and the letters of the words, making the performance evaluation at the phoneme, stress and letter levels extremely difficult.

Furthermore, there were some differences in the symbols employed to encode the phonemes in the two systems. The symbol "*", for example, was used in DECtalk's outputs to indicate a boundary between compound words such as "counterclaim" (see Table 61), but in the NETtalk dictionary it is the phoneme corresponding to the pronunciation of the letter "w" in words like "while" and "whistle". Several other such phoneme symbol disparities existed. Table 62 lists all the phoneme

Table 61. Comparing the original format of the outputs from DECtalk's rule-base with the format employed in the NETtalk dictionary.

WORDS	DECtalk's Outputs	NETtalk Dictionary Outputs		Simplified Stresses
		Phonemes	Stresses	
AFFIDAVIT	xf'IdxvIt	@f-xdevxt	2<>0>1>0<	2----1---
AUXILIARY	cksIlx'eri	c-#Ily-Ri	0<<1<0<>0	---1-----
BIMETALLISM	b'Imxtclxzxm	bAmEt-L-IzM	>0>1<0<>2<0	---1----2--
CONVEX	kxnv'Eks	kanvEX	>1<>1<	-1--1-
CONSTITUTIVE	kxnst'Ityutxv	kanstxtYtIv-	>1<>>0>0<0<<	-1----------
COUNTERCLAIM	k'WntR*kl'em	kW-nt-Rkle-m	>1<<<0<>>2<<	-1-------2--
ELECTIONEER	xlEkSxn'ir	IlEkS-xnI-r	0>2<>0<>1<<	--2-----1--
EXEMPLAR	xks'EmplR	I#Emplar	0<1<>>2<	--1---2-
FAILURE	f'elyur	fe-lYr-	>1<<0<<	-1-----
FUNICULAR	fyun'IkyLR	fYnIkYl-R	>0>1<0>0<	---1-----
PURISM	p'Rxzxm	pYrIzM	>1<2<0	-1-2--
UNPROFITABLE	'^nprxf'AtxbL	^nprafxtxbL-	1<>>1<0<0>>0	1---1-------

Table 62. Phoneme symbols that appeared in DECtalk's outputs and required special processing. The "Action to take" column shows the action recommended by Tony Vitali in order to "equate" these outputs with the phoneme symbols employed in the NETtalk dictionary.

DECtalk's Phoneme symbol	Action to Take	NETtalk's phoneme symbol
Group A:		
#	Delete.	None.
*	Delete.	None.
|	Convert to:	x
Y	Convert to:	yu
Group B:		
kS	Convert to:	K
kw	Convert to:	Q
ks	Convert to:	X
ts	Convert to:	!
gz	Convert to:	#
w^	Convert to:	*

symbols that appeared in DECtalk's outputs and that required special processing, along with the action recommended by Tony Vitali in order to "equate" the outputs of the two systems.

Because we wanted our evaluation of the learning methods—and the comparison with DECtalk—to be valid and comprehensive, we decided to convert DECtalk's outputs to the standard *aligned* format employed in the NETtalk dictionary. The next section details the procedure we followed to do this conversion.

6.1.1 Aligning DECtalk's Outputs

Our starting point for the format conversion (or re-aligning) process was a program (called Klattize) written by Charles Rosenberg[49] that did the *reverse* of what we were after: convert a string of phonemes and a string of stresses from the format employed by the NETtalk dictionary to that employed by DECtalk. To simplify the exposition, we will refer to the format employed by DECtalk as the "Klattized" format and the format employed by the NETtalk dictionary as the "aligned" format.

Given a k-letter word $\vec{W}$ and the corresponding output from DECtalk's rule-base $\vec{D}_{klattized}$, our goal is to come up with a string of phonemes $\vec{P}_{D,aligned}$ and a string of stresses $\vec{S}_{D,aligned}$ such that

$$\vec{D}_{klattized} = \text{Klattize}(\vec{P}_{D,aligned}, \vec{S}_{D,aligned})$$

where

> Klattize(.,.) is the function described in the previous paragragh,
> $\vec{P}_{D,aligned}$ and $\vec{S}_{D,aligned}$ are strings of length k each and,
> Each of the k symbols in $\vec{P}_{D,aligned}$ and $\vec{S}_{D,aligned}$ must correspond to the pronunciation of one of the k letters of $\vec{W}$.

Note that in order to achieve the one-to-one correspondence required above, silent phonemes and no-stress symbols must be introduced. Furthermore, if there is more than one choice[50] (for $\vec{P}_{D,aligned}$ or $\vec{S}_{D,aligned}$) that satisfies the above requirements, then the choice that is *closest* to the "correct" pronunciation (i.e. the pronunciation found in the NETtalk dictionary) would be selected.

Let

[49]These programs are distributed by Thinking Machines Corporation as one of their example applications.

[50]This is possible because a silent phoneme (for example) for a two letter combination such as "wh" in "where" could be assigned to *either* letter and, we would still get the same Klattized format after passing it through the function Klattize.

$\vec{P}_{correct}$ and $\vec{S}_{correct}$ be the correct (i.e. dictionary) aligned pronunciation of the word $\vec{W}$ and,

$\vec{C}_{klattized} = \text{Klattize}(\vec{P}_{correct}, \vec{S}_{correct})$

The procedure we followed in order to determine $\vec{P}_{D,aligned}$ and $\vec{S}_{D,aligned}$ can then be summarized as follows:

- Check if any substitutions outlined in Table 62 should be made[51] on $\vec{D}_{klattized}$. If so, apply these substitutions.

- Compare $\vec{D}_{klattized}$ (possibly modified after the above step) with $\vec{C}_{klattized}$. Identify a list of phoneme and/or stress substitutions that when applied to $\vec{C}_{klattized}$ will make it the same as $\vec{D}_{klattized}$.

- Apply the substitutions determined by the above step to $\vec{P}_{correct}$ and $\vec{S}_{correct}$ thereby producing the re-aligned outputs for DECtalk's rule-base: $\vec{P}_{D,aligned}$ and $\vec{S}_{D,aligned}$.

Almost all of the above procedure was automated to ensure that the required string comparisons and substitutions were accurately performed. However, the results of *each* comparison, and a suggested substitution list, were *always* presented to the user for confirmation and for possibly requesting a re-alignment of any two strings being compared. We will clarify these points and some of the difficulties we encountered in the re-alignment process by considering some examples.

Consider the simple word "ACTION", for which $\vec{D}_{klattized}$ was "`'@kSxn`". The Klattized version of the pronunciation of the NETtalk dictionary is also "`'@kSxn`", so the dictionary pronunciations $\vec{P}_{correct}$ ="`@kS-xn`" and $\vec{S}_{correct}$ = "`1-----`" are taken as $\vec{P}_{D,aligned}$ and $\vec{S}_{D,aligned}$ respectively without any substitutions. Note that

[51]The first 3 actions outlined in Table 62 (2 deletions and one substitution) were performed on all the words in advance. The other substitutions were only made if they were appropriate to the particular word $\vec{W}$. Here "appropriate" means that it will make DECtalk's output closer to the dictionary.

this means we did *not* apply the substitution of "kS" with "K" as suggested in Table 62. In general, all the substitutions suggested in that table were made *only* if they were appropriate. Several examples of situations in which the substitutions should be made are shown in Table 61. "ks" in DECtalk's pronunciation of "AUXILIARY", for example, would be converted to "-X" because that would be compared during the evaluation process (at a later stage) with the correct "-#" corresponding to it. This would result in counting only one phoneme error instead of the two that would be counted if "ks" was compared with "-#" which would be the case had the substitution not been carried out. Similarly consider the word "CONVEX". Replacing "ks" with "X" in DECtalk's pronunciation of "CONVEX" would be appropriate judging from the dictionary pronunciation also shown in Table 61. Several examples of other appropriate substitutions also appear in that table.

Next, let us consider a more difficult word, "AFFIDAVIT", for which

DECtalk's output $\vec{D}_{klattized}$ was "`xf'IdxvIt`" and,
NETtalk's output $\vec{C}_{klattized}$ was "`'@fxd'evxt`".

Comparing these two suggested that the following substitutions were necessary:

$$(@ \rightarrow x)\ (x \rightarrow I)\ (e \rightarrow x)\ (x \rightarrow I)$$

Applying these substitutions to $\vec{P}_{correct}$ ="`@f-xdevxt`" yields

$$\vec{P}_{D,aligned} = \text{"}\texttt{xf-IdxvIt}\text{"}$$

A straight forward procedure tailor-made to determine the aligned stresses when only primary stress is present in $\vec{D}_{klattized}$[52] suggested that

[52]When only primary stress is present, it is only necessary to determine the *position* of that stress. A simple routine that tries all the possible k positions and selects the one that results in a match between the final Klattized strings does the job. Care was taken to ensure that when there is more than one position that satisfies this condition, the user would be asked to intervene and select the best choice.

$$\vec{S}_{D,aligned} = \text{“}\texttt{---1-----}\text{”}.$$

As a final example let us consider the word “BIMETALLISM” , for which

DECtalk's output $\vec{D}_{klattized}$ was “`b'Imxtclxzxm`” and,
NETtalk's output $\vec{C}_{klattized}$ was “`bAm'EtL'IzM`”.

This word illustrates another difficulty that we encountered. Consider the phoneme substitutions (and other actions) necessary to make $\vec{C}_{klattized}$ the same as $\vec{D}_{klattized}$:

$$(\mathrm{A} \rightarrow \mathrm{I})\ (\mathrm{E} \rightarrow \mathrm{x})\ (\text{insert c})\ (\mathrm{L} \rightarrow \mathrm{l})\ (\mathrm{I} \rightarrow \mathrm{x})\ (\text{insert x})\ (\mathrm{M} \rightarrow \mathrm{m})$$

Note that in addition to the 5 substitutions required, two insertions are also needed. Insertions, however, are only allowed if a silent phoneme exists at the position in question which can be changed into the required symbol to be inserted. Otherwise, the length of the resulting phoneme string will be longer than k—the number of letters in the word—and we will lose the one-to-one correspondence that we desperately try to preserve. In this case, the correct aligned phoneme string is

$$\vec{P}_{correct} = \text{“}\texttt{bAmEt-L-IzM}\text{”},$$

which enables us to transform the silent phoneme “-” between the “t” and the “L” to the required “c” which needs to be inserted. Inserting the “x” between the “z” and the “M” however would not be allowed. Hence the final aligned outputs assigned were

$$\vec{P}_{D,aligned} = \text{“}\texttt{bImxtcl-xzm}\text{” and,}$$

$$\vec{S}_{D,aligned} = \text{“}\texttt{-1---------}\text{”}.$$

The above examples should be enough to convince the reader that the process of re-aligning DECtalk's outputs was involved, to say the least. Nevertheless, we took extreme care[53] to ensure that the outputs for *every* word—of the 1000-word

[53]The conversion routines that the author wrote were designed to be fool-proof, not allowing the user to make a mistake even if he wanted to.

test set that we converted—were correctly re-aligned. In rare situations where that was not possible (due to the one-to-one correspondence requirement between the aligned phonemes and the letters of the words), the resulting omission of a phoneme (similar to that seen in the last example) would translate to counting one *less* phoneme error against DECtalk.

The performance figures for DECtalk's *aligned* outputs are presented in Table 72 (Section 6.4), where they are compared with the "best" performance of three selected learning methods.

6.2 Effects of Simplified Stresses on Performance

Table 63 shows the effect of simplified stresses on several learning methods when the separate-ps approach is employed. Table 64 presents similar results, but for the combined-ps case. The results shown in both tables indicate that employing simplified stresses gives rise to an improved performance at the stress (and consequently the letter) level of performance. Phoneme and word level performance, however, vary depending on the method and configuration employed. These figures are slightly reduced (with simplified stresses) when distributed output codes are combined with the separate-ps approach;[54] they are slightly improved when multiclass ID3 is employed with the separate-ps approach; but they are significantly improved with the combined-ps approach whether multiclass ID3 or error-correcting codes are employed.

6.3 Best Configurations

In this section, we will undertake a brief "hill climbing" search to find the "best" configuration for three learning methods that we chose to compare with DECtalk: multiclass ID3, Wolpert, and ID3 with the error-correcting output code method.

[54]One reason for this is the loss of the syllable boundary information which is part of the *extended context* when full stresses are employed.

Table 63. Effect of simplified stresses on several learning methods when the separate-ps approach is employed.

Learning Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
STRESSES	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
Distributed (S & R) code							
Full stresses	24.4	74.2	83.9	82.6	96.9	180.1	22.0
Simplified Stresses	22.4	78.3	83.1	88.9	96.3	212.5	25.1
Multiclass							
Full stresses	22.1	73.0	82.8	83.0	N/A	884.0	31.4
Simplified Stresses	23.5	77.7	83.1	88.4	N/A	781.4	37.4
Ecc code: $pd = 31, sd = 15$							
Full stresses	32.6	78.1	87.3	84.9	93.2	454.1	36.5
Simplified Stresses	30.5	81.2	87.0	88.9	93.8	437.7	38.1

Table 64. Effect of simplified stresses on several learning methods when the combined-ps approach is employed.

Learning Method	% correct (1000-word test set)					Decision Tree data (mean)	
	Level of Aggregation						
STRESSES	Word	Letter	Phon.	Stress	Bit	Leaves	Depth
Multiclass							
Full stresses	21.9	73.0	81.5	81.8	N/A	1468.0	30.9
Simplified Stresses	26.6	79.8	83.6	88.8	N/A	1270.9	39.0
Ecc code: $n = 127, d = 63$							
Full stresses	32.2	78.1	86.8	83.7	88.9	739.8	46.5
Simplified Stresses	35.1	82.9	87.0	90.1	91.5	612.7	48.9

For the experiments conducted as part of this search, the learning systems were trained on our standard 1000-word training set, but the testing was done on an *alternate test set*—disjoint from both the training set and our standard test set. Except for the Wolpert method, the input configuration will not be included as a parameter to be set as part of this search since Chapter 4 showed—rather conclusively—that "ExtContext(7-BW)" is the best input configuration for the other two methods. The result of testing the best configurations obtained as a result of this search—but trained on 19,002 words—will be postponed until Section 6.4. There, we will present these results and compare them to the performance of DECtalk's rule-base.

6.3.1 Best Configuration: Multiclass

Having established earlier that the "best" input configuration for the multiclass method—as for all the others—is "ExtContext(7-BW)", the only remaining parameters to be fixed for the multiclass approach are the window size and whether to build one decision tree that classifies phonemes and stresses taken as pairs (the combined-ps case) or to build two separate trees: one to classify stresses and the other to classify phonemes (the separate-ps case). Table 65 shows the performance of the multiclass method—on our *alternate test set*, with *simplified stresses*, and "ExtContext(7-BW)"—for both the combined-ps and the separate-ps case. The results for both window sizes compared show that the combined-ps case performs equal or better at *all* levels of aggregation and *significantly* better at the letter and word levels. Hence, we decided to drop the separate-ps case from the race.

To decide on the best value for the window size, Table 66 shows the performance of the multiclass combined-ps approach—again tested on our *alternate test set*, with *simplified stresses* and "ExtContext(7-BW)"—for several values of the parameter window size. These results show that a standard 7-letter window gives the best performance at the phoneme, stress and letter levels of aggregation. It also gives essentially the same word-level performance as that for the 15-letter

Table 65. Comparison of the performance of the combined-ps and the separate-ps multiclass methods—on our ***alternate test set***, with ***simplified stresses***, and "ExtContext(7-BW)"—for window sizes of seven and fifteen.

	Number	% correct (1000-word test set(2))				Decision Tree	
	of	Level of Aggregation				data (mean)	
Configuration	Trees	Word	Letter	Phoneme	Stress	Leaves	Depth
window size= 7:							
combined-ps:	1	26.6	79.8	83.6	88.8	1270.9	39.0
separate-ps:	2	23.0	77.6	83.4	88.5	781.4	37.4
window size= 15:							
combined-ps:	1	26.7	79.2	82.3	88.7	1174.9	36.0
separate-ps:	2	24.3	77.5	82.3	89.1	690.0	36.4

Table 66. Effect of window size on the performance of the combined-ps multiclass method—tested on our ***alternate test set***, with ***simplified stresses***, and "ExtContext(7-BW)".

