Speech Recognition using Neural Networks

Joe Tebelskis

May 1995
CMU-CS-95-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Submitted in partial fulfillment of the requirements for
a degree of Doctor of Philosophy in Computer Science

Thesis Committee:

Alex Waibel, chair
Raj Reddy
Jaime Carbonell
Richard Lippmann, MIT Lincoln Labs

Copyright ©1995 Joe Tebelskis

This research was supported during separate phases by ATR Interpreting Telephony Research Laboratories, NEC Corporation, Siemens AG, the National Science Foundation, the Advanced Research Projects Administration, and the Department of Defense under Contract No. MDA904-92-C-5161.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ATR, NEC, Siemens, NSF, or the United States Government.

Keywords: Speech recognition, neural networks, hidden Markov models, hybrid systems, acoustic modeling, prediction, classification, probability estimation, discrimination, global optimization.

Abstract

This thesis examines how artificial neural networks can benefit a large vocabulary, speaker independent, continuous speech recognition system. Currently, most speech recognition systems are based on hidden Markov models (HMMs), a statistical framework that supports both acoustic and temporal modeling. Despite their state-of-the-art performance, HMMs make a number of suboptimal modeling assumptions that limit their potential effectiveness. Neural networks avoid many of these assumptions, while they can also learn complex functions, generalize effectively, tolerate noise, and support parallelism. While neural networks can readily be applied to acoustic modeling, it is not yet clear how they can be used for temporal modeling. Therefore, we explore a class of systems called *NN-HMM hybrids*, in which neural networks perform acoustic modeling, and HMMs perform temporal modeling. We argue that a NN-HMM hybrid has several theoretical advantages over a pure HMM system, including better acoustic modeling accuracy, better context sensitivity, more natural discrimination, and a more economical use of parameters. These advantages are confirmed experimentally by a NN-HMM hybrid that we developed, based on context-independent phoneme models, that achieved 90.5% word accuracy on the Resource Management database, in contrast to only 86.0% accuracy achieved by a pure HMM under similar conditions.

In the course of developing this system, we explored two different ways to use neural networks for acoustic modeling: prediction and classification. We found that predictive networks yield poor results because of a lack of discrimination, but classification networks gave excellent results. We verified that, in accordance with theory, the output activations of a classification network form highly accurate estimates of the posterior probabilities $P(class|input)$, and we showed how these can easily be converted to likelihoods $P(input|class)$ for standard HMM recognition algorithms. Finally, this thesis reports how we optimized the accuracy of our system with many natural techniques, such as expanding the input window size, normalizing the inputs, increasing the number of hidden units, converting the network's output activations to log likelihoods, optimizing the learning rate schedule by automatic search, backpropagating error from word level outputs, and using gender dependent networks.

Acknowledgements

I wish to thank Alex Waibel for the guidance, encouragement, and friendship that he managed to extend to me during our six years of collaboration over all those inconvenient oceans — and for his unflagging efforts to provide a world-class, international research environment, which made this thesis possible. Alex's scientific integrity, humane idealism, good cheer, and great ambition have earned him my respect, plus a standing invitation to dinner whenever he next passes through my corner of the world. I also wish to thank Raj Reddy, Jaime Carbonell, and Rich Lippmann for serving on my thesis committee and offering their valuable suggestions, both on my thesis proposal and on this final dissertation. I would also like to thank Scott Fahlman, my first advisor, for channeling my early enthusiasm for neural networks, and teaching me what it means to do good research.

Many colleagues around the world have influenced this thesis, including past and present members of the Boltzmann Group, the NNSpeech Group at CMU, and the NNSpeech Group at the University of Karlsruhe in Germany. I especially want to thank my closest collaborators over these years — Bojan Petek, Otto Schmidbauer, Torsten Zeppenfeld, Hermann Hild, Patrick Haffner, Arthur McNair, Tilo Sloboda, Monika Woszczyna, Ivica Rogina, Michael Finke, and Thorsten Schueler — for their contributions and their friendship. I also wish to acknowledge valuable interactions I've had with many other talented researchers, including Fil Alleva, Uli Bodenhausen, Herve Bourlard, Lin Chase, Mike Cohen, Mark Derthick, Mike Franzini, Paul Gleichauff, John Hampshire, Nobuo Hataoka, Geoff Hinton, Xuedong Huang, Mei-Yuh Hwang, Ken-ichi Iso, Ajay Jain, Yochai Konig, George Lakoff, Kevin Lang, Chris Lebiere, Kai-Fu Lee, Ester Levin, Stefan Manke, Jay McClelland, Chris McConnell, Abdelhamid Mellouk, Nelson Morgan, Barak Pearlmutter, Dave Plaut, Dean Pomerleau, Steve Renals, Roni Rosenfeld, Dave Rumelhart, Dave Sanner, Hidefumi Sawai, David Servan-Schreiber, Bernhard Suhm, Sebastian Thrun, Dave Touretzky, Minh Tue Voh, Wayne Ward, Christoph Windheuser, and Michael Witbrock. I am especially indebted to Yochai Konig at ICSI, who was extremely generous in helping me to understand and reproduce ICSI's experimental results; and to Arthur McNair for taking over the Janus demos in 1992 so that I could focus on my speech research, and for constantly keeping our environment running so smoothly. Thanks to Hal McCarter and his colleagues at Adaptive Solutions for their assistance with the CNAPS parallel computer; and to Nigel Goddard at the Pittsburgh Supercomputer Center for help with the Cray C90. Thanks to Roni Rosenfeld, Lin Chase, and Michael Finke for proofreading portions of this thesis.

I am also grateful to Robert Wilensky for getting me started in Artificial Intelligence, and especially to both Douglas Hofstadter and Allen Newell for sharing some treasured, pivotal hours with me.

Many friends helped me maintain my sanity during the PhD program, as I felt myself drowning in this overambitious thesis. I wish to express my love and gratitude especially to Bart Reynolds, Sara Fried, Mellen Lovrin, Pam Westin, Marilyn & Pete Fast, Susan Wheeler, Gowthami Rajendran, I-Chen Wu, Roni Rosenfeld, Simona & George Necula, Francesmary Modugno, Jade Goldstein, Hermann Hild, Michael Finke, Kathie Porsche, Phyllis Reuther, Barbara White, Bojan & Davorina Petek, Anne & Scott Westbrook, Richard Weinapple, Marv Parsons, and Jeanne Sheldon. I have also prized the friendship of Catherine Copetas, Prasad Tadepalli, Hanna Djajapranata, Arthur McNair, Torsten Zeppenfeld, Tilo Sloboda, Patrick Haffner, Mark Maimone, Spiro Michaylov, Prasad Chalisani, Angela Hickman, Lin Chase, Steve Lawson, Dennis & Bonnie Lunder, and too many others to list. Without the support of my friends, I might not have finished the PhD.

I wish to thank my parents, Virginia and Robert Tebelskis, for having raised me in such a stable and loving environment, which has enabled me to come so far. I also thank the rest of my family & relatives for their love.

This thesis is dedicated to Douglas Hofstadter, whose book “Godel, Escher, Bach” changed my life by suggesting how consciousness can emerge from subsymbolic computation, shaping my deepest beliefs and inspiring me to study Connectionism; and to the late Allen Newell, whose genius, passion, warmth, and humanity made him a beloved role model whom I could only dream of emulating, and whom I now sorely miss.

Table of Contents

Abstract iii

Acknowledgements v

1 Introduction. 1
1.1 Speech Recognition 2
1.2 Neural Networks. 4
1.3 Thesis Outline. 7

2 Review of Speech Recognition 9
2.1 Fundamentals of Speech Recognition 9
2.2 Dynamic Time Warping 14
2.3 Hidden Markov Models 15
2.3.1 Basic Concepts 16
2.3.2 Algorithms 17
2.3.3 Variations 22
2.3.4 Limitations of HMMs. 26

3 Review of Neural Networks 27
3.1 Historical Development 27
3.2 Fundamentals of Neural Networks. 28
3.2.1 Processing Units. 28
3.2.2 Connections 29
3.2.3 Computation. 30
3.2.4 Training 35
3.3 A Taxonomy of Neural Networks 36
3.3.1 Supervised Learning. 37
3.3.2 Semi-Supervised Learning 40
3.3.3 Unsupervised Learning. 41
3.3.4 Hybrid Networks 43
3.3.5 Dynamic Networks. 43
3.4 Backpropagation. 44
3.5 Relation to Statistics. 48

4 Related Research 51
4.1 Early Neural Network Approaches 51
4.1.1 Phoneme Classification. 52
4.1.2 Word Classification 55
4.2 The Problem of Temporal Structure 56
4.3 NN-HMM Hybrids 57
4.3.1 NN Implementations of HMMs 57
4.3.2 Frame Level Training 58
4.3.3 Segment Level Training 60
4.3.4 Word Level Training 61
4.3.5 Global Optimization 62
4.3.6 Context Dependence 63
4.3.7 Speaker Independence 66
4.3.8 Word Spotting 69
4.4 Summary 71

5 Databases 73
5.1 Japanese Isolated Words 73
5.2 Conference Registration 74
5.3 Resource Management 75

6 Predictive Networks 77
6.1 Motivation... and Hindsight 78
6.2 Related Work 79
6.3 Linked Predictive Neural Networks 81
6.3.1 Basic Operation. 81
6.3.2 Training the LPNN 82
6.3.3 Isolated Word Recognition Experiments 84
6.3.4 Continuous Speech Recognition Experiments 86
6.3.5 Comparison with HMMs 88
6.4 Extensions 89
6.4.1 Hidden Control Neural Network. 89
6.4.2 Context Dependent Phoneme Models. 92
6.4.3 Function Word Models 94
6.5 Weaknesses of Predictive Networks 94
6.5.1 Lack of Discrimination 94
6.5.2 Inconsistency 98