	% correct (1000-word test set(2))				Decision Tree	
Window	Level of Aggregation				data (mean)	
size	Word	Letter	Phoneme	Stress	Leaves	Depth
7	26.6	79.8	83.6	88.8	1270.9	39.0
9	24.4	78.8	82.5	88.4	1207.9	36.0
11	25.3	78.9	82.4	88.2	1200.1	36.9
15	26.7	79.2	82.3	88.7	1174.9	36.0

window. Furthermore, the training time for a 15-letter window is at least twice that of a 7-letter window. We will therefore adopt a window size of 7 in our "best" configuration for the multiclass approach.

6.3.2 Best Configuration: Error-correcting Codes

In order to overcome the limitations discussed in Section 3.4.8 on the length of the stress codes, we decided to adopt the combined-ps case in the best configuration for the error-correcting code technique. This decision was also motivated by the fact that, in our earlier experiments, better performance was associated with longer codes (achieved through the combined-ps approach) having better error correcting capability. Having made that decision, the remaining parameters to be set were the length of the error-correcting code and the size of the window to be employed.

To decide on the best value for the window size to be employed, we constructed Table 67, which shows the performance of the $n = 63$, $d = 31$ (combined-ps) error-correcting code—tested on our *alternate test set*, with *simplified stresses* and "ExtContext(7-BW)"—for several values of the parameter window size. These results show that increasing the window size from 7 towards 15 gradually enhances the stress performance, while at the same time slightly degrading the phoneme performance. Stress performance seems to peak at window size = 13, but the word level performance is best for the largest window size considered.[55] We decided to adopt a window size of 15 in our best configuration for the error-correcting code technique.

To determine the length of the error-correcting code to be employed, we evaluated the performance of several *efficient*[56] BCH error-correcting codes of various lengths (Table 68). The performance for all the codes was tested on our *alternate test set*, with a window size of 15, *simplified stresses* and "ExtContext(7-BW)". While it is true that the longest code ($n = 511, d = 255$ code) attains the highest performance at all levels of aggregation, the much shorter code ($n = 127, d = 63$)

[55]Limitations on the available *virtual* storage capacity on our machines (128 mega-bytes) prevented us from considering windows larger than 15 when training on 19,002 words.

[56]We consider an error-correcting code to be *efficient* if the mininum inter-word Hamming distance, d, is roughly equal to half the length of the code, n. This enables the code to correct errors in t bits, where t is approximately one-fourth the length n.

Table 67. Effect of window size on the performance of $n = 63$, $d = 31$ (combined-ps) error-correcting code—tested on our ***alternate test set***, with ***simplified stresses***, and "ExtContext(7-BW)".

Window size	% correct (1000-word test set(2))					Decision Tree data (mean)	
	Level of Aggregation						
	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
7	33.9	82.8	86.5	90.2	91.2	638.5	50.9
9	35.0	82.7	86.2	90.3	91.2	598.8	47.9
11	35.6	83.0	86.1	90.5	91.3	581.8	48.5
13	35.9	83.0	86.0	90.8	91.3	570.7	48.5
15	36.7	83.0	86.0	90.6	91.3	568.1	49.2

does not lag very far behind. In fact the differences between the performances of the two codes are not statistically significant at any level of aggregation. Considering also that the $(n = 511, d = 255)$ code takes at least four times longer to train than the $(n = 127, d = 63)$ code,[57] it was an easy decision to adopt the $(n = 127, d = 63)$ BCH code in our "best" configuration for the error-correcting code technique.

6.3.3 Best Configuration: Wolpert

We saw in Chapter 5 that employing error-correcting output codes *marginally* improved the performance of the Wolpert method. Hence, we decided to employ the *same* $n = 127, d = 63$ BCH error-correcting code that would be employed with ID3 (discussed in the previous section) for the Wolpert method. Furthermore, we decided to employ common weights (the combined-ps weights) when computing the phoneme and the stress bits. This decision was made due to efficiency consider-

[57]It took about three CPU-WEEKS on a SUN4 work-station to build the 127 decision trees required for the $(n = 127, d = 63)$ code, trained on 19,002 words with the configuration adopted here, i.e. "ExtContext(7-BW)" with a window size of 15.

Table 68. Performance of several *efficient* BCH error-correcting codes of various lengths—tested on our *alternate test set*, with a window size of 15, *simplified stresses* and "ExtContext(7-BW)". The combined-ps approach is employed.

Code Length	Interword Distance	% correct (1000-word test set(2))					Decision Tree data (mean)	
		Level of Aggregation						
n	d (min.)	Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
63	31	36.7	83.0	86.0	90.6	91.3	568.1	49.2
127	63	37.3	83.9	86.9	91.6	92.0	550.1	45.9
255	127	37.3	83.8	86.8	91.3	91.8	557.6	46.6
511	255	38.0	84.1	86.9	91.8	92.1	551.1	47.7

ations, since the alternative—employing separate weights for computing phoneme and stress bits—would take twice as long and was not found to offer any performance improvement. Finally, all the weights employed will be those derived from the mutual information data as detailed in Chapter 5. These mutual information values were re-computed for all the elements of the extended context for the case when *simplified stresses* are employed. These are shown in Tables 93 through 98 of Appendix E.

Having made these decisions, the remaining parameters to be fixed for the Wolpert method were the number of "nearest neighbours" (nn) to be considered, the window size (w) and whether to employ a standard or an extended context. For the extended context, we will continue to use the labels "ExtContext(2-BW)" and "ExtContext(7-BW)".[58]

Let us first consider the standard context case. Table 69 shows the effect of varying the window size w on the performance of the Wolpert method—tested

[58] "ExtContext(2-BW)" refers to the case in which the input space consists of w letters, $\lfloor w/2 \rfloor$ *right* phonemes and $\lfloor w/2 \rfloor$ *right* stresses. "ExtContext(7-BW)" means that—in addition to these—the additional features defined by "ExtContext(7-BW)" are included in the input space. See Section 5.3.5 for more details.

Table 69. Impact of varying the window size on the performance of the Wolpert method—tested on our *alternate test set*, with simplified stresses, standard context and BCH error-correcting output code ($n = 127, d = 31$). Four nearest nighbours are considered in these experiments.

Window size w	% correct (1000-word test set(2))				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit
7	20.5	79.2	83.0	87.7	89.7
9	21.6	79.7	83.3	88.6	90.0
11	22.4	79.6	83.1	88.7	89.9
13	23.1	79.9	83.3	89.0	90.1
15	22.9	79.8	83.3	89.0	90.1

on our *alternate test set* and with *simplified stresses.* The best performance is observed for $w = 13$. Next, Table 70 shows the effect of varying nn—the number of nearest neighbours considered in the "center of mass" computations—when a 13-letter window is employed. $nn = 4$ seems to offer the best performance in this table. Hence, we decided to adopt these values for w and nn in subsequent tests.

Finally, to choose between standard and extended context, Table 71 compares the best performance observed with standard context with that obtained with ExtContext(2-BW) and ExtContext(7-BW). For each of these configurations, two results are shown. The first—marked by $p = 1.0$—is when the original values for the mutual information between the phonemes (stresses) in the extended context are used as weights for the corresponding phonemes (stresses) in the input space. The second—marked by $p = 0.2$—is when these values are reduced by multiplying them by p. Several values for p were tried. $p = 0.2$ seemed to offer the best performance. These results suggested that the "best" configuration to be adopted for the Wolpert method should employ ExtContext(7-BW), $w = 13$ and $nn = 4$.

Table 70. Impact of varying the number of nearest neighbours considered on the performance of the Wolpert method—tested on our ***alternate test set***, with simplified stresses, standard context and BCH error-correcting output code ($n = 127, d = 31$). The window size w is set to 13 in these experiments.

Number of neighbours considered (nn)	% correct (1000-word test set(2))				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit
3	22.4	79.4	82.9	88.7	89.8
4	23.1	79.9	83.3	89.0	90.1
5	22.0	79.7	83.2	89.0	90.0
6	22.1	79.4	83.0	88.8	89.9
7	21.1	79.2	82.7	88.6	89.8

Table 71. Performance of the Wolpert method with $w = 13$, $nn = 4$ both when standard and extended context are employed—tested on our ***alternate test set***, with simplified stresses and BCH error-correcting output code ($n = 127, d = 31$). See text for additional details.

Context employed	p	% correct (1000-word test set(2))				
		Level of Aggregation				
		Word	Letter	Phoneme	Stress	Bit
StdContext	N/A	23.1	79.9	83.3	89.0	90.1
ExtContext(2-BW)	1.0	20.4	78.0	81.6	88.5	89.2
	0.2	23.3	80.3	83.8	89.4	90.3
ExtContext(7-BW)	1.0	22.6	79.2	82.9	89.2	89.8
	0.2	25.5	80.9	84.5	89.5	90.6

6.4 Results

The best configurations determined in the previous secion for each of the three learning methods—multiclass ID3, Wolpert and the error-correcting output code method—were employed in the results reported in this section. Training was done for all the learning methods on 19,002 words—the full NETtalk dictionary less our standard 1000-word test set. These generated 139,671 examples (or equivalently, "classified windows"). Testing was done on our standard 1000-word test set—disjoint from the training set. For all the three methods, *extended context* was employed, with the letters of the words processed backwards in a right-to-left sequence. This implies that the extended context included the *phonetic context* of the letters to the *right* of the current letter. For the multiclass ID3 and the error-correcting code methods, the extended context was converted to binary using the "ExtContext(7)" representation defined in Section 4.2.3. In this representation:

- Each letter of the extended context is replaced with a 31-bit binary string.
- Each phoneme of the extended context is replaced with a 55-bit binary string.
- Each stress of the extended context is replaced with an 8-bit binary string.[59]

Hence, each 7-letter window employed in the multiclass configuration was converted to a 406-bit[60] binary feature-vector, while each 15-letter window employed for the error-correcting code method was converted to a 906-bit[61] binary feature-vector!

For the Wolpert method, ExtContext(7-BW) meant that the same information conveyed by the *extra* bits in ExtContext(7) was made available in the repre-

[59]With the stresses simplified to only three symbols representing primary, secondary and no stress, we could have reduced the ExtContext(7) representation for each stress to an equivalent representation with only 4 bits instead of 8. This reduction was minor and was not carried out in these experiments.

[60]7 letters $\times 31 + 3$ phonemes $\times 55 + 3$ stresses $\times 8$.

[61]15 letters $\times 31 + 7$ phonemes $\times 55 + 7$ stresses $\times 8$.

sentation of the windows, and that the letters of the words were processed in a right-to-left order. Hence, the 13-letter *extended context* employed for this method meant that each window consisted of a 63-character string as follows:

- 13 characters (each $\in \mathcal{A}$, where $\mathcal{A}$ is the set containing the 26 letters and the three symbols "-", "_", and ".") representing the 13 letters in the window.
- 13 characters (each $\in$ {0,1}) indicating whether each of the 13 letters in the window is a consonant, (0), or not, (1).
- 13 characters (each $\in$ {0,1}) indicating whether each of the 13 letters in the window is a non-low vowel (1), or not, (0).
- 6 characters (each $\in \mathcal{P}$, where $\mathcal{P}$ is the set of 54 phonemes defined earlier) representing the 6 *right* phonemes which are part of the *extended context.*
- 6 characters (each $\in$ {0,1}) indicating whether each of the above *right* phonemes is tense, (1) or not, (0).
- 6 characters (each $\in \mathcal{S}$, where $\mathcal{S}$ = `{1,2,-}` is the set of 3 stress symbols representing the simplified stresses defined earlier) representing the 6 *right* stresses which are part of the *extended context.*
- 6 characters (each $\in$ {0,1}) indicating whether each of the above *right* stresses is "-", (0) or not, (1).

The above representation of the windows for the Wolpert method made the classification of the words in the test set extremely costly. A typical word generates an average of seven windows. To classify each window w, the Wolpert method requires that the 63-character string representing w be compared with each of the 139,671 63-character strings representing classified windows generated from the training set. This expensive nearest neighbours matching process resulted in a

Table 72. Comparison of the performance of the best configurations of three learning systems with the performance of DECtalk's rule-base on our standard 1000-word test set.

System and Configuration	Window Size	% correct (1000-word test set)					Decision Tree data (mean)	
		Level of Aggregation						
		Word	Letter	Phoneme	Stress	Bit	Leaves	Depth
ID3 & Error Correcting Codes	15	64.8	91.4	93.7	95.1	95.7	5678	71.2
Wolpert & Error Correcting Codes	13	45.6	88.1	92.7	91.9	94.0	N/A	N/A
Multiclass ID3 (combined-ps)	7	44.1	86.8	91.5	91.3	N/A	13,703	65.1
DECtalk's rule-base	N/A	36.4	83.3	87.4	91.8	N/A	N/A	N/A

classification time of about 15 minutes for a typical word.[62] By contrast, the other (decision tree) methods take only a fraction of a second to classify a new word, but they require a much longer time to build the decision trees.

Table 72 shows the performance of the three learning systems—with the configurations detailed above—along with the performance of DECtalk's rule-base on our standard 1000-word test set.

Let us first concentrate on the three learning methods compared. Of these three, multiclass ID3's performance is the worst at all levels of aggregation. Wolpert's performance is slightly better, especially at the phoneme and letter levels. The most impressive performance of all is that of ID3 with error-correcting codes. This method outperforms the other two by a wide margin, specially at the stress and word levels. The stress level performance of 95.1% is a significant improvement on the 91.3% (91.9%) performance of the multiclass (Wolpert) methods. In terms of

[62]Techniques for speeding-up nearest neighbour searches are available (See [Bentley75], for example), but they were not employed in our implementation of the Wolpert method.

error rates, this reflects a reduction in the stress error-rate from 8.7% (8.3%) to only 4.9%, a reduction of 43.7% (39.5%) over the multiclass (Wolpert) methods. Similarly, the correctness figure of 64.8% at the full-word level is a radical improvement over the 44.1% (45.6%) achieved by the multiclass (Wolpert) methods. In terms of error rates, these represent a reduction of 37.0% (35.3%) over the multiclass (Wolpert) methods. Errors at the phoneme and letter levels are also reduced. Phoneme error rates are reduced by 25.9% (13.7%), while letter error rates are reduced by 34.8% (27.7%) compared to the multiclass (Wolpert) methods.

Even though the results presented in Table 72 clearly indicate that ID3 when coupled with error-correcting output codes outperforms the other learning methods in this domain, one may still wonder whether or not this method subsumes the other two. To answer this question, we decided to study exactly how each of the 7,242 windows (corresponding to letters to be mapped to phoneme/stress pairs) in the test set are handled by each of the learning methods. Table 73 categorizes each of these windows according to whether it was correctly classified by all methods, by only two of the methods, by only one of the methods, or by none of the methods. These data are depicted pictorially in Figure 6, in which each of the methods is represented by a circle. The numbers shown in the intersection of all the circles represent the windows classified correctly by all the methods, those shown in the intersection of only two circles represent the windows classified correctly the corresponding two methods, those appearing in only one circle represent the windows classified correctly only by that method and none of the others. The windows not classified correctly by any of the methods appear outside all the circles but within the rectangular boundary encompassing all the 7,242 windows present in the test set.