7 Classification Networks 101
7.1 Overview 101
7.2 Theory 103
7.2.1 The MLP as a Posterior Estimator 103
7.2.2 Likelihoods vs. Posteriors 105
7.3 Frame Level Training 106
7.3.1 Network Architectures 106
7.3.2 Input Representations 115
7.3.3 Speech Models 119
7.3.4 Training Procedures 120
7.3.5 Testing Procedures 132
7.3.6 Generalization 137
7.4 Word Level Training 138
7.4.1 Multi-State Time Delay Neural Network 138
7.4.2 Experimental Results 141
7.5 Summary 143

8 Comparisons 147
8.1 Conference Registration Database 147
8.2 Resource Management Database 148

9 Conclusions 151
9.1 Neural Networks as Acoustic Models 151
9.2 Summary of Experiments 152
9.3 Advantages of NN-HMM hybrids 153

Appendix A. Final System Design 155

Appendix B. Proof that Classifier Networks Estimate Posterior Probabilities. 157

Bibliography 159

Author Index 169

Subject Index 173

1. Introduction

Speech is a natural mode of communication for people. We learn all the relevant skills during early childhood, without instruction, and we continue to rely on speech communication throughout our lives. It comes so naturally to us that we don't realize how complex a phenomenon speech is. The human vocal tract and articulators are biological organs with nonlinear properties, whose operation is not just under conscious control but also affected by factors ranging from gender to upbringing to emotional state. As a result, vocalizations can vary widely in terms of their accent, pronunciation, articulation, roughness, nasality, pitch, volume, and speed; moreover, during transmission, our irregular speech patterns can be further distorted by background noise and echoes, as well as electrical characteristics (if telephones or other electronic equipment are used). All these sources of variability make speech recognition, even more than speech generation, a very complex problem.

Yet people are so comfortable with speech that we would also like to interact with our computers via speech, rather than having to resort to primitive interfaces such as keyboards and pointing devices. A speech interface would support many valuable applications — for example, telephone directory assistance, spoken database querying for novice users, "hands-busy" applications in medicine or fieldwork, office dictation devices, or even automatic voice translation into foreign languages. Such tantalizing applications have motivated research in automatic speech recognition since the 1950's. Great progress has been made so far, especially since the 1970's, using a series of engineered approaches that include template matching, knowledge engineering, and statistical modeling. Yet computers are still nowhere near the level of human performance at speech recognition, and it appears that further significant advances will require some new insights.

What makes people so good at recognizing speech? Intriguingly, the human brain is known to be wired differently than a conventional computer; in fact it operates under a radically different computational paradigm. While conventional computers use a very fast & complex central processor with explicit program instructions and locally addressable memory, by contrast the human brain uses a massively parallel collection of slow & simple processing elements (neurons), densely connected by weights (synapses) whose strengths are modified with experience, directly supporting the integration of multiple constraints, and providing a distributed form of associative memory.

The brain's impressive superiority at a wide range of cognitive skills, including speech recognition, has motivated research into its novel computational paradigm since the 1940's, on the assumption that brainlike models may ultimately lead to brainlike performance on many complex tasks. This fascinating research area is now known as *connectionism*, or the study of *artificial neural networks*. The history of this field has been erratic (and laced with

hyperbole), but by the mid-1980's, the field had matured to a point where it became realistic to begin applying connectionist models to difficult tasks like speech recognition. By 1990 (when this thesis was proposed), many researchers had demonstrated the value of neural networks for important subtasks like phoneme recognition and spoken digit recognition, but it was still unclear whether connectionist techniques would scale up to large speech recognition tasks.

This thesis demonstrates that neural networks can indeed form the basis for a general purpose speech recognition system, and that neural networks offer some clear advantages over conventional techniques.

1.1. Speech Recognition

What is the current state of the art in speech recognition? This is a complex question, because a system's accuracy depends on the conditions under which it is evaluated: under sufficiently narrow conditions almost any system can attain human-like accuracy, but it's much harder to achieve good accuracy under general conditions. The conditions of evaluation — and hence the accuracy of any system — can vary along the following dimensions:

- **Vocabulary size and confusability**. As a general rule, it is easy to discriminate among a small set of words, but error rates naturally increase as the vocabulary size grows. For example, the 10 digits "zero" to "nine" can be recognized essentially perfectly (Doddington 1989), but vocabulary sizes of 200, 5000, or 100000 may have error rates of 3%, 7%, or 45% (Itakura 1975, Miyatake 1990, Kimura 1990). On the other hand, even a small vocabulary can be hard to recognize if it contains confusable words. For example, the 26 letters of the English alphabet (treated as 26 "words") are very difficult to discriminate because they contain so many confusable words (most notoriously, the E-set: "B, C, D, E, G, P, T, V, Z"); an 8% error rate is considered good for this vocabulary (Hild & Waibel 1993).
- **Speaker dependence vs. independence.** By definition, a *speaker dependent* system is intended for use by a single speaker, but a *speaker independent* system is intended for use by any speaker. Speaker independence is difficult to achieve because a system's parameters become tuned to the speaker(s) that it was trained on, and these parameters tend to be highly speaker-specific. Error rates are typically 3 to 5 times higher for speaker independent systems than for speaker dependent ones (Lee 1988). Intermediate between speaker dependent and independent systems, there are also *multi-speaker* systems intended for use by a small group of people, and *speaker-adaptive* systems which tune themselves to any speaker given a small amount of their speech as enrollment data.
- **Isolated, discontinuous, or continuous speech.** *Isolated speech* means single words; *discontinuous speech* means full sentences in which words are artificially separated by silence; and *continuous speech* means naturally spoken sentences. Isolated and discontinuous speech recognition is relatively easy because word boundaries are detectable and the words tend to be cleanly pronounced. Continu-

ous speech is more difficult, however, because word boundaries are unclear and their pronunciations are more corrupted by *coarticulation*, or the slurring of speech sounds, which for example causes a phrase like "could you" to sound like "could jou". In a typical evaluation, the word error rates for isolated and continuous speech were 3% and 9%, respectively (Bahl et al 1981).

- **Task and language constraints.** Even with a fixed vocabulary, performance will vary with the nature of constraints on the word sequences that are allowed during recognition. Some constraints may be *task-dependent* (for example, an airline-querying application may dismiss the hypothesis "The apple is red"); other constraints may be *semantic* (rejecting "The apple is angry"), or *syntactic* (rejecting "Red is apple the"). Constraints are often represented by a *grammar*, which ideally filters out unreasonable sentences so that the speech recognizer evaluates only plausible sentences. Grammars are usually rated by their *perplexity*, a number that indicates the grammar's average branching factor (i.e., the number of words that can follow any given word). The difficulty of a task is more reliably measured by its perplexity than by its vocabulary size.
- **Read vs. spontaneous speech.** Systems can be evaluated on speech that is either read from prepared scripts, or speech that is uttered spontaneously. Spontaneous speech is vastly more difficult, because it tends to be peppered with disfluencies like "uh" and "um", false starts, incomplete sentences, stuttering, coughing, and laughter; and moreover, the vocabulary is essentially unlimited, so the system must be able to deal intelligently with unknown words (e.g., detecting and flagging their presence, and adding them to the vocabulary, which may require some interaction with the user).
- **Adverse conditions.** A system's performance can also be degraded by a range of adverse conditions (Furui 1993). These include environmental noise (e.g., noise in a car or a factory); acoustical distortions (e.g, echoes, room acoustics); different microphones (e.g., close-speaking, omnidirectional, or telephone); limited frequency bandwidth (in telephone transmission); and altered speaking manner (shouting, whining, speaking quickly, etc.).

In order to evaluate and compare different systems under well-defined conditions, a number of standardized databases have been created with particular characteristics. For example, one database that has been widely used is the DARPA Resource Management database — a large vocabulary (1000 words), speaker-independent, continuous speech database, consisting of 4000 training sentences in the domain of naval resource management, read from a script and recorded under benign environmental conditions; testing is usually performed using a grammar with a perplexity of 60. Under these controlled conditions, state-of-the-art performance is about 97% word recognition accuracy (or less for simpler systems). We used this database, as well as two smaller ones, in our own research (see Chapter 5).

The central issue in speech recognition is dealing with variability. Currently, speech recognition systems distinguish between two kinds of variability: acoustic and temporal. *Acoustic variability* covers different accents, pronunciations, pitches, volumes, and so on,

while *temporal variability* covers different speaking rates. These two dimensions are not completely independent — when a person speaks quickly, his acoustical patterns become distorted as well — but it's a useful simplification to treat them independently.

Of these two dimensions, temporal variability is easier to handle. An early approach to temporal variability was to linearly stretch or shrink (*"warp"*) an unknown utterance to the duration of a known template. Linear warping proved inadequate, however, because utterances can accelerate or decelerate at any time; instead, nonlinear warping was obviously required. Soon an efficient algorithm known as *Dynamic Time Warping* was proposed as a solution to this problem. This algorithm (in some form) is now used in virtually every speech recognition system, and the problem of temporal variability is considered to be largely solved[1].

Acoustic variability is more difficult to model, partly because it is so heterogeneous in nature. Consequently, research in speech recognition has largely focused on efforts to model acoustic variability. Past approaches to speech recognition have fallen into three main categories:

1. **Template-based approaches**, in which unknown speech is compared against a set of prerecorded words (*templates*), in order to find the best match. This has the advantage of using perfectly accurate word models; but it also has the disadvantage that the prerecorded templates are fixed, so variations in speech can only be modeled by using many templates per word, which eventually becomes impractical.