These results show that the windows correctly learned by the error-correcting code method do not form a superset of those learned by either of the other two methods. Instead, the three methods share 5,939 correct windows, and then each method correctly classifies several windows that the other method (or methods)

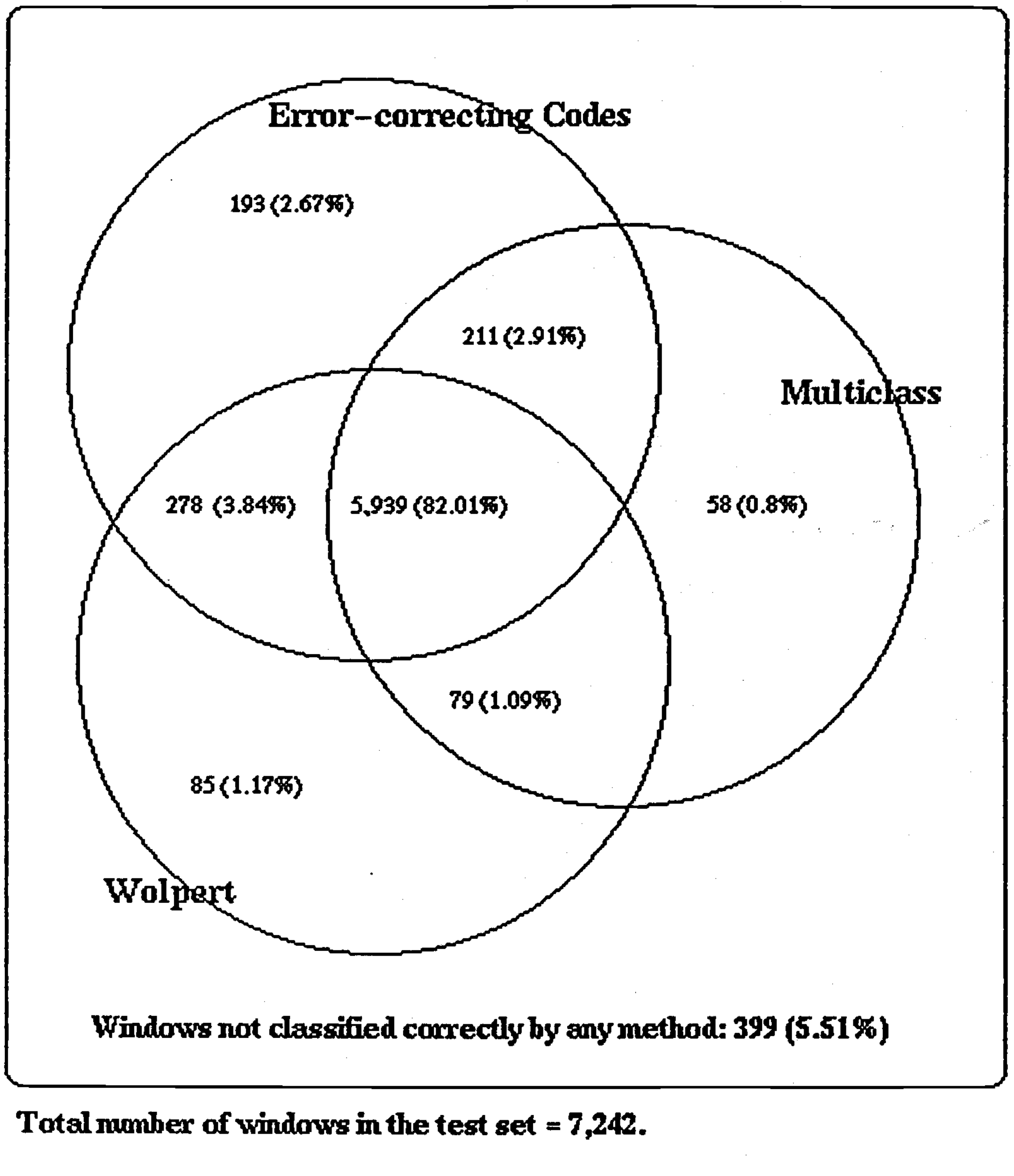

Figure 6. Classification of test set windows by three learning methods: ID3 with error-correcting codes, Wolpert and Multiclass ID3.

get(s) wrong. However, the number of windows correctly classified by the error-correcting code method and missed by the other two is 193 compared to corresponding numbers of 58 (respectively 85) for the multiclass (respectively, the Wolpert) method.

An interesting "thought" experiment can be performed to determine whether employing voting among the three methods would lead to any improvement in the performance at the letter level. With voting, we would correctly classify the 5,939 windows in the shared region among all the circles and the windows in the three areas common to any two circles, i.e. 211 + 278 + 79 additional windows. With regard to the areas where only one method classifies the windows correctly, there are two possibilities. The first possibility is that when one method classifies the window correctly and the other two misclassify it, they map it to the *same* wrong output. In this case the voting will force this window to be misclassied and hence none of these windows should be added to the number of successfully classified windows. The other possibility is that when one method classifies the window correctly and the other two misclassify it, they map it to a *different* wrong output. In this case, there will be a 3-way tie, which we can assume will be broken by choosing one output at random. Since there is a 1 in 3 chance of randomly selecting the *correct* output, we could add to the number of successfully classified windows as much as one-third of the windows correctly classified by only one method. This, at best, will add 112 additional[63] windows. The total windows correctly classified with voting will amount to 6,619 windows (at best), or, 91.4% of all the 7,242 windows present in the test set. Compare this with the performance of the best method—the error-correcting code method—taken alone, which amounts to adding the four figures appearing in the circle corresponding to that method: $193+278+5,939+211 = 6,621$, or 91.4% of the total windows. Hence, even voting across methods does not outperform the performance of the error-correcting code

[63]58 for Multiclass + 85 for Wolpert + 193 for the error-correcting code method ÷3.

Table 73. Classification of test set windows by three learning methods: ID3 with error-correcting codes, Wolpert and Multiclass ID3.

System correctly classifies all w windows			Number of	% of total
ID3 & ECC	Wolpert	Multiclass ID3	Windows (w)	(7,242 windows)
NO	NO	NO	399	5.51
NO	NO	YES	58	0.80
NO	YES	NO	85	1.17
NO	YES	YES	79	1.09
YES	NO	NO	193	2.67
YES	NO	YES	211	2.91
YES	YES	NO	278	3.84
YES	YES	YES	5,939	82.01

method taken alone!

Let us now turn our attention to the performance of DECtalk's rule-base. The dismal word level correctness figure of 37% should tell us that our correctness criteria are too stringent. We consider a word to be mispronounced if *any* of its letters were mapped to *either* an incorrect phoneme *or* an incorrect stress—regardless of the seriousness of the errors committed. This criterion, however stringent, is nevertheless straight forward and very objective. By contrast, previous evaluations of text-to-speech systems reported only "serious" error rates. Table 74 shows error rates as relayed to us by [Vitali90] for three commercial text-to-speech systems based on this—rather subjective—measure. It is unfortunate that we can not go much beyond simply reporting these figures, since we do not know precisely what constitutes a *serious* error and what does not.

It is surprising that *all* the three learning methods outperform DECtalk's rule-base at the phoneme, stress, letter and word levels of aggregation. While one can argue that the lower phoneme level performance for DECtalk[64] could be attributed

[64]The particular version of DECtalk that is referred to throughout this chapter is DECtalk3.

Table 74. Serious error-rates for three commercial text-to-speech systems based on a 1987 study by Bill Huggins at BBN (Bolt, Baranek, and Newman). The test was over a set of 1,678 polysyllabic inflected forms, like "reinforce".

System	"Serious" Error Rate
DECtalk2	12.9%
Speech+ calltext 3	8.3%
DECtalk3	5.1%

to the manner in which DECtalk's phonetic rules were encoded,[65] no such argument can be made in the case of the stresses. The stress error rate of 4.9% for the best learning method represents a reduction of 40.2% from the 8.2% error rates committed by DECtalk's rules for determining the stresses. This unambiguously suggests that the machine learning techniques developed as part of this research work compete rather well with human-engineered rules for the text-to-speech domain.

A four way detailed categorization of how each of the four systems classifies each of the 7,242 "windows" in the test set is shown in Table 75. These data show that while there are 280 windows (3.87%) correctly classified by DECtalk's rule-base that are missed by all the three learning methods, there are 935 windows (12.91%) correctly classified by all the three learning methods that are misclassified by DECtalk's rule-base. The four systems agree on 5,123 windows (70.74%); they all correctly classify 5,004 windows (69.10%) and all misclassify only 119 windows (1.64%).

Table 76 narrows the comparison down to two: DECtalk's rule base and the error-correcting code learning method. Here, the learning method correctly clas-

[65]These rules may be biased towards some particular *accent* or way of pronunciation of certain words which may not be entirely wrong but nevertheless different from the way the NETtalk dictionary encodes these pronunciations. This point will be re-visited at a later stage.

Table 75. Classification of test set windows by the three learning methods—ID3 with error-correcting codes, Wolpert and Multiclass ID3—and DECtalk's rule-base.

System correctly classifies all w windows				Number of	% of total
ID3 & ECC	Wolpert	Multiclass ID3	DECtalk rules	Windows (w)	(7,242 windows)
NO	NO	NO	NO	119	1.64
NO	NO	NO	YES	280	3.87
NO	NO	YES	NO	11	0.15
NO	NO	YES	YES	47	0.65
NO	YES	NO	NO	19	0.26
NO	YES	NO	YES	66	0.91
NO	YES	YES	NO	14	0.19
NO	YES	YES	YES	65	0.90
YES	NO	NO	NO	25	0.35
YES	NO	NO	YES	168	2.32
YES	NO	YES	NO	37	0.51
YES	NO	YES	YES	174	2.40
YES	YES	NO	NO	50	0.69
YES	YES	NO	YES	228	3.15
YES	YES	YES	NO	935	12.91
YES	YES	YES	YES	5,004	69.10

Table 76. Classification of test set windows by the ID3 & error-correcting codes learning method and DECtalk's rule-base.

		Error-correcting Codes		
		Correct	Incorrect	
DECtalk's Rule-base	Correct	5,574 (76.97%)	458 (6.32%)	Disagree: 1,505 (20.78%)
	Incorrect	1,047 (14.46%)	163 2.25%	Agree: 5,737 (79.22%)

sifies 1,047 windows (14.46%) that are missed by DECtalk's rule-base, while the latter correctly classifies 458 windows (6.32%) that are missed by the learning method.

6.5 A Human Study Comparison

The criteria for correctness employed in the comparative study of the previous section were very strict: the phoneme and stress classes were required to match *exactly*. A little thought shows that this is overly strict. Some phoneme errors (e.g., substituting /k/ for /e/) are very serious, while others (e.g., substituting /x/ for /@/) are virtually indistinguishable in the pronunciation of some words. Similarly, some stress errors (e.g., substituting *primary* stress with *no* stress) are generally much more serious than other stress errors (e.g., substituting *primary* stress with *secondary* stress). Hence, while these comparisons are valuable, the performance figures shown in the previous section do not give a good indication of the quality of the speech produced by the learning systems relative to that produced by DECtalk's rule-base.

In the next subsection, we present results of a different kind of comparative study: one in which the correctness criteria were relaxed to consider pronunciations that sound similar to human subjects. These results give further support to the validity of the previous *strict* comparison and show that the machine learning systems considered outperform the human-developed letter-to-sound rules incorporated in DECtalk's rule-base *regardless* of the correctness criteria employed.

6.5.1 Evaluations by Human Subjects: The procedure

In order to get a good indication of the quality of the speech produced by the learning systems—considered in this chapter—relative to that produced by DECtalk's rule-base, we decided to let human subjects judge the "correctness" of the pronunciation of each system when played through DECtalk's hardware.[66]

For each of the methods considered, three subjects were asked to compare how close the pronunciation output by the method—for each of the 1000 words in our standard test set—compared with the correct (NETtalk dictionary) pronunciation.[67] The subjects were allowed to listen to the correct (NETtalk dictionary) pronunciation of each word followed by the method's pronunciation. Then, they were asked to select among one of the following responses:

1. There is *No* difference between the two pronunciations, or,
2. There is a *Noticeable* difference between the two pronunciations, or,
3. There is a *Serious* difference between the two pronunciations, or,
4. The method's pronunciation is better than the dictionary pronunciation. (Therefore, count the method's pronunciation as correct.)

[66]The DECtalk machine optionally allows passing the *Klattized* format of the pronunciations (see Section 6.1.1) of the words through the speech synthesis hardware, by-passing DECtalk's dictionary and rule-base.

[67]If the method's pronunciation (phoneme/stress strings) matched the dictionary *exactly*, the word was *automatically* considered correct.

Each of the three subjects was free to respond with any of the above, so the responses had the form: 111, 122, 233, etc. Majority vote was taken, so that 122 would be counted as 2, 233 would be counted as 3, and so on.

The procedure described above would give us a basis for evaluating errors at the word level but not at the individual phoneme, stress, or, letter levels. To get a subjective evaluation of the performance at these lower levels of aggregation, we extended the above experimental procedure to count the individual phoneme/stress errors as follows:

1. If the word-response was 1 or 4, the Phonemes and Stresses corresponding to *All* the letters of the word was taken as correct.
2. Otherwise (i.e. word-response was 2 or 3):
    - The method's phoneme/stress list was compared with the dictionary phoneme/stress list to find all the errors (differences). Then,
    - The dictionary phoneme/stress list was mutated with one error at a time (either a phoneme or a stress error), and the resulting pronunciation was presented to the subjects (through DECtalk's hardware as before) for them to judge the seriousness of the particular phoneme or stress error.
    - Acceptable responses were the same as explained in the previous paragagh.

Table 77 illustrates the second and third steps of the above procedure, applied to the pronunciation of the word "AFFIDAVIT". Note that in this example, the subjects were asked to judge seven additional (hypothetical) pronunciations of "AFFIDAVIT" in order to determine the severity of each of the seven individual phoneme/stress errors within their present context.

Table 77. Format of the pronunciations presented to human subjects (through DECtalk hardware) for evaluating the individual phoneme/stress errors in the word "AFFIDAVIT".

Pronunciation:	Original		Presented to Subjects	
Pronunciation Source	Phonemes	Stresses	Phonemes	Stresses
Method:	`xf-IdxvIt`	`---1-----`	`xf-IdxvIt`	`---1-----`
Dictionary:	`@f-xdevxt`	`2----1---`	`@f-xdevxt`	`2----1---`
Dict. + Error (1):	`@→x`		`xf-xdevxt`	`2----1---`
Dict. + Error (2):	`x→I`		`@f-Idevxt`	`2----1---`
Dict. + Error (3):	`e→x`		`@f-xdxvxt`	`2----1---`
Dict. + Error (4):	`x→I`		`@f-xdevIt`	`2----1---`
Dict. + Error (5):		`2→-`	`@f-xdevxt`	`-----1---`
Dict. + Error (6):		`-→1`	`@f-xdevxt`	`2--1-1---`
Dict. + Error (7):		`1→-`	`@f-xdevxt`	`2--------`

6.5.2 Evaluations by Human Subjects: Results

Responses from the 3 subjects were gathered—as described in the previous section—for errors committed by each of the four methods on the 1000 words in our test set, and for the individual phoneme/stress errors in the 7,242 corresponding letters in these words. The results are compiled in Tables 78 and 79. Table 78 shows the performance figures if an error is counted when the majority of the subjects' responses was either 2 (noticeable difference) or 3 (serious difference). Table 79, on the other hand, shows the performance figures if an error is counted *only* when the majority of the subjects' responses was 3: serious difference. In both tables, the column marked with "Direct Words Correct" shows performance figures directly computed from the subjects' responses reflecting how close the method's pronunciation (for the full word, of course) was to the correct pronunciation, independent from their responses reflecting the seriousness of the individual phoneme or stress errors taken in isolation. The other four columns, on the other hand, are *all* derived from the subjects' responses reflecting the severity of the individual phoneme/stress errors

Table 78. Results of the human subject comparative study if an error is counted when the majority of the subjects responses was either 2 (noticeable difference) or 3 (serious difference). See text for details.

System and Configuration	Direct Words Correct	% correct (1000-word test set)			
		Level of Aggregation			
		Word	Letter	Phon.	Stress
ID3 & Error Correcting Codes	71.2	72.9	93.9	95.1	97.3
Wolpert & Error Correcting Codes	56.7	61.2	92.1	94.4	95.8
Multiclass ID3 (combined-ps)	57.0	60.6	90.9	93.2	95.6
DECtalk's rule-base	55.7	58.5	90.1	91.7	96.5

taken in isolation.

There are several things to note in Tables 78 and 79. First, comparing both tables with Table 72 for the strict evaluation case, we find that the performance of *all* the methods improves (by varying degrees) when the correctness criteria are relaxed. However, the *order* of the performance of the methods relative to one another remains essentially unchanged. Second, the methods that commit the most mistakes are—predictably—the ones that benefit the most from the increasingly more forgiving correctness criteria employed in these latter tables.

Next, let us focus our attention on the differences between the "direct words" correctness figures and the word-level correctness figures computed from the subjects' responses reflecting the severity of the *individual* phoneme/stress errors taken in isolation. Note that these differences are more pronounced in Table 79 than in Table 78. This is understandable, since multiple phoneme and/or stress errors in the same word might interact in a manner that influences the subjects to decide that the pronunciation of the word is *seriously* different from the correct pronun-

Table 79. Results of the human subject comparative study if an error is counted *only* when the majority of the subjects responses was 3: serious difference. See text for details.

System and Configuration	Direct Words Correct	% correct (1000-word test set)			
		Level of Aggregation			
		Word	Letter	Phon.	Stress
ID3 & Error Correcting Codes	83.3	85.3	97.6	97.7	99.8
Wolpert & Error Correcting Codes	79.5	84.6	97.5	97.7	99.6
Multiclass ID3 (combined-ps)	76.2	81.1	96.7	97.0	99.5
DECtalk's rule-base	73.9	81.6	97.1	97.2	99.7

ciation. However, when each of the errors is listened to in isolation, the subjects may judge *all* the individual errors as *noticeable* rather than *serious*.

Another puzzling observation is the stress level correctness figures in Table 79. It is striking that the subjects hardly ever considered a stress error as *serious*, contrary to the general notion held by domain experts. This could be partially explained by the personal observation of the author that the subjects were extremely hesitant—in general—to label errors as serious. Hence, it is the opinion of the author that the performance figures shown in Table 78 are more credible than those shown in Table 79. Another possible explanation—for the extremely low *serious* stress errors—might be due to the fact that only first order effects of each error (in isolation) were being judged by the subjects. For the case of stresses, this often meant judging the effect of having primary stress placed on the nucleus of *two* syllables of the word instead of just one (see the row marked "Dict. + Error (6)" in Table 77 as an example). This may not be as serious as *shifting* the primary stress to a new syllable in the word.

Table 80. Breakdown of the individual subject responses when evaluating correctness at the word level. Figures shown reflect the cumulative responses for all the four systems compared.

Majority Response	Individual Responses*	Actual Count	% of Total	% of *all* 2,088 Responses
(1) No difference:				
1	111	426	86.2%	
1	112	68	13.8%	
1	113	0	0.0%	
	Total:	494	(100%)	23.66%
(2) Noticeable difference:				
2	221	87	12.0%	
2	222	505	69.9%	
2	223	131	18.1%	
	Total:	723	(100%)	34.63%
(3) Serious difference:				
3	331	2	0.2%	
3	332	114	13.1%	
3	333	755	86.7%	
	Total:	871	(100%)	41.71%

*Example: 332 means two subjects responded with 3, and the third responded with 2.

To summarize, one can conclude that the results of the above—rather subjective—study give further support to the validity of the *strict* comparison presented in earlier sections. They show that the machine learning systems considered outperform the human-developed letter-to-sound rules incorporated in DECtalk's rule-base *regardless* of the correctness criteria employed. The effect of relaxing the correctness criteria is to merely *blur* the differences between the methods, especially the differences between the learning systems.

6.5.3 Inter-subject Agreement Data

As with any study that involves human subjects, the data presented in the previous section are *necessarily* dependent (to some degree) on the particular subjects involved in that study. To get an idea as to the extent of this dependency, Table 80 shows the breakdown of the individual responses obtained from the three subjects when evaluating correctness at the word level. The figures shown in the table reflect the cumulative responses obtained for all the four systems compared.[68] The results are encouraging. Out of the 494 responses that were classified as (1), (i.e. no difference), 426 (86.2%) were by a unanimous decision. Similarly, out of the 871 responses that were classified as (3), (i.e. serious difference), 755 (86.7%) were by a unanimous vote.

The situation, however, is not so clear cut for responses classified as (2), (i.e. noticeable difference). Only 505 of the 723 such cases (69.9%) were by a unanimous decision. The rest (218 cases), by a vote of two out of three. In 131 out of these 218 cases (60.1% of the time), the third subject who disagreed thought that difference is serious rather than noticeable, while in the remaining (39.9% of the 218 cases) the subject who disagreed thought that difference is negligible.

It is interesting that there were *no* cases of the "113" response (two subjects responding with "1", the third with a "3"), only 2 cases of "331", and only one case of a "123" response (which was later changed to a "222"). Furthermore, all the three subjects agreed on 1,686 responses[69] out of 2,088 (80.7%). These observations suggest that the results of the human study presented in the previous section are fairly robust against variations caused by differences between individual human subjects.

[68]The responses add up to 2,088 only instead of 4,000 (1000 words in our test set × 4 systems compared with the dictionary). The "supposedly" missing 1,912 responses represent cases in which the pronunciation output by each system was exactly matching that of the dictionary. As explained in Section 6.5.1, these were by-passed and were automatically assumed to be correct.

[69]426 + 505 + 755 from Table 80.

6.6 Robustness of the Error-correcting Code Approach

An interesting point that is often raised by proponents of neural networks is the networks' resistance to "brain damage" and the gradual degradation in performance they exhibit as more and more of the weights are altered or "corrupted". In this section we show that the same kind of behavior can be observed with ID3 trees when error-correcting codes are employed to encode the classes.

For this experiment, we used our best performing configuration with the $n = 127, d = 63$ BCH error-correcting code employed in Section 6.4. To simulate "brain damage", we would replace a portion of the 127-trees with a random number generator that flips a coin to decide whether that bit is to be classified as 0 or 1. The portion that would be randomized was controlled by a parameter *random-portion.* Which bit positions to randomize was also determined randomly. The algorithm shown in Table 81 was used to return a bit-vector, $\vec{v}$, of the same length as the number of trees used. The classification routine would then check bit_i of this bit vector $\vec{v}$ to decide whether $tree_i$ should be used to determine this bit (if bit_i is 1 $\vec{v}$) or whether the bit should be randomized.

We employed the above procedure and increased *random-portion* from 0.05 to 0.9 in steps of 0.5. The results are shown in Table 82 and graphed in Figure 7. Notice that the performance holds fairly stable until about 50% of the trees are randomized, then it starts declining. This is not surprising, since the particular output code employed is capable of correcting errors in about one fourth of the output bits, or 32 bits. Each randomized bit has a 50% chance of being correct, so with 50% of the bits randomized, only one-fourth will be wrong. Hence the code still performs fairly well. Beyond this point, a sharp drop in the performance is observed, as it becomes impossible to correct for the overwhelming number of output bits that are in error. While these results are completely predictable, they show the extent to which error-correction could go should the individual bit-error rates be truly independent from one another.

Table 81. Algorithm *classify-or-randomize.*

```
algorithm: classify-or-randomize
input: num-trees, random-portion
output: bit-vector of length num-trees
begin {classify-or-randomize }
  Avg-num-trees-to-randomize := num-trees × random-portion
  generate a bit-vector bit-vector of length num-trees
  for $i = 1$ to num-trees do
    begin
      set random-number = generate a random number
                          betweeen 0 and num-trees less 1
      if random-number < Avg-num-trees-to-randomize then
        set $bit_i$ bit-vector of to 0; {randomize}
      else
        set $bit_i$ bit-vector to 1; {classify}
    end;
  Return the bit-vector;
end {classify-or-randomize }
```

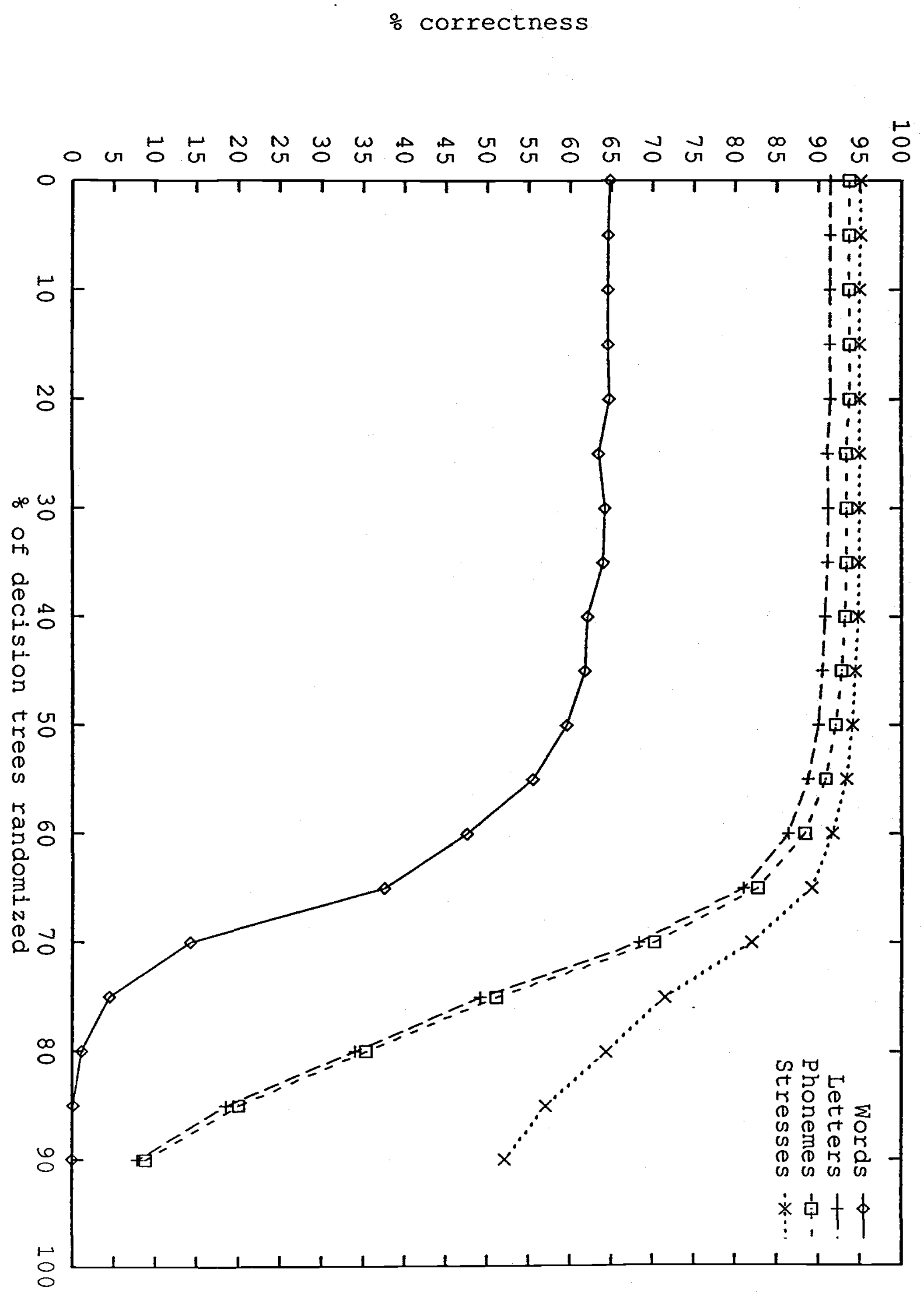

Figure 7. Performance of ID3 and error-correcting codes when a certain portion of the trees is randomized.

Table 82. Effect of damaging various portions of the decision trees on the performance of error-correcting output codes.

Portion of the trees Randomized (%)	% correct (1000-word test set)				
	Level of Aggregation				
	Word	Letter	Phoneme	Stress	Bit (maen)
5	64.6	91.4	93.7	95.1	95.8
10	64.6	91.4	93.7	95.0	95.8
15	64.6	91.4	93.8	95.0	95.8
20	64.8	91.5	93.8	95.0	95.8
25	63.5	91.1	93.4	95.0	95.6
30	64.2	91.2	93.4	94.9	95.7
35	64.0	91.1	93.4	94.9	95.6
40	62.1	90.8	93.2	94.8	95.5
45	61.8	90.5	92.8	94.4	95.3
50	59.6	90.0	92.1	94.1	95.1
55	55.5	88.8	90.9	93.4	94.5
60	47.6	86.4	88.4	91.7	93.3
65	37.5	81.0	82.7	89.2	90.7
70	14.3	68.4	70.3	82.0	84.4
75	4.5	49.2	51.1	71.6	75.0
80	1.1	34.0	35.3	64.4	67.5
85	0.1	18.6	20.1	57.1	60.0
90	0.0	7.9	8.9	52.2	54.7

Chapter 7

Conclusions and Future Work

In Chapter 1 we formulated two objectives that we were striving to achieve at the outset of this research: a main objective and a domain objective. The main objective was to develop machine learning algorithms and/or techniques for the task of learning discrete multi-valued functions from examples (multiclass learning); techniques that are general, efficient and outperform existing methods for this task. The domain objective was the automatic generation of a high performance rule base that can compete with current expert systems for the task of converting isolated English words to strings of phonemes and stresses.

We have achieved *both* of these objectives.

The following two sections will address the contributions of this research to both of the above tasks. Section 7.3 will then discuss several directions that can be pursued for extending the work presented in this dissertation.

7.1 Multiclass Learning

A theme underlying this thesis has been that error-correcting output codes provide an excellent method for applying binary learning algorithms to multiclass learning problems. In particular, we have demonstrated that the standard ID3 algorithm coupled with error-correcting output codes outperforms the direct multiclass method, the one-per-class method, a domain-specific distributed output code (the Sejnowski-Rosenberg code), and the Wolpert method in the NETtalk domain. These results were shown to hold regardless of the feature set employed to encode the context in this domain.

Furthermore, in [Bakiri91a], we show that error-correcting output codes also improve performance in a very different domain—the isolated letter speech recognition task [Cole90, Lang90]—and with a quite different learning algorithm: the Backpropagation algorithm used to train connectionist networks. These additional results suggest that error-correcting output codes provide a domain-independent, algorithm-independent approach to multiclass learning problems.

We also demonstrated that codes generated at random can act as excellent error-correcting codes. Indeed, random codes provide an attractive alternative to BCH methods in practical applications. However, regardless of the method employed to generate the code, it is important to ensure that no two columns of the code are complementary.

We discussed several limitations of the error-correcting output coding technique and demonstrated that, for it to be effective, errors in the individual bit positions of the error-correcting codes employed should not be highly correlated with one another. This explained why error-correcting codes offered only a marginal improvement when combined with the Wolpert method. The fact that all the bit values of each output code are computed from the same four nearest neighbours in the Wolpert method results in a high value for these bit-error correlations, rendering the method ineffective.

Finally, we showed that the error-correcting code method is superior to the approach of generating multiple hypotheses and employing some form of voting among them.

Let us leave this section with the thought that error-correcting codes may explain part of the success of biological neural networks. If these networks employ distributed representations, some random way of generating the codes of such representations is biologically plausible, and we have shown that such randomly generated codes are in fact excellent error-correcting codes!

7.2 The Text-to-Speech Domain

For the task of mapping isolated English words to strings of phonemes and stresses representing their pronunciations, we have demonstrated that machine learning techniques can compete effectively with human-developed letter-to-sound rules. The best machine learning system that we developed substantially outperformed the expert-developed letter-to-sound rules incorporated in DECtalk[70] in all the comparative tests we performed. These tests included comparing performance results at the phoneme, stress, letter and word levels of aggregation both when the correctness criteria required a strict (exact) match and when the correctness criteria were relaxed to consider pronunciations which differed but *sounded similar* to human subjects. This is a significant finding (in our opinion), as it opens the door to applying similar machine learning techniques to the task of discovering pronunciation rules in other languages. We expect that only a few months would be required to apply machine learning methods to build a text-to-speech system for a new language. In contrast, we would expect that several years would be needed to construct an expert rule-based system for each new language. One disadvantage of the "rules" derived through machine learning techniques is that they are too complex to be understandable by human experts.

In addition to our main contribution described above in the English text-to-speech domain, other (smaller) contributions stem from the fact that during the course of our investigations in this domain, we replicated the efforts of several other researchers who had previously tackled this task, but we applied a *uniform* yard-stick to evaluate the effectiveness of the techniques they proposed and/or employed. We replicated the original works reported in [Sejnowski87], [Shavlik89],

[70]At the time the comparison was made, the version of the software (rule-base) was referred to as DECtalk3.

[Lucassen83], and [Wolpert90c], but with all the systems trained with our standard 1000-word training set and tested on our standard (disjoint) 1000-word test set. Some of our findings as a result of these replications will be described in the following paragraphs.

Early in this project, we replicated the work of [Sejnowski87] and [Shavlik89] and compared the performance of the ID3 and the Backpropagation (BP) algorithms in this domain. In [Dietterich90a,b] we investigated several possible sources for the differences observed between ID3 and BP, and concluded that the ability of the numerical parameters in the BP network to capture statistical information—not captured by ID3—was the main reason for the observed differences in the performance of BP and ID3 in this domain. We showed that augmenting ID3 with improved decoding strategies such as *observed* or *block* decoding brings the performance of the two algorithms to nearly the same levels.