2. **Knowledge-based approaches**, in which "expert" knowledge about variations in speech is hand-coded into a system. This has the advantage of explicitly modeling variations in speech; but unfortunately such expert knowledge is difficult to obtain and use successfully, so this approach was judged to be impractical, and automatic learning procedures were sought instead.

3. **Statistical-based approaches**, in which variations in speech are modeled statistically (e.g., by *Hidden Markov Models*, or *HMMs*), using automatic learning procedures. This approach represents the current state of the art. The main disadvantage of statistical models is that they must make a priori modeling assumptions, which are liable to be inaccurate, handicapping the system's performance. We will see that neural networks help to avoid this problem.

1.2. Neural Networks

Connectionism, or the study of artificial neural networks, was initially inspired by neurobiology, but it has since become a very interdisciplinary field, spanning computer science, electrical engineering, mathematics, physics, psychology, and linguistics as well. Some researchers are still studying the neurophysiology of the human brain, but much attention is

1. Although there remain unresolved secondary issues of duration constraints, speaker-dependent speaking rates, etc.

now being focused on the general properties of neural computation, using simplified neural models. These properties include:

- **Trainability.** Networks can be taught to form associations between any input and output patterns. This can be used, for example, to teach the network to classify speech patterns into phoneme categories.
- **Generalization.** Networks don't just memorize the training data; rather, they learn the underlying patterns, so they can generalize from the training data to new examples. This is essential in speech recognition, because acoustical patterns are never exactly the same.
- **Nonlinearity.** Networks can compute nonlinear, nonparametric functions of their input, enabling them to perform arbitrarily complex transformations of data. This is useful since speech is a highly nonlinear process.
- **Robustness.** Networks are tolerant of both physical damage and noisy data; in fact noisy data can help the networks to form better generalizations. This is a valuable feature, because speech patterns are notoriously noisy.
- **Uniformity.** Networks offer a uniform computational paradigm which can easily integrate constraints from different types of inputs. This makes it easy to use both basic and differential speech inputs, for example, or to combine acoustic and visual cues in a multimodal system.
- **Parallelism.** Networks are highly parallel in nature, so they are well-suited to implementations on massively parallel computers. This will ultimately permit very fast processing of speech or other data.

There are many types of connectionist models, with different architectures, training procedures, and applications, but they are all based on some common principles. An artificial neural network consists of a potentially large number of simple processing elements (called *units*, *nodes*, or *neurons*), which influence each other's behavior via a network of excitatory or inhibitory weights. Each unit simply computes a nonlinear weighted sum of its inputs, and broadcasts the result over its outgoing connections to other units. A training set consists of patterns of values that are assigned to designated input and/or output units. As patterns are presented from the training set, a learning rule modifies the strengths of the weights so that the network gradually learns the training set. This basic paradigm[1] can be fleshed out in many different ways, so that different types of networks can learn to compute implicit functions from input to output vectors, or automatically cluster input data, or generate compact representations of data, or provide content-addressable memory and perform pattern completion.

1. Many biological details are ignored in these simplified models. For example, biological neurons produce a sequence of pulses rather than a stable activation value; there exist several different types of biological neurons; their physical geometry can affect their computational behavior; they operate asynchronously, and have different cycle times; and their behavior is affected by hormones and other chemicals. Such details may ultimately prove necessary for modeling the brain's behavior, but for now even the simplified model has enough computational power to support very interesting research.

Neural networks are usually used to perform static pattern recognition, that is, to statically map complex inputs to simple outputs, such as an N-ary classification of the input patterns. Moreover, the most common way to train a neural network for this task is via a procedure called *backpropagation* (Rumelhart et al, 1986), whereby the network's weights are modified in proportion to their contribution to the observed error in the output unit activations (relative to desired outputs). To date, there have been many successful applications of neural networks trained by backpropagation. For instance:

- *NETtalk* (Sejnowski and Rosenberg, 1987) is a neural network that learns how to pronounce English text. Its input is a window of 7 characters (orthographic text symbols), scanning a larger text buffer, and its output is a phoneme code (relayed to a speech synthesizer) that tells how to pronounce the middle character in that context. During successive cycles of training on 1024 words and their pronunciations, NETtalk steadily improved is performance like a child learning how to talk, and it eventually produced quite intelligible speech, even on words that it had never seen before.
- *Neurogammon* (Tesauro 1989) is a neural network that learns a winning strategy for Backgammon. Its input describes the current position, the dice values, and a possible move, and its output represents the merit of that move, according to a training set of 3000 examples hand-scored by an expert player. After sufficient training, the network generalized well enough to win the gold medal at the computer olympiad in London, 1989, defeating five commercial and two non-commercial programs, although it lost to a human expert.
- *ALVINN* (Pomerleau 1993) is a neural network that learns how to drive a car. Its input is a coarse visual image of the road ahead (provided by a video camera and an imaging laser rangefinder), and its output is a continuous vector that indicates which way to turn the steering wheel. The system learns how to drive by observing how a person drives. ALVINN has successfully driven at speeds of up to 70 miles per hour for more than 90 miles, under a variety of different road conditions.
- *Handwriting recognition* (Le Cun et al, 1990) based on neural networks has been used to read ZIP codes on US mail envelopes. Size-normalized images of isolated digits, found by conventional algorithms, are fed to a highly constrained neural network, which transforms each visual image to one of 10 class outputs. This system has achieved 92% digit recognition accuracy on actual mail provided by the US Postal Service. A more elaborate system by Bodenhausen and Manke (1993) has achieved up to 99.5% digit recognition accuracy on another database.