One weakness of the work reported in [Lucassen83] was that it presented the performance results for one particular choice of input and output representations. In our work, we reproduced their results (as closely as feasible) under standard conditions that could be easily compared with other methods that we investigated. We showed, for example, that their elaborate method for selecting "good" binary features to represent the input was inferior to the simple choice of selecting a local encoding (i.e. weight-1 codes) for the letters, phonemes and stresses used as part of the extended context. Similarly, we presented in Chapter 3 several higher performance alternatives to the direct multiclass approach employed in their study. We also showed that their decision to incorporate the phonetic context of the previous letters in the context for the current letter actually *hurt* the performance of their system when the *left* phonetic context was employed. On the other hand, we found that their decision proved to be a judicious one when combined with processing the letters of the words backwards—in a right-to-left sequence. Including the *right* phonetic context in this manner offered a substantial improvement in the overall performance—mainly at the stress level—in this domain. This finding is consistent with what the linguists have suspected all along: that the stress patterns in English words can best be described by rules that refer to certain properties of the syllables of the words taken in a *right-to-left* order [Halle71].

We also implemented Wolpert's method and found that Wolpert's claim that his method outperformed Sejnowski's NETtalk were exaggerated—given the experimental data available to him at the time. On the other hand, we showed that with the improved weights that we employed, his method did outperform the basic ID3 algorithm and to a lesser extent the Backpropagation algorithm on this task. Furthermore, we investigated several enhancements to the Wolpert method. First, we showed that a good assignment for the required "weights" in the method could be made by setting the weight for each letter position j in the w-letter window equal to the *mutual information* between the letter at position j in the window and the output (pronunciation of the current letter). Next we enhanced the method by incorporating error-correcting codes on the output side and *extended context* on the input side. We found that despite the superiority of the basic method over ID3 in this domain, the Wolpert method was not able to utilize the extended context as effectively as ID3. It was also unable to capture the complex patterns required to learn the stresses, and—for reasons briefly touched upon in the previous section—was *marginally* able to take advantage of the performance boost normally offered by the error-correcting output code technique to other learning algorithms.

7.3 Future Work

There are several directions that can be pursued for extending the work presented in this thesis. We will first cover (in the next section) several suggestions for future work related to the general technique of applying error-correcting codes in the field of machine learning. Next, in Section 7.3.2, we present several directions that can be explored for improving the performance of machine learning algorithms in the English text-to-speech domain. Future work in other related domains are briefly considered in Section 7.3.3.

7.3.1 Error-correcting Codes

Despite the fact that error-correcting codes have long been applied in the communications field, their application in machine learning poses additional requirements mainly because attention must be paid not only to the separation between the codewords (i.e. *rows* of the codeword matrix) but also between the *columns* of the codeword matrix, which correspond to the boolean concepts that need to be learned. Having identical columns or even columns which are complimentary to one another in the codeword matrix would result in errors that are 100% correlated with one another—thereby reducing the effectiveness of the error-correcting codes employed.

Even though the BCH method we employed for generating error-correcting codes generate 2^k codewords which do *not* have identical or complimented columns, the process of selecting only a subset of these for coding the C classes that need to be learned could potentially introduce the identical or complimented column problem in the subset of the codewords selected. Coding theory techniques *are* available that allow the generation of a specified number, C, of codewords with good row *and* column separations. However, more work is needed to encapsulate these ideas in a straight forward algorithm that could be readily employed by machine learning researchers.

Another area that can be the subject of future investigation is determining the "optimum" length for the codewords to be employed in any particular multiclass learning situation. To tackle this problem, one has to have an idea about the average values for bit-error rates and inter-bit correlations. These could be measured by preliminary tests on random codes generated for this purpose. The question would then be: Given these values, how long a code will still improve performance?

A third area for future work—on the application of error-correcting codes in the text-to-speech domain—is investigating the possibility of correcting single letter errors. In the communications field, there is a technique called ***block transmission*** used for correcting ***burst errors*** during transmission.[71] It would be intriguing to come up with a similar technique for output encoding that will allow us to correct for single letter errors within a word—the letter in question corresponding to a burst error. If successful, this scheme can then be combined with the normal error-correcting coding techniques to come up with a code that combines the power of correcting bit errors with that of correcting single letter errors. Another possible approach to correcting single letter errors is to employ Reed-Solomon block codes [Lin83].

Finally, employing error-correcting codes in machine learning applications has proven to be extremely successful in the two domains that we have studied (Converting English text to speech, and the the isolated letter speech recognition task) and for two learning algorithms (ID3 and Backpropagation). Further work is needed to extend this technique to other domains, and to employ it with other known boolean learning algorithms.

7.3.2 The Text-to-Speech Domain

There are several directions that can be explored for further improving the performance of learning systems on the task of converting English text to speech. In the following discussion, we will first try to point out some of the special difficulties encountered in this domain, and then we will propose some means of overcoming these difficulties.

[Klatt87] points out three properties of the domain that present special challenges to inductive learning methods:

(1) the considerable extent of letter context that can influence stress patterns in a long word (and hence affect vowel quality in words like

[71] A burst error is a transmission error that corrupts a sequence of *consecutive* bits rather than bit positions at random.

"photograph/photography"), (2) the confusion caused by some letter pairs, like CH, which function as a single letter in a deep sense, and thus misalign any relevant letters occurring further from the vowel, and (3) the difficulty of dealing with compound words (such as "houseboat" with its silent "e"), i.e., compounds act as if a space were hidden between two of the letters inside the word.

Besides the above problems, several additional features are necessary to discriminat alternative pronunciations of identical letter strings. These include *grammatical tense* to distinguish between the two pronunciations of "read" for instance, and the *part of speech* to select among the different pronunciations of "perfect" in "The weather is perfect today" and "to perfect something".

Long-distance interactions pose a difficult problem for computationally expensive learning algorithms such as Backpropagation, since capturing them presumably requires a very wide window. This in turn requires a very large network with many weights, and these will be much more difficult and time-consuming to train. ID3, on the other hand, scales very well as the number of irrelevant features grows, so we have been able to apply it to much wider windows without problems. General solutions to the other two problems mentioned by Dennis Klatt appear to be quite challenging. Nevertheless, we will attempt to sketch below some *possible* means of getting around these problems.

One technique to be pursued is to refine the block decoding method discussed in [Hild90, Dietterich90a, Dietterich90b]. As it is, the block decoding technique is not compatible with extended context, since the latter requires that the pronunciations for the previous letters be known before an attempt is made to pronounce the current letter. Block decoding on the other hand delays the decoding phase, and attempts to decode a group of letters together as a unit or a "block". To bring these two conflicting methods together, we propose that blocks should be selected carefully, perhaps around a vowel or vowel group (V), and only if treating the vowel as part of the larger group would further *constrain* the mapping of that vowel or vowel group. Once a method is devised that can uniquely and unambiguously breaks the words into their constituent blocks, these blocks can be then each be treated as a unit which should alleviate the misalignment problem alluded to by Klatt.

Another technique to be pursued is to learn first how to identify syllable boundaries, then employ these (along with the normal features) to learn the pronunciations of the letters of the words. We started to work on this path, but found that the syllable boundary information encoded in the NETtalk dictionary as part of the stresses was not always correct. As a result, we were only able to learn the syllable boundaries with an accuracy of 92.6% when trained on 19,002 words (Syllable boundaries for 75.1% of the words in the 1000-word test set were guessed completely right). We concluded that these accuracy figures were too low and would degrade the performance of the next stage if it was based on these learned syllable boundaries. We also decided that the encoding of syllable information in the NETtalk dictionary needed to be double checked before a more serious attempt at this technique is warranted.

A third direction—that we also pursued briefly—is to employ the knowledge of the domain to define more complex features and present these as additional inputs to the learning systems. Some of these additional attributes that we have had a *modest* success with are:

- One bit each to encode whether the word has a suffix which is normally auto-stressed, pre-stressed-1, or pre-stressed-2. See [Halle71] for details.
- One bit to indicate if the center letter is a consonant between two vowels.
- One bit to indicate if the word final is a doubled consonant. For this purpose, "CK", "TCH", and "DGE" are considered doubled consonants.
- One bit each to indicate if the center letter is the first (second) letter of a

doubled consonant.

- Five bits to encode in unary the number of vowel groups in the word (up to five allowed for).
- Five bits to encode in unary the number of consonant groups in the word (up to five allowed for).
- Five bits to encode in unary the value of n, where n is such that the center letter is the n^{th} vowel group from word-final.

We did not investigate the separate effects of each of these features, nor did we employ them together with the additional features that we introduced in ExtContext(7). However, these features are easy to compute, and they should be included along with possibly others derived from the knowledge of the domain if they are found to boost the performance of the learning methods.

A fourth direction that we have not yet pursued is to implement one of the published methods for obtaining class probability estimates from decision trees. [Buntine90], for example, presents an algorithm that provides fairly accurate probability estimates at the leaves of a decision tree, rather than the simple binary outputs that we employed. This could eliminate the need for "observed" decoding.

7.3.3 Other Domains

The impressive performance of the machine learning systems that we developed for the task of English text-to-speech conversion should encourage researchers to consider applying similar machine learning techniques to the task of discovering pronunciation rules in other languages. It should also prompt them to consider these techniques for tackling similar or related tasks such as the pronunciations of proper names.

References

[Abramson63] Abramson, N. (1963). Information theory and coding. McGraw-Hill Book Company, Inc.

[Almuallim90] Almuallim, H. & Dietterich, T. G. (1990). Bounds on Optimal Coverage Learning Algorithms. Workshop on Computational Learning Theory and Natural Learning Systems, Princeton, NJ, Sep. 5, 1990.

[Bakiri91a] Bakiri, G., & Dietterich, T. G. (1991a). Applying machine learning techniques to construct high-performance letter-to-sound rules for English. Forthcoming.

[Bakiri91b] Bakiri, G., & Dietterich, T. G. (1990b). Boosting the performance of inductive learning programs via error-correcting output codes. Forthcoming.

[Bentley75] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18, 9 (September 1975).

[Blum87a] Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1987). Learnability and the Vapnik-Chervonenkis Dimension. Technical Report UCSC-CRL-87-20, Department of Computer and Information Sciences, University of California, Santa Cruz, Nov. 1987. Also in *Journal of ACM,* 36(4):929-965, 1989.

[Blum87b] Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1987). Occam's Razor. *Information Processing Letters,* 24:377-380,1987.

[Bose60] Bose, R. C., & Ray-Chaudhuri, D. K. (1960). On a class of error-correcting binary group codes. *Inf. Control*, 3, pp. 68-79, March 1960.

[Breiman84] Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and Regression Trees.* Monterey, CA: Wadsworth and Brooks.

[Buntine89] Buntine, W. L. (1989). Learning classification rules using Bayes. In *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 94–98). Ithaca, NY: Morgan Kaufmann.

[Buntine90] Buntine, W. (1990). A theory of learning classification rules. Doctoral dissertation. University of Technology, School of Computing Science, Sydney.

[Carterette74] Carterette, E. C., & Jones, M. G. (1974). *Informal speech.* (Los Angeles: University of California Press).

[Church85] Church, K. W. (1985). Stress assignment in letter-to-sound rules for speech synthesis. *Proc. 23rd Meeting Assoc. Comp. Ling.*, 246-253.

[Cole90] Cole, R. Muthusamy, Y. and Fanty, M. (1990). The ISOLET spoken letter database. Technical Report No. CSE 90-004. Beaverton, OR: Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering.

[Dietterich89a] Dietterich, T. G. (1989a). Limitations on Inductive Learning. *Proceedings of the Machine Learning Workshop*, July 1989.

[Dietterich89b] Dietterich, T. G. (1989b). Personal communication.

[Dietterich90a] Dietterich, T. G., Hild, H., & Bakiri, G. (1990). A comparative study of ID3 and Backpropagation for English Text-to-Speech Mapping. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 24–31). Austin, TX: Morgan Kaufmann.

[Dietterich90b] Dietterich, T. G., Hild, H., Bakiri, G. (1990b) A comparison of ID3 and backpropagation for English text-to-speech mapping. Technical report 90-30-4. Corvallis, OR: Oregon State University, Department of Computer Science.

[Dietterich90c] Dietterich, T. G. (1990c). Personal communication between Thomas Dietterich and David Wolpert.

[DUDA79] Duda, R., Gasching, J., & Hart, P. (1979). Model design in the Prospector consultant system for mineral exploration. In D. Michie (Ed.), *Expert systems in the micro electronic age*, Edinburgh: Edinburgh University Press.

[Fisher89] Fisher, D. H. & McKusik, K. B. (1989). An Empirical Comparison of ID3 and Backpropagation. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI., August 1989.

[Halle71] Halle, M. & Keyser S. J. (1971). English stress: its form, its growth, and its role in verse. New York: Harper and Row.

[Hamming86] Hamming, R. W. (1986). *Coding and Information Theory.* Second Edition, Prentice-Hall, New Jersey.

[Haussler88] Haussler, D. (1988). Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework. *Artificial Intelligence*, 36:177-221.

[Hild89] Hild, H. (1989). Variations on ID3 for Text-to-Speech Conversion. Master Thesis, CS Dept., Oregon State University, June 1989.

[Hinton87] Hinton G. (1987). Connectionist Learning Procedures. Rep. No. CMU-CS-87-115 (version 2), (1987). Department of Computer Science, Carnegie-Mellon University.

[Hocquenghem59] Hocquenghem, A. (1959). Codes correcteurs d'erreurs. *Chiffres*, 2, pp. 147-156.

[Klatt79] Klatt, D. H. (1979). Synthesis by Rule of Consonant-Vowel Syllables. In Speech Communications Group Working Papers, Cambridge, Massachusetts: Massachusetts Institute of Technology.

[Klatt82] Klatt, D. H., & Shipman, D. W. (1982). Letter-to-phoneme rules: A semi-automatic discovery procedure. *J. Acoust. Soc. Am. Suppl. 1* 72, S48.

[Klatt87] Klatt, D. H. (1987). Review of text-to-speech conversion for English. *J. Acoust. Soc. Am., 82*, (3), 737–793.

[Kuchera67] Kuchera, H., & Francis, W. N. (1967). *Computational Analysis of Modern-Day American English*, Brown University Press, Providence, RI, 1967.

[Lang90] Lang, K. J, Waibel, A. H, & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks, 3*, 33-43.

[Lin83] Lin, S., & Costello, D. J. Jr. (1983). *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, Inc., Englewood Cliffs, N.J.

[Lucassen83] Lucassen, J. M. (1983). Discovering phonemic base forms automatically: An information theoretic approach. Research Report RC 9833. Yorktown Heights, NY: IBM T. J. Watson Research Center.

[Lucassen84] Lucassen, J. M., & Mercer, R. L. (1984). An information theoretic approach to the automatic determination of phonemic base forms. *Proc. Int. Conf. Acoust. Speech Signal Process.* ICASSP-84, 42.5.1–42.5.4.

[Minsky88] Minsky, M.L., & Papert, S. (1988). *Perceptrons: Expanded Edition*, MIT Press, Cambridge, MA. (Original edition published in 1969.)

[Mooney89] Mooney, R. J., Shavlik, J. W., Towell, G. G., & Gove, A. (1989). An Experimental Comparison of Symbolic and Connectionist Learning Algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI., August 1989.

[McClelland88] McClelland, J. L., & Rumelhart, D. E. (1988). *Explorations in Parallel Distributed Processing,* Cambridge, MA: MIT Press.

[Mingers89] Mingers, J. (1989). An empirical comparison of pruning methods for decision tree induction. *Machine Learning, 4* (2), 227–243.

[Mooney89] Mooney, R., Shavlik, J., Towell, G., and Gove, A. (1989). An experimental comparison of symbolic and connectionist learning algorithms. *IJCAI-89: Eleventh International Joint Conference on Artificial Intelligence.* 775–80.

[Pagallo88] Pagallo, G., & Haussler, D. (1988). Feature Discovery in Empirical Learning. Technical Report, Department of Computer and Information Science, University of California, Santa Cruz, August 1988.

[Quinlan83] Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess endgames, in Michalski, R. S., Carbonell, J., and Mitchell, T. M., (eds.), *Machine learning: An artificial intelligence approach, Vol. I,* Palo Alto: Tioga Press. 463–482.

[Quinlan86a] Quinlan, J. R. (1986a). The effect of noise on concept learning. In Michalski, R. S., Carbonell, J., and Mitchell, T. M., (eds.), *Machine learning, Vol. II,* Palo Alto: Tioga Press. 149–166.

[Quinlan86b] Quinlan, J. R. (1986b). Induction of Decision Trees, *Machine Learning,1* (1), 81–106.

[Quinlan87] Quinlan, J. R. (1987). Simplifying Decision Trees. *International Journal of Man-Machine Studies, 27,* 221–234.

[Rivest87] Rivest, R. L. (1987). Learning Decision Lists. *Machine Learning,* 2 (3), 229-246.

[Rosenblatt58] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review, 65* (6), 386–408.

[Rumelhart86] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., and McClelland, J. L., (eds.) *Parallel Distributed Processing*, Vol 1. 318–362.

[Schlimmer86] Schlimmer, J. C., & Granger, R. H., Jr. (1986). Incremental learning from noisy data. *Machine Learning*, 1, (1986), 317-354.

[Schlimmer87] Schlimmer, J. C. (1987). Learning and representation change. In *Proceedings of AAAI-87*, Seattle, WA. Los Altos, CA: Morgan-Kaufmann, 511-515.

[Sejnowski87] Sejnowski, T. J., & Rosenberg, C. R. (1987). Parallel networks that learn to pronouce English text. *Complex Systems, 1*, 145–168.

[Shannon48] Shannon, C. E. (1948). A mathematical theory of communication. *Bell Syst. J.*, 27, pp. 379-423 (part I), 623-656 (Part II), July 1948.

[Shavlik89] Shavlik, J. W., Mooney, R. J., & Towell, G. G. (1989). Symbolic and Neural Learning Algorithms: An Empirical Comparison. *proceedings of the Machine Learning Workshop*, July 1989.

[Shavlik90] Shavlik, J. W., Mooney, R. J., & Towell, G. G. (1990). Symbolic and neural learning algorithms: An experimental comparison (revised). Technical Report 955. Madison, WI: University of Wisconsin-Madison, Computer Sciences Department.

[Utgoff88] Utgoff, P. E. (1988). Perceptron Trees: A Case Study in Hybrid Concept Representations. *Proceedings of the National Conference on Artificial Intelligence*, St. Paul, MN., August 1988, pp. 601-606.

[Tesauro89] Tesauro, G., & Sejnowski, T. J. (1989). A Parallel System that Learns to Play Backgammon. *Artificial Intelligence*, 39,3.

[Valiant84] Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM, 27,* 1134–1142.

[Valiant85] Valiant, L. G. (1985). Learning disjunctions of conjunctions. In *Proceedings of the 9th IJCAI*, Pages 560-566, Morgan Kaufmann, 1985.

[Vitali90] Tony Vitali (1990). Personal communication (electronic mail) between Tony Vitali of Digital Equipment Corporation (DEC) and Tom Dietterich.

[Wettschereck90] Wettschereck, D. (1990). An implementation and initial test of generalized radial basis functions. Master Thesis, CS Dept., Oregon State University, June 1990.

[Wolpert89] David H. Wolpert (1989). A benchmark for how well neural nets generalize. *Biological Cybernetics*, 61, 303-313.

[Wolpert90a] David H. Wolpert (1990a). A mathematical theory of generalization: Part I. *Complex Systems.*

[Wolpert90b] David H. Wolpert (1990b). A mathematical theory of generalization: Part II. *Complex Systems.*

[Wolpert90c] David H. Wolpert (1990c). Constructing a generalizer superior to NETtalk via a mathematical theory of generalization. *Neural Networks*, vol. 3, pp. 445-452.

APPENDICES

Appendix A

A Brief Introduction to some Machine Learning Algorithms

This appendix will give the reader a brief introduction to the ID3, FRINGE, Perceptron and the Backpropagation algorithms discussed primarily in Chapter 2.

A.1 The ID3 Algorithm

ID3 is a learning algorithm of the TDIDT (Top-Down Induction of Decision Trees) family [Quinlan86]. Given a subset of the learning examples (called the training set), the algorithm constructs a decision tree that can then be employed to classify all the examples of a particular concept. A learning example is a pair: $(\vec{x}, v)$ where $\vec{x}$ is a vector of attributes: $\langle x_1, x_2, \ldots, x_n \rangle$[72] and v is the class associated with $\vec{x}$. In the general case, the features x_i and the outcome v can be multivalued. For simplicity, we will talk about a particular version of the ID3-algorithm that applies only to binary feature vectors and binary classes.

[72]We will use $a_1, a_2, \ldots, a_n$ to refer to the names of the attributes, and $\langle x_1, x_2, \ldots, x_n \rangle$ to refer to the vector of values for these attributes.

Sketch of the ID3 algorithm:

Function build-tree (*training-set*)

INPUT: A *training-set* of m training examples. Each example is a pair: $(\vec{x}_i, v_i)$ where $\vec{x}_i$ is an n dimensional binary attribute (feature) vector: $\langle x_1, x_2, \ldots, x_n \rangle$, $x_j \in \{0,1\}$ and $v_i \in \{+,-\}$ giving the binary class associated with $\vec{x}_i$. An example is called positive if v_i is $+$, and negative otherwise.

OUTPUT: A (binary) decision tree.

The decision tree is formed recursively as follows:

```
begin {build-tree}
  If the training-set consists only of positive examples then
    output a leaf node marked +.
  else if the training-set consists only of negative examples then
    output a leaf node marked −.
  else
    • Select one of the attributes $a_1, a_2, \ldots, a_n$ to be at the root of the tree.
      (The criterion for the selection will be detailed later.)
      Call that attribute best-a.
    • Divide the training-set to two sets:
        The zero-set containing all examples that have a value of zero for attribute best-a.
        The one-set containing all examples that have a value of one for attribute best-a.
    • Mark attribute best-a as already used.
    • Let:
        zero-set-subtree := build-tree(zero-set)
        one-set-subtree := build-tree(one-set)
      (Note that above are two recursive calls to build-tree)
    • Output a binary tree with best-a as the root,
        the zero-set-subtree as the left subtree, and
        the one-set-subtree as the right subtree.
end. {build-tree}
```

Figure 8 shows how a decision tree is built from a simple training set consisting of 6 examples and 4 attributes: $a_1, \ldots, a_4$.

We will now turn our attention to the criterion for determining which attribute should be tested at the root of a (sub)tree and hence serve as the basis for further splitting the examples reaching that node. This criterion in ID3 is biased to select attributes that will lead to a smaller decision tree. It is a heuristic, so ID3 is not *guaranteed* to come up with the smallest possible decision tree for a given set of training examples.

To detail the attribute selection criterion, we will consider some node in the tree with a set of p positive and n negative training examples reaching that node. The uncertainty in the class value to be assigned for that node is measured by the entropy function:

$$entr(n,p) = -\tfrac{n}{n+p}\log_2 \tfrac{n}{n+p} - \tfrac{p}{n+p}\log_2 \tfrac{p}{n+p}$$

The above formula is intuitively appealing, since it assigns a maximum value (1) for class uncertainty when the sample is split evenly between negative and positive examples ($n = p$), and a minimum value (0) for the uncertainty when the sample consists of only one type of examples: either positive ($n = 0$) or negative ($p = 0$).

ID3 selects the feature that provides the most information about the class value, i.e. the one that *minimizes* the uncertainty in the class after the split—calculated as the weighted average of the entropies of the *zero-set* and the *one-set*:

$$unc(a_i) = \tfrac{n_0+p_0}{n+p}entr(n_0,p_0) + \tfrac{n_1+p_1}{n+p}entr(n_1,p_1)$$

where

n, p	= number of negative, positive examples in the training set reaching the node
a_i	= attribute being considered as a basis for the split
n_0, p_0	= number of negative, positive examples in the zero-set, and
n_1, p_1	= number of negative, positive examples in the one-set.

The uncertainty is calculated for all the attributes (not yet tested on the path from the root to the current node) and the one that minimizes the uncertainty is selected as *best-a*.

Generalizations of the above attribute selection criterion to handle non-boolean classes and/or non-boolean values for the attributes are found in [Breiman84, Pagallo88].

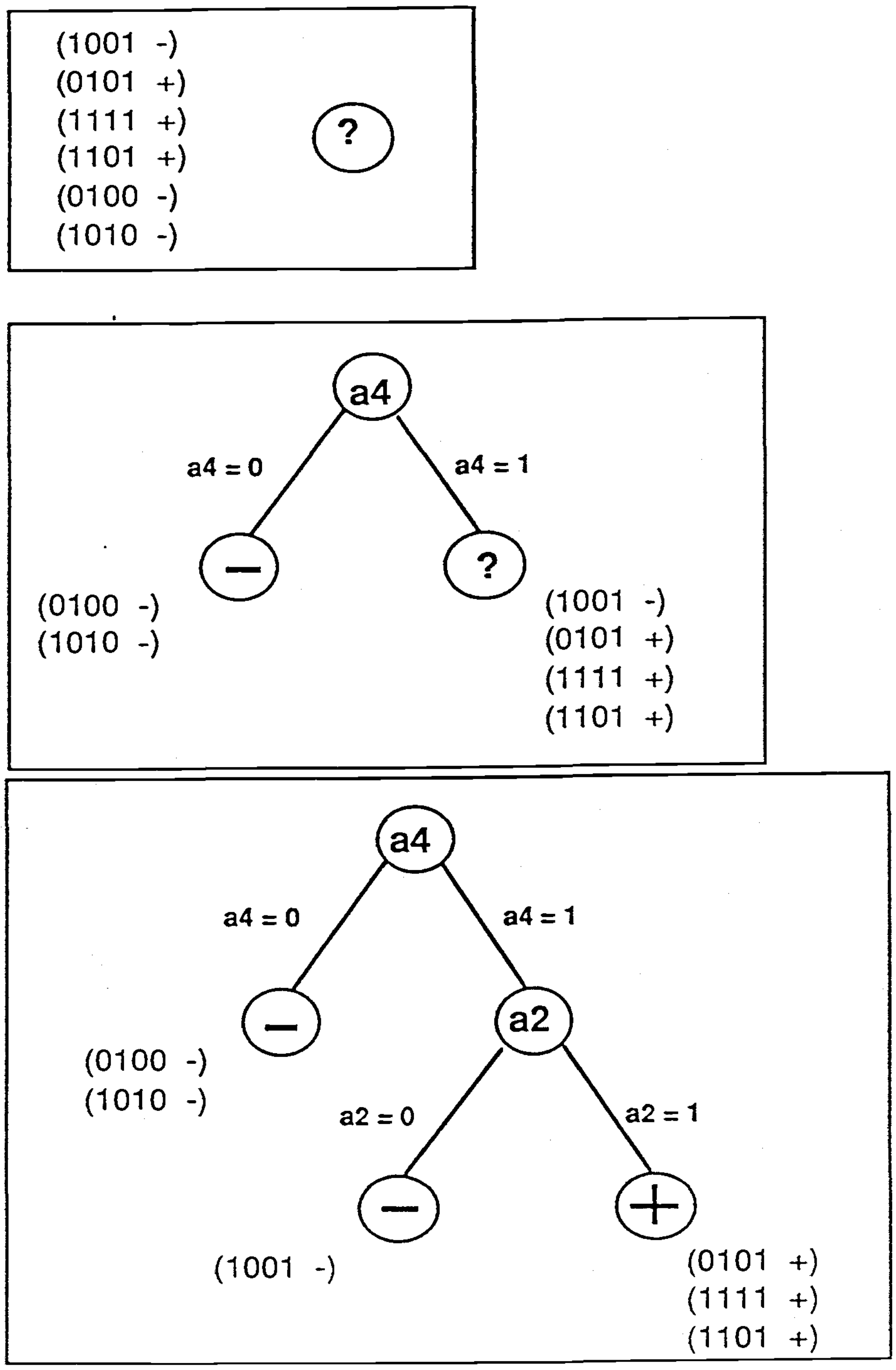

Figure 8. An ID3 example. Building a decision tree from 6 examples and 4 features: $a_1, \ldots, a_4$.

A.2 The FRINGE Algorithm

FRINGE is an algorithm that dynamically creates and uses *new* features defined as boolean combinations of the original (primitive) features [Pagallo88]. The algorithm begins with a set V of primitive attributes and employs ID3 to build a decision tree for a set of training examples expressed only in terms of the variables in V. Then, an *extract-feature* heuristic generates new features as Boolean combinations of the variables that are tested in the nodes near the *fringe* of the tree. This heuristic will be described in more detail below. The set of new features is added to the variable set, V, and the training examples are augmented to include the newly defined features. ID3 is again called upon to build a decision tree using the expanded feature set, V. The process is repeated until one of the following occurs:

- No new features are defined by the extract-feature procedure, or
- The decision trees output by ID3 are identical for two successive iterations, or,
- A predefined maximum number of iterations is reached, or,
- A predefined maximum number of newly defined features is reached.

The *extract-feature* procedure takes as input a binary decision tree and outputs the set of *newly-defined features.* It scans the tree, and, for every positive leaf l (at depth > 1 from the root), it defines a new feature as a *conjunction* involving the variables tested at the parent node, p, and the grand-parent node, g, of the leaf l as follows:

```
let $v_p$ and $v_g$ be the test variables at nodes $p$ and $g$ respectively.
if $l$ is on the right subtree of $p$ and $p$ is on the right subtree of $g$ then
    define $v_p v_g$ as a new feature
else if $l$ is on the right subtree of $p$ and $p$ is on the left subtree of $g$ then
    define $v_p \bar{v}_g$ as a new feature
else if $l$ is on the left subtree of $p$ and $p$ is on the right subtree of $g$ then
    define $\bar{v}_p v_g$ as a new feature
else $l$ must be on the left subtree of $p$ and $p$ on the right subtree of $g$
    define $\bar{v}_p \bar{v}_g$ as a new feature.
```

(The above assumes that the decision tree adopts the convention that the left edge represents the negative or 0 outcome of the variable tested at the node, and the right edge represents the positive or 1 outcome.)

The procedure generates simple features at each step, but it adaptively assembles more complex combinations of the attributes through the iterative process. So, the k^{th} iteration may produce variables of size up to 2^k. The algorithm was conceived to solve the replication problem in decision trees (The replication problem means that the same sequence of tests leading to a positive leaf is replicated in the tree, making it unnecessarily large. See [Pagallo88] for a more detailed description of this problem).

[Pagallo88] tested the FRINGE algorithm on various random boolean concepts in disjunctive normal form (DNF), and found that it outperforms the standard decision tree learning algorithm ID3 on those concepts. The algorithm is yet to be tested on real life domains to verify the generality of these results. It should also be noted that FRINGE requires considerably more computer resources (both space and time) than ID3. This may render the algorithm impractical for large tasks (e.g. the NETtalk domain).

A.3 The Perceptron Algorithm

The perceptron learning algorithm is a method for adjusting the weights of a single linear threshold unit (Figure 9), so that it converges on the correct outputs for a set of training examples. The output of each linear threshold unit (perceptron) is related to its input by the relation:

$$O_p = \sum_{i=1}^{n} w_i x_{pi} - \phi$$

where the the subscript p refers to a particular training example: $(\vec{x}_p, t_p)$,

$\vec{x}_p = \langle x_1, x_2, \ldots, x_n \rangle, x_j \in \{0,1\}$ being the feature vector part of the training example, and $t_p \in \{0,1\}$ the desired output.

To handle the threshold, ϕ, cleanly, a new vector $\vec{y}_p = \{1, x_1, x_2, \ldots, x_n\}$ is normally defined. The weight vector $\vec{w}$ will be a vector of real-valued weights of the same length as $\vec{y}_p$. The first element in $\vec{w}$ will then play the role of the threshold. A given $\vec{w}$ classify an example as positive (O_p is considered 1) if $\vec{w} \cdot \vec{y}_p > 0$, and negative ($O_p$ is considered 0) otherwise.