Speech recognition, of course, has been another proving ground for neural networks. Researchers quickly achieved excellent results in such basic tasks as voiced/unvoiced discrimination (Watrous 1988), phoneme recognition (Waibel et al, 1989), and spoken digit recognition (Franzini et al, 1989). However, in 1990, when this thesis was proposed, it still remained to be seen whether neural networks could support a large vocabulary, speaker independent, continuous speech recognition system.

In this thesis we take an incremental approach to this problem. Of the two types of variability in speech — acoustic and temporal — the former is more naturally posed as a static

pattern matching problem that is amenable to neural networks; therefore we use neural networks for acoustic modeling, while we rely on conventional Hidden Markov Models for temporal modeling. Our research thus represents an exploration of the space of *NN-HMM hybrids*. We explore two different ways to use neural networks for acoustic modeling, namely *prediction* and *classification* of the speech patterns. Prediction is shown to be a weak approach because it lacks discrimination, while classification is shown to be a much stronger approach. We present an extensive series of experiments that we performed to optimize our networks for word recognition accuracy, and show that a properly optimized NN-HMM hybrid system based on classification networks can outperform other systems under similar conditions. Finally, we argue that hybrid NN-HMM systems offer several advantages over pure HMM systems, including better acoustic modeling accuracy, better context sensitivity, more natural discrimination, and a more economical use of parameters.

1.3. Thesis Outline

The first few chapters of this thesis provide some essential background and a summary of related work in speech recognition and neural networks:

- Chapter 2 reviews the field of speech recognition.
- Chapter 3 reviews the field of neural networks.
- Chapter 4 reviews the intersection of these two fields, summarizing both past and present approaches to speech recognition using neural networks.

The remainder of the thesis describes our own research, evaluating both predictive networks and classification networks as acoustic models in NN-HMM hybrid systems:

- Chapter 5 introduces the databases we used in our experiments.
- Chapter 6 presents our research with predictive networks, and explains why this approach yielded poor results.
- Chapter 7 presents our research with classification networks, and shows how we achieved excellent results through an extensive series of optimizations.
- Chapter 8 compares the performance of our optimized systems against many other systems on the same databases, demonstrating the value of NN-HMM hybrids.
- Chapter 9 presents the conclusions of this thesis.

2. Review of Speech Recognition

In this chapter we will present a brief review of the field of speech recognition. After reviewing some fundamental concepts, we will explain the standard Dynamic Time Warping algorithm, and then discuss Hidden Markov Models in some detail, offering a summary of the algorithms, variations, and limitations that are associated with this dominant technology.

2.1. Fundamentals of Speech Recognition

Speech recognition is a multileveled pattern recognition task, in which acoustical signals are examined and structured into a hierarchy of subword units (e.g., phonemes), words, phrases, and sentences. Each level may provide additional temporal constraints, e.g., known word pronunciations or legal word sequences, which can compensate for errors or uncertainties at lower levels. This hierarchy of constraints can best be exploited by combining decisions probabilistically at all lower levels, and making discrete decisions only at the highest level.

The structure of a standard speech recognition system is illustrated in Figure 2.1. The elements are as follows:

- **Raw speech**. Speech is typically sampled at a high frequency, e.g., 16 KHz over a microphone or 8 KHz over a telephone. This yields a sequence of amplitude values over time.
- **Signal analysis.** Raw speech should be initially transformed and compressed, in order to simplify subsequent processing. Many signal analysis techniques are available which can extract useful features and compress the data by a factor of ten without losing any important information. Among the most popular:
    - Fourier analysis (FFT) yields discrete frequencies over time, which can be interpreted visually. Frequencies are often distributed using a *Mel* scale, which is linear in the low range but logarithmic in the high range, corresponding to physiological characteristics of the human ear.
    - Perceptual Linear Prediction (PLP) is also physiologically motivated, but yields coefficients that cannot be interpreted visually.