The learning algorithm begins with all elements of $\vec{w}$ equal to zero. $\vec{w}$ is then trained on all the examples in the training set as follows: For each example, p, the vector $\vec{y}_p$ is constructed and the output, O_p, calculated as described above. If O_p turned out the same as t_p, the target or desired output, then no change to the weights are made. Otherwise, each component of the weight vector, w_i, is changed by the amount

$$\Delta_p w_i = y_i(t_p - O_p)$$

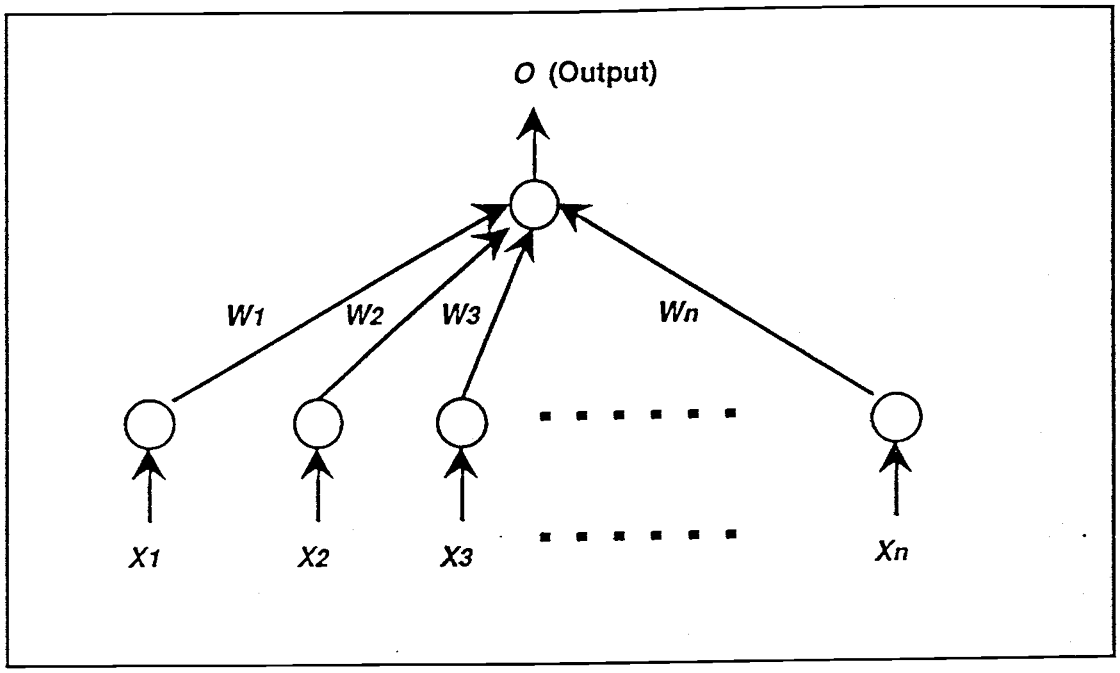

Figure 9. A linear threshold unit (perceptron).

where y_i is the corresponding element of $\vec{w}, i = 1, \ldots, (1+n)$. The process is iterated until $\vec{w}$ classifies all the training examples correctly, i.e. until convergence, or until a maximum number of iterations is reached.

The above procedure performs gradient descent (i.e. hill climbing) in weight space in an attempt to minimize the sum of the squares of the errors across all the training examples. It is guaranteed to converge only if the data are linearly separable [Minsky88].

A.4 The Backpropagation Algorithm

Many of the limitations of perceptrons can be overcome by multi-layer networks: Networks that have one or more layer(s) of hidden units (in addition to the input and output layers). A hidden unit is one that neither receives external input nor produces external output. These multi-layer networks can learn much more complicated functions than networks that lack hidden units, but the learning is generally much slower, because it must explore the space of possible ways of using the hidden units [Hinton87].

Backpropagation (also called the generalized delta rule) [Rumelhart86] is the most popular algorithm employed to train multi-layer Neural Networks, and it has been tested on several large-scale problems [Sejnowski87, Tesauro89]. The algorithm uses gradient descent in an attempt to minimize the sum of the squares of the errors across all of the training inputs. For the method to work, the thresholding function of each unit must be smoothed so that it is everywhere differentiable. The most common output function used is the sigmoid function presented below. For each unit j, its output O_{pj} for an input/output pair p is:

$$O_{pj} = \frac{1}{1 + e^{-(\sum_i w_{ji} O_{pi} + \phi_j)}}$$

where w_{ji} is the weight from unit i to unit j and ϕ_j is the threshold for unit j . The weights for each unit are changed after the presentation of each example (input/ output pattern) p by back-propagating error measures layer by layer from the output back to the input according to the following equations:

$$\Delta_p w_{ji} = \eta \delta_{pj} O_{pi} + \alpha \Delta_{p-1} w_{ji}$$

where

$$\delta_{pj} = O_{pj}(1 - O_{pj})(t_{pj} - O_{pj}) \qquad \text{if } j \text{ is an output unit, and}$$

$$\delta_{pj} = O_{pj}(1 - O_{pj}) \textstyle\sum_k delta_{pk} w_{kj} \qquad \text{if } j \text{ is a hidden unit.}$$

Note that the sum over k means the sum over all the units that unit j feeds into. η is a parameter called the learning rate[73]; α is a parameter called the momentum term constant[74] which reduces fluctuations during hill climbing; t_{pj} is the target output for output unit j for pattern p; and δ_{pj} measures the error of the output of unit j for pattern p.

The training process is extremely expensive, since it requires many passes[75] through the training set, and all the weights in the network have to be updated for every single example in every pass. The process is not guaranteed to converge, since the algorithm may get stuck at local minima instead of the global minimum error state desired. However, work done on the NETtalk domain shows that this does not seem to present serious problems in practice.

[73]A value of 0.25 is typically used for the learning rate in practice.

[74]A value of 0.9 is typically used for the momentum term constant in practice.

[75]Each pass is called an *epoch*. Up to 100 epochs have been used on the NETtalk task.

Appendix B

The 26-bit Code for Phoneme/Stress Pairs

Sejnowski and Rosenberg developed the following distributed code for representing the phonemes and stresses. The examples were supplied with their database.

Phoneme Code		
Phoneme	Codeword	Examples
/a/	000010000000100100000	wAd, dOt, Odd
/b/	000100000001010000000	Bad
/c/	000001000000000010000	Or, cAUght
/d/	100000000001010000000	aDd
/e/	010000000000100010000	Angel, blAde, wAy
/f/	000100010000000000000	Farm
/g/	000001000001010000000	Gap
/h/	001000000100000000000	Hot, WHo
/i/	000100000000101000000	Eve, bEe
/k/	000001000001000000000	Cab, Keep
/l/	010000000100010000000	Lad
/m/	000100000010010000000	Man, iMp
/n/	100000000010010000000	GNat, aNd
/o/	001000000000100010000	Only, Own
/p/	000100000001000000000	Pad, aPt
/r/	000010000100010000000	Rap
/s/	100000010000000000000	Cent, aSk
/t/	100000000001000000000	Tab
/u/	001000000000101000000	bOOt, OOze, yOU

Phoneme Code

Phoneme	Codeword	Examples
/v/	000100010000010000000	Vat
/w/	000100001000010000000	We, liqUid
/x/	000010000000000010000	pirAte, welcOme
/y/	000010001000010000000	Yes, senIor
/z/	100000010000010000000	Zoo, goeS
/A/	110000000000100010000	Ice, hEIght, EYe
/C/	000010100000000000000	CHart, Cello
/D/	010000010000010000000	THe, moTHer
/E/	010100000000000010000	mAny, End, hEAd
/G/	000001000010010000000	leNGth, loNG, baNk
/I/	000100000000001000000	gIve, bUsy, captAIn
/J/	000010100000010000000	Jam, Gem
/K/	000011110000000000000	aNXious, seXual
/L/	100000000100010000000	eviL, abLe
/M/	010000000010010000000	chasM
/N/	000010000010010000000	shorteN, basiN
/O/	100010000000100010000	OIl, bOY
/Q/	000101100001010000000	Quilt
/R/	000001000100010000000	honeR, afteR, satyR
/S/	000010010000000000000	oCean, wiSH
/T/	010000010000000000000	THaw, baTH
/U/	000001000000001000000	wOOd, cOUld, pUt
/W/	000011000000101010000	oUT, toWel, hoUse
/X/	110000100000000000000	miXture, anneX
/Y/	110100000000101000000	Use, fEUd, nEw
/Z/	000010010000010000000	uSual, viSion
/@/	010000000000000100000	cAb, plAId
/!/	010100100000000000000	naZi, piZZa
/#/	000011100000010000000	auXiliary, eXist
/*/	100100001000010100000	WHat
/^/	100000000000000100000	Up, sOn, blOOd
/+/	000000000000000000000	abattOIr, mademOIselle
/-/	000000000000000001001	silence
/_/	000000000000000001010	word-boundary
/./	000000000000000000110	period

Here are the meanings of the individual bit positions:

Bit Position	Meaning
1	Alveolar = Central1
2	Dental = Front2
3	Glottal = Back2
4	Labial = Front1
5	Palatal = Central2
6	Velar = Back1
7	Affricative
8	Fricative
9	Glide
10	Liquid
11	Nasal
12	Stop
13	Tensed
14	Voiced
15	High
16	Low
17	Medium
18	Elide
19	FullStop
20	Pause
21	Silent

The stress code actually encodes syllable boundary information as well as stresses.

Stress Code		
Stress	Codeword	Meaning
<	10000	a consonant or vowel following the first vowel of the syllable nucleus.
>	01000	a consonant prior to a syllable nucleus.
0	00010	the first vowel in the nucleus of an unstressed syllable.
2	00100	the first vowel in the nucleus of a syllable receiving secondary stress.
1	00110	the first vowel in the nucleus of a syllable receiving primary stress.
-	11001	silence

Appendix C

Generating BCH Codewords

Below is the C program that we employed[76] to generate the full set of 2^k BCH codewords, where k is the number of information bits. This routine takes as input the values of n, k and t and the *binary* representation of the generator polynomial and prints out the full set of codewords (See Appendix C in [Lin83] for details).

As an example, the input to the program for the case

n	k	t	Generator Polynomial (Octal)
31	6	7	313365047

will be as follows:

```
31 6  7
1 1  0 0 1  0 1 1  0 1 1  1 1 0  1 0 1  0 0 0  1 0 0  1 1 1
```

Note that the left most zero introduced when 313365047 was converted from octal to binary was omitted from the input to the program.

[76]We thank Dr. Sulaiman Al-Bassam for providing us with this C routine.

```
int g[20][1025], cword[1025];
int n,k,d,r,i,j,twok,x,y,t,count;

main()
{
   printf("Enter  n k and t --> ");
   scanf("%d %d %d",&n,&k,&t);
   r=n-k; d=2*t+1;
   printf("Enter the generator polynomial of degree %d --> ",r);
   for (i=0; i<=r; i++) scanf("%d",&g[0][i]);
   for (i=1; i<k; i++) for (j=i; j<k+r; j++) g[i][j] = g[i-1][j-1];
    printf(";;k = %d  r = %d  and  d = %d \n",k,r,d);
   printf("(setq *distant-codes-k=%d-d=%d-r=%d* '(\n",k,d,r);
   for (i=0, twok=1; i<k; i++) twok = 2*twok;
   for (x=0; x<twok; x++) { /* generate all 2**k words */
      printf("(#%d*",k+r);
      for (j=0; j<k+r; j++) {
          y=x;
          for (cword[j]=0, i=0; i<k; i++, y = y>>1) {
               cword[j] = cword[j] ^ ((y&1) && g[i][j]);
               }
          printf("%d",cword[j]);
          }
      printf(")\n");
      }
   printf("))\n");
}
```

Appendix D

Feature Sets Defined by the Method of Mercer & Lucassen

This appendix presents the feature sets defined by the information theoretic approach of Mercer & Lucassen (discussed in Section 4.2.1) for the current letter, the letter to the left, the letter to the right, the phonemes corresponding to letters to the *left* of the current letter, and, the stresses corresponding to letters to the *left* of the current letter.

Table 83. Values for binary features (questions) about the current letter, the letter to the left and the letter to the right, defined by the information theoretic approach adopted from [Lucassen83]. Values shown for the last 2 rows (the last three rows for the current letter) were assigned *arbitrarily* since these symbols did not appear in our data set.

Letter	Letter to the left						Current Letter						Letter to the right							
or input	QLL1 through QLL6						QL1 through QL6						QLR1 through QLR8							
symbol	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	7	8
A	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
B	0	0	0	1	0	1	1	0	0	0	1	0	1	1	1	1	0	1	1	1
C	1	1	1	0	1	1	1	0	1	1	0	0	0	1	1	1	1	0	1	1
D	1	0	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	0	1	0
E	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
F	0	1	1	1	0	1	1	1	1	0	0	1	1	0	1	0	0	1	1	1
G	1	1	1	1	1	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1
H	1	0	0	1	0	1	0	1	0	0	0	0	0	1	1	0	1	1	0	0
I	1	1	1	0	1	0	0	0	0	1	1	0	0	1	0	0	0	1	1	1
J	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	1	1	1	1	1
K	1	1	0	1	0	1	1	0	1	1	1	1	0	1	1	0	0	1	0	1
L	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	0	1	0	0	0
M	1	0	1	0	1	0	1	1	0	1	1	1	1	0	0	1	0	1	1	1
N	1	0	1	1	0	1	1	1	0	0	1	0	0	1	1	1	1	1	0	1
O	1	0	1	1	1	1	0	0	1	1	1	0	0	1	0	1	0	1	1	0
P	0	0	1	1	0	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1
Q	1	1	1	1	1	0	1	0	1	1	0	1	0	1	1	0	1	0	1	0
R	0	1	1	0	0	1	1	1	1	1	0	1	0	1	1	0	0	1	0	0
S	1	0	1	0	0	1	1	0	1	0	1	0	1	0	0	0	0	1	1	1
T	1	0	0	0	0	0	1	0	0	1	1	0	0	1	1	1	0	0	0	0
U	1	0	0	1	1	1	0	1	0	1	1	0	0	0	1	1	0	1	1	0
V	0	0	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	1	1	1
W	1	0	1	0	1	1	0	1	0	0	0	1	0	1	1	0	0	1	1	1
X	1	1	1	1	0	1	1	1	1	0	0	0	1	1	0	1	1	1	1	1
Y	1	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1	1	1	1
Z	0	1	0	0	1	1	1	0	1	0	0	0	1	0	1	1	0	1	1	1
-	0	1	0	0	0	1	1	1	1	1	1	1	0	0	0	1	0	1	1	1
-	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
.	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 84. Values for binary features (questions) about the phonemes corresponding to letters to the *left* of the current letter, defined by the information theoretic approach adopted from [Lucassen83]. Values shown for ".", "Q" and "M" were assigned *arbitrarily* since these symbols did not appear in our data set.