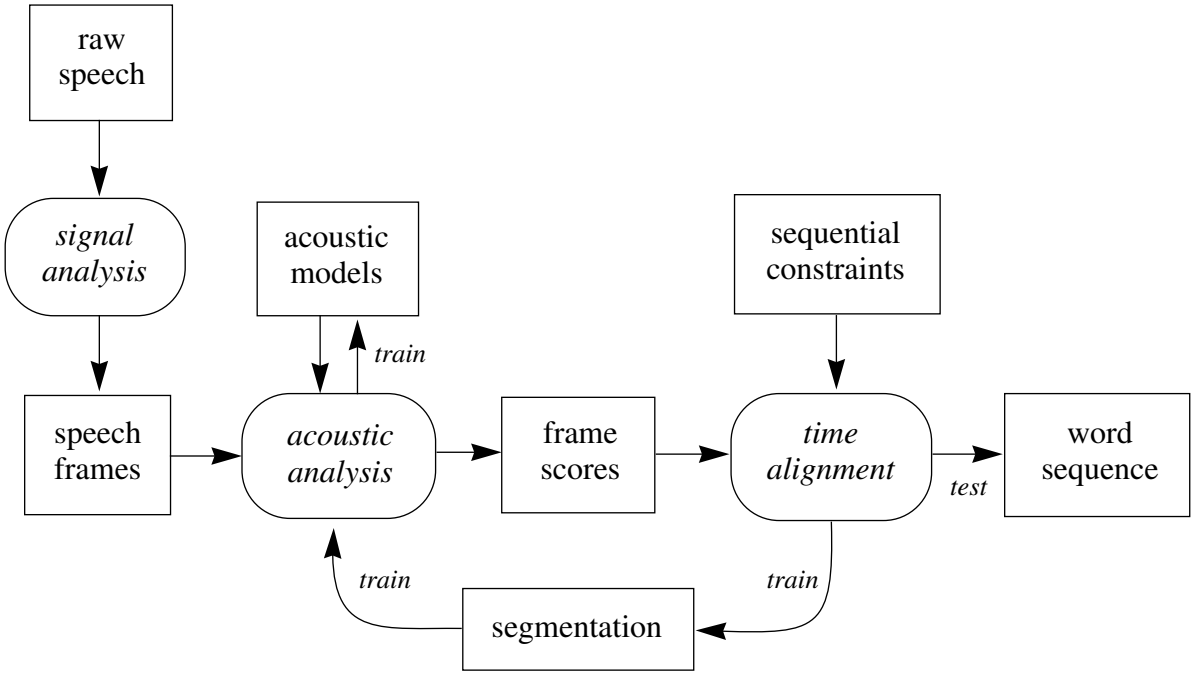

Figure 2.1: Structure of a standard speech recognition system.

- Linear Predictive Coding (LPC) yields coefficients of a linear equation that approximate the recent history of the raw speech values.
- Cepstral analysis calculates the inverse Fourier transform of the logarithm of the power spectrum of the signal.

In practice, it makes little difference which technique is used[1]. Afterwards, procedures such as Linear Discriminant Analysis (LDA) may optionally be applied to further reduce the dimensionality of any representation, and to decorrelate the coefficients.

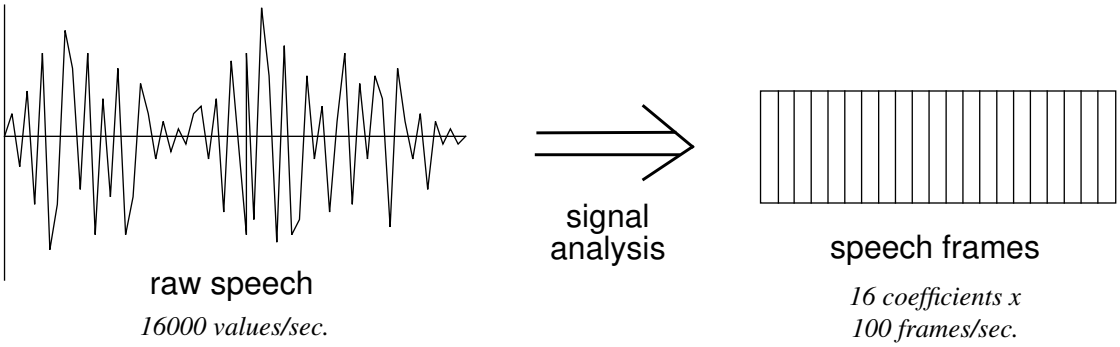

Figure 2.2: Signal analysis converts raw speech to speech frames.

1. Assuming benign conditions. Of course, each technique has its own advocates.

- **Speech frames.** The result of signal analysis is a sequence of *speech frames*, typically at 10 msec intervals, with about 16 coefficients per frame. These frames may be augmented by their own first and/or second derivatives, providing explicit information about speech dynamics; this typically leads to improved performance. The speech frames are used for acoustic analysis.
- **Acoustic models.** In order to analyze the speech frames for their acoustic content, we need a set of *acoustic models*. There are many kinds of acoustic models, varying in their representation, granularity, context dependence, and other properties.