	QP1 through QP10									
Phon.	1	2	3	4	5	6	7	8	9	10
!	1	0	1	1	1	0	0	1	0	1
#	1	1	0	0	1	1	0	1	0	1
*	1	1	1	0	0	0	1	1	0	1
+	1	1	1	0	0	0	0	0	0	1
-	0	0	0	0	1	1	0	1	0	1
@	0	1	0	0	1	0	1	0	0	0
A	0	1	1	0	0	1	0	1	1	0
C	1	1	0	0	0	1	1	1	0	1
D	1	1	1	0	0	0	1	0	1	1
E	0	1	0	0	1	1	0	0	0	0
G	0	1	0	1	1	0	0	0	0	1
I	0	1	0	0	0	0	0	0	0	0
J	1	1	0	1	0	1	0	1	1	1
K	1	1	0	0	0	0	0	1	0	1
L	1	1	0	0	0	0	1	0	0	1
N	1	1	1	1	1	0	0	1	1	1
O	1	1	1	0	0	0	0	0	0	0
R	0	1	1	0	0	1	0	0	1	0
S	1	1	0	0	0	1	0	1	0	1
T	1	1	1	0	0	0	1	1	1	1
U	0	1	0	1	0	0	0	0	0	0
W	1	1	1	0	0	0	0	0	1	1
X	1	0	1	0	1	0	0	1	1	1
Y	0	1	1	0	0	1	1	1	1	0
Z	1	1	0	0	0	0	0	0	0	1
^	0	1	0	0	1	0	0	0	0	0
_	1	0	0	1	1	0	0	1	1	0

	QP1 through QP10									
Phon.	1	2	3	4	5	6	7	8	9	10
a	0	1	0	0	1	1	1	0	0	0
b	1	1	0	0	1	0	1	1	0	1
c	0	1	0	1	0	1	0	0	0	0
d	1	1	0	0	1	0	0	0	0	1
e	0	1	1	0	1	0	0	1	1	0
f	1	1	0	0	1	0	1	0	0	1
g	1	1	0	0	1	1	1	1	1	1
h	1	1	0	1	0	0	0	1	1	1
i	1	1	1	1	0	1	0	1	1	0
k	1	1	0	0	1	1	0	0	1	1
l	1	1	0	1	1	1	0	1	1	1
m	1	1	0	1	0	1	1	1	1	1
n	1	1	1	1	1	1	0	1	1	1
o	0	1	1	0	0	0	0	0	0	0
p	1	1	0	0	1	0	1	1	1	1
r	1	1	0	1	1	0	0	1	1	1
s	1	1	1	0	1	0	0	1	1	1
t	1	1	0	0	1	0	0	0	1	1
u	1	1	1	0	0	0	0	1	1	1
v	1	1	0	0	1	0	0	1	1	1
w	1	1	0	1	0	0	1	1	1	1
x	0	1	1	1	1	0	0	0	0	0
y	0	0	1	0	1	0	0	0	1	1
z	1	1	1	0	0	1	0	0	0	1
.	0	0	0	0	0	0	0	0	0	0
Q	1	1	1	1	1	1	1	1	1	1
M	0	0	0	0	0	1	1	1	1	1

Table 85. Values for binary features (questions) about the stresses corresponding to letters to the *left* of the current letter, defined by the information theoretic approach adopted from [Lucassen83].

Stress symbol	QS_1	QS_2	QS_3	QS_4
-	1	0	0	0
0	0	0	0	0
1	0	0	1	1
2	0	0	1	0
<	0	1	1	1
>	1	1	0	0

Appendix E

Mutual Information Data

This appendix presents the values for the weights employed in the various configurations of the Wolpert method employed in Chapters 5 and 6. Table 86 shows the original set of weights that Wolpert arrived at through cross-validation techniques. Tables 87 through 92 show the mutual information weights when full stresses are employed (Chapter 5), while Tables 93 through 98 present the mutual information weights when simplified stresses are employed (Chapter 6).

Table 86. The "best" set of weights that Wolpert arrived at through cross-validation techniques.

Letter position i	Which side	Wolpert's Weights
-3	Letters	1.0
-2	to the	1.0
-1	Left.	2.0
0	Current Letter	5.0
+1	Letters	2.0
+2	to the	1.0
+3	Right.	1.0

Table 87. Mutual information data between the letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only.

Letter position i	Which side	Output Considered		
		Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.210421	0.120230	0.065382
-6	Letters	0.272027	0.146963	0.074936
-5	to	0.321749	0.172124	0.091272
-4	the	0.394367	0.209157	0.111120
-3	Left.	0.472386	0.258011	0.132691
-2		0.646019	0.361062	0.203981
-1		1.175573	0.799755	0.561864
0	Current Letter	3.387800	3.250498	0.788493
+1		1.352364	0.937440	0.506969
+2		0.787762	0.439971	0.300070
+3	Letters	0.536494	0.279319	0.186361
+4	to	0.402002	0.211752	0.105923
+5	the	0.316185	0.169647	0.072892
+6	right	0.256352	0.136780	0.055075
+7		0.205849	0.109149	0.046244

Table 88. Mutual information data between *phonemes* corresponding to letters *i* positions away from the current letter (in a 15-letter window) and the output class of the current letter. Three values are shown for each phoneme position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only.

Phoneme position *i*	Which side	Output Considered: Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.238805	0.142476	0.070655
-6	Phonemes	0.318173	0.182300	0.084393
-5	to	0.392106	0.221002	0.103524
-4	the	0.490989	0.275235	0.130538
-3	Left.	0.626356	0.351652	0.171562
-2		0.860354	0.510260	0.282594
-1		1.473031	1.017784	0.633474
0	Current Phoneme	4.744520	4.744520	1.021150
+1		1.537205	1.038414	0.638245
+2		0.923165	0.529978	0.346896
+3	Phonemes	0.658606	0.350298	0.218635
+4	to	0.484380	0.268148	0.121293
+5	the	0.385824	0.217712	0.088757
+6	right	0.308311	0.175743	0.066142
+7		0.229530	0.129057	0.054136

Table 89. Mutual information data between *stresses* corresponding to letters *i* positions away from the current letter (in a 15-letter window) and the output class of the current letter. Three values are shown for each stress position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only.

Stress position *i*	Which side	Output Considered: Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.140473	0.074596	0.058028
-6	Stresses	0.184422	0.091415	0.082377
-5	to	0.223616	0.111539	0.106448
-4	the	0.273866	0.138958	0.117482
-3	Left.	0.346596	0.180724	0.140518
-2		0.485113	0.252028	0.249313
-1		0.968124	0.589386	0.692403
0	Current Stress	2.074696	1.021150	2.074696
+1		1.046594	0.606069	0.693227
+2		0.569097	0.270679	0.312549
+3	Stresses	0.385126	0.180858	0.189079
+4	to	0.252142	0.122241	0.098607
+5	the	0.199157	0.095190	0.078622
+6	right	0.151356	0.072373	0.051641
+7		0.111284	0.053625	0.033957

Table 90. Mutual information data between the features L_{f1} and L_{f2} for letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only. See Section 5.3.5 for a definition of L_{f1} and L_{f2}.

Letter position i	Which side	Output Considered As: Phoneme Stress pairs L_{f1}	Phoneme Stress pairs L_{f2}	Phonemes Only L_{f1}	Phonemes Only L_{f2}	Stresses Only L_{f1}	Stresses Only L_{f2}
-7	Letters to the Left.	0.061863	0.033022	0.031072	0.017257	0.031461	0.014327
-6		0.077281	0.033928	0.035753	0.017211	0.037114	0.014980
-5		0.085919	0.040848	0.044834	0.019882	0.044247	0.017444
-4		0.087428	0.047375	0.040088	0.025401	0.038871	0.020913
-3		0.092152	0.049317	0.050462	0.024385	0.043900	0.010816
-2		0.136013	0.072339	0.057989	0.048807	0.070843	0.015248
-1		0.420121	0.289716	0.319242	0.183141	0.266210	0.220013
0	Current Letter	0.897993	0.631529	0.837232	0.573052	0.681713	0.329940
+1	Letters to the right	0.399681	0.254041	0.324966	0.181176	0.195822	0.144907
+2		0.197969	0.108871	0.084425	0.078051	0.111343	0.015793
+3		0.114572	0.071465	0.060366	0.024901	0.052148	0.034427
+4		0.078823	0.053513	0.038132	0.023958	0.026511	0.023982
+5		0.070426	0.042731	0.037442	0.020840	0.029431	0.011622
+6		0.054394	0.033882	0.031173	0.016252	0.017838	0.009252
+7		0.042467	0.032320	0.018417	0.018836	0.014560	0.010512

Table 91. Mutual information data between the feature P_{f1} for *phonemes* corresponding to letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only. See Section 5.3.5 for a definition of P_{f1}.

Phoneme position i	Which side	Output Considered: Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.011187	0.005593	0.003174
-6	Phonemes	0.014898	0.008119	0.004667
-5	to	0.017848	0.007130	0.006853
-4	the	0.024507	0.014363	0.008630
-3	Left.	0.031864	0.015795	0.005873
-2		0.052141	0.033388	0.018055
-1		0.078337	0.048753	0.033503
0	Current Phoneme	0.505651	0.505651	0.193877
+1		0.154284	0.089839	0.109193
+2		0.055195	0.031707	0.007973
+3	Phonemes	0.026438	0.012418	0.001859
+4	to	0.026561	0.016432	0.005193
+5	the	0.019775	0.009690	0.004418
+6	right	0.021173	0.008527	0.006003
+7		0.014605	0.006146	0.004171

Table 92. Mutual information data between the features S_{f1} and S_{f2} for *stresses* corresponding to letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only. See Section 5.3.5 for a definition of S_{f1} and S_{f2}.

Letter		Output Considered As					
position	Which	Phoneme Stress pairs		Phonemes Only		Stresses Only	
i	side	S_{f1}	S_{f2}	S_{f1}	S_{f2}	S_{f1}	S_{f2}
-7		0.041196	0.025943	0.021231	0.014170	0.018117	0.010343
-6	Stresses	0.042750	0.030482	0.021518	0.015627	0.020512	0.009820
-5	to	0.048827	0.033764	0.024015	0.017105	0.020890	0.011564
-4	the	0.064772	0.051740	0.037468	0.029208	0.029045	0.017982
-3	Left.	0.066380	0.085730	0.031511	0.038516	0.017577	0.033095
-2		0.100676	0.118852	0.072824	0.083446	0.020557	0.056947
-1		0.374650	0.280030	0.217383	0.125936	0.292535	0.211813
0	Current Stress	0.931302	0.689304	0.736214	0.488113	0.931302	0.689304
+1		0.511399	0.347710	0.286095	0.166143	0.400033	0.294165
+2		0.124193	0.097700	0.074989	0.056482	0.011864	0.032351
+3	Stresses	0.077051	0.038706	0.038072	0.017718	0.024235	0.005831
+4	to	0.069031	0.032249	0.037639	0.021100	0.025004	0.007082
+5	the	0.050129	0.027898	0.025220	0.014402	0.014294	0.009311
+6	right	0.038508	0.028228	0.017021	0.009439	0.013430	0.012100
+7		0.031345	0.015665	0.014709	0.006930	0.011386	0.005216

Table 93. Mutual information data between the letters *i* positions away from the current letter (in a 15-letter window) and the output class of the current letter when ***simplified stresses*** are employed. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only.

Letter position *i*	Which side	Output Considered: Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7	Letters to the Left.	0.144952	0.120230	0.021559
-6		0.189612	0.146963	0.022161
-5		0.231189	0.172124	0.034464
-4		0.282440	0.209157	0.042405
-3		0.342375	0.258011	0.046520
-2		0.445944	0.361062	0.052500
-1		0.885255	0.799755	0.162515
0	Current Letter	3.295792	3.250498	0.295007
+1	Letters to the right	1.027048	0.937440	0.129404
+2		0.541750	0.439971	0.091921
+3		0.351334	0.279319	0.033462
+4		0.280576	0.211752	0.026638
+5		0.225033	0.169647	0.011880
+6		0.187223	0.136780	0.012363
+7		0.155412	0.109149	0.017902

Table 94. Mutual information data between *phonemes* corresponding to letters *i* positions away from the current letter (in a 15-letter window) and the output class of the current letter when *simplified stresses* are employed. Three values are shown for each phoneme position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only.

Phoneme position *i*	Which side	Output Considered: Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.168572	0.142476	0.023400
-6	Phonemes	0.227810	0.182300	0.026309
-5	to	0.285919	0.221002	0.039142
-4	the	0.358774	0.275235	0.049240
-3	Left.	0.447846	0.351652	0.057021
-2		0.606616	0.510260	0.090379
-1		1.103489	1.017784	0.173562
0	Current Phoneme	4.744520	4.744520	0.495562
+1		1.142475	1.038414	0.177778
+2		0.637798	0.529978	0.116034
+3	Phonemes	0.444175	0.350298	0.043911
+4	to	0.348521	0.268148	0.035083
+5	the	0.286341	0.217712	0.020079
+6	right	0.234836	0.175743	0.019206
+7		0.176188	0.129057	0.020552

Table 95. Mutual information data between *stresses* corresponding to letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter when *simplified stresses* are employed. Three values are shown for each stress position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only.

Stress position i	Which side	Output Considered		
		Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.025167	0.021979	0.004770
-6	Stresses	0.035141	0.025947	0.009218
-5	to	0.044225	0.026839	0.017224
-4	the	0.060901	0.036770	0.019946
-3	Left.	0.065243	0.043957	0.019687
-2		0.117843	0.092358	0.043098
-1		0.144014	0.134001	0.058652
0	Current Stress	0.835408	0.495562	0.835408
+1		0.180948	0.172494	0.053384
+2		0.085050	0.063808	0.021168
+3	Stresses	0.048548	0.027724	0.007624
+4	to	0.040354	0.026672	0.011921
+5	the	0.036561	0.017261	0.014226
+6	right	0.029850	0.013466	0.010800
+7		0.015277	0.010288	0.002167

Table 96. Mutual information data between the features L_{f1} and L_{f2} for letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter when simplified stresses are employed. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only. See Section 5.3.5 for a definition of L_{f1} and L_{f2}.

Letter position	Which	Output Considered As					
		Phoneme Stress pairs		Phonemes Only		Stresses Only	
i	side	L_{f1}	L_{f2}	L_{f1}	L_{f2}	L_{f1}	L_{f2}
-7		0.038826	0.020788	0.031072	0.017257	0.008585	0.004627
-6	Letters	0.049083	0.022863	0.035753	0.017211	0.011715	0.004024
-5	to	0.062508	0.027236	0.044834	0.019882	0.016039	0.007193
-4	the	0.056164	0.035366	0.040088	0.025401	0.013555	0.007429
-3	Left.	0.069644	0.030348	0.050462	0.024385	0.018528	0.003467
-2		0.062380	0.060176	0.057989	0.048807	0.005661	0.006782
-1		0.331755	0.190556	0.319242	0.183141	0.107688	0.073356
0	Current Letter	0.844558	0.594100	0.837232	0.573052	0.274116	0.105774
+1		0.337031	0.186358	0.324966	0.181176	0.096008	0.033595
+2		0.102362	0.089928	0.084425	0.078051	0.007612	0.014406
+3	Letters	0.067439	0.038467	0.060366	0.024901	0.001304	0.008609
+4	to	0.052512	0.028563	0.038132	0.023958	0.006167	0.000441
+5	the	0.043314	0.030022	0.037442	0.020840	0.002085	0.000965
+6	right	0.036879	0.022974	0.031173	0.016252	0.000624	0.001615
+7		0.025841	0.026876	0.018417	0.018836	0.001887	0.004260

Table 97. Mutual information data between the feature P_{f1} for *phonemes* corresponding to letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter when simplified stresses are employed. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only. See Section 5.3.5 for a definition of P_{f1}.

Phoneme		Output Considered		
position i	Which side	Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.007030	0.005593	0.000720
-6	Phonemes	0.010451	0.008119	0.001089
-5	to	0.011639	0.007130	0.001963
-4	the	0.020811	0.014363	0.003632
-3	Left.	0.020506	0.015795	0.003684
-2		0.042164	0.033388	0.006898
-1		0.051296	0.048753	0.016073
0	Current Phoneme	0.505651	0.505651	0.127744
+1		0.091985	0.089839	0.024259
+2		0.039340	0.031707	0.001018
+3	Phonemes	0.017526	0.012418	0.000204
+4	to	0.019370	0.016432	0.001616
+5	the	0.012327	0.009690	0.000306
+6	right	0.013483	0.008527	0.001183
+7		0.008678	0.006146	0.000715

Table 98. Mutual information data between the feature S_{f1} for *stresses* corresponding to letters i positions away from the current letter (in a 15-letter window) and the output class of the current letter when simplified stresses are employed. Three values are shown for each letter position corresponding to the output being considered as phoneme/stress pairs, phonemes only or stresses only. See Section 5.3.5 for a definition of S_{f1}.

Phoneme position i	Which side	Output Considered: Phoneme/ Stress pairs	Phonemes Only	Stresses Only
-7		0.016550	0.014170	0.003534
-6	Stresses	0.019564	0.015627	0.000995
-5	to	0.024799	0.017105	0.004791
-4	the	0.045358	0.029208	0.011276
-3	Left.	0.057404	0.038516	0.017514
-2		0.108265	0.083446	0.042774
-1		0.135776	0.125936	0.058599
0	Current Stress	0.689304	0.488113	0.689304
+1		0.174464	0.166143	0.053094
+2		0.073258	0.056482	0.018132
+3	Stresses	0.029009	0.017718	0.001878
+4	to	0.025021	0.021100	0.001905
+5	the	0.022413	0.014402	0.003468
+6	right	0.019805	0.009439	0.004505
+7		0.010880	0.006930	0.001050