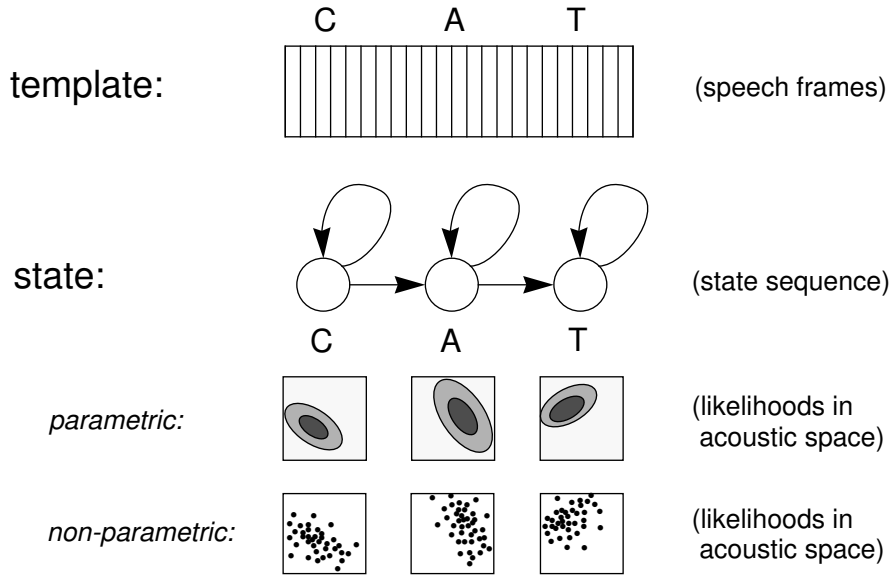

Figure 2.3: Acoustic models: template and state representations for the word "cat".

Figure 2.3 shows two popular representations for acoustic models. The simplest is a *template*, which is just a stored sample of the unit of speech to be modeled, e.g., a recording of a word. An unknown word can be recognized by simply comparing it against all known templates, and finding the closest match. Templates have two major drawbacks: (1) they cannot model acoustic variabilities, except in a coarse way by assigning multiple templates to each word; and (2) in practice they are limited to whole-word models, because it's hard to record or segment a sample shorter than a word — so templates are useful only in small systems which can afford the luxury of using whole-word models. A more flexible representation, used in larger systems, is based on trained acoustic models, or *states*. In this approach, every word is modeled by a sequence of trainable states, and each state indicates the sounds that are likely to be heard in that segment of the word, using a probability distribution over the acoustic space. Probability distributions can be modeled *parametrically*, by assuming that they have a simple shape (e.g., a Gaussian distribution) and then trying to find the parameters that describe it; or *non-parametrically*, by representing the distribution directly (e.g., with a histogram over a quantization of the acoustic space, or, as we shall see, with a neural network).

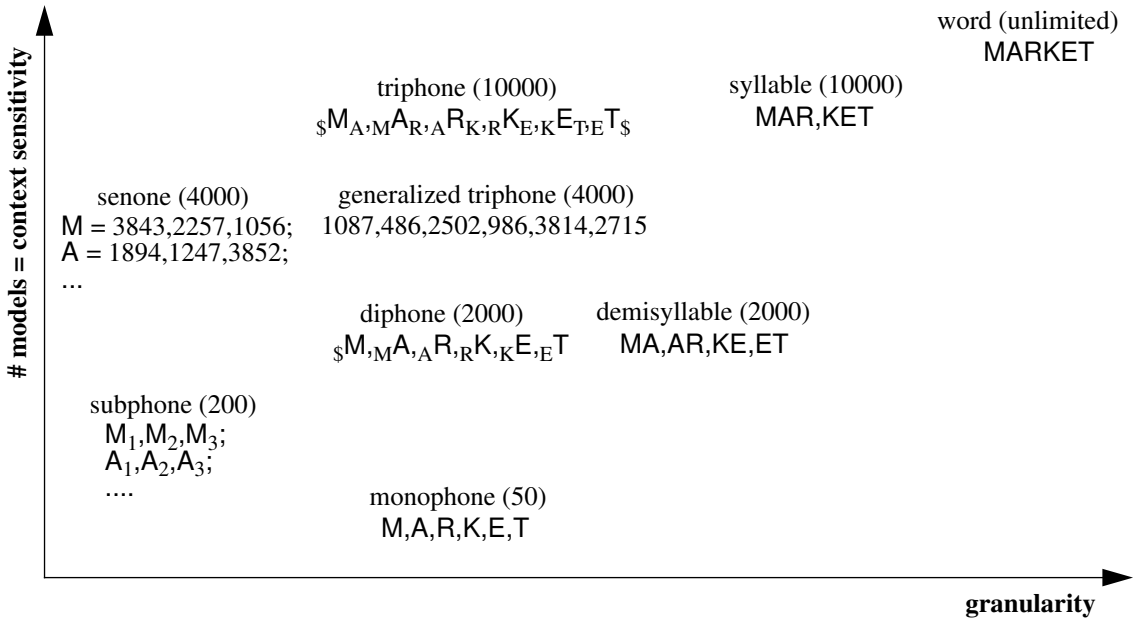

Figure 2.4: Acoustic models: granularity vs. context sensitivity, illustrated for the word "market".

Acoustic models also vary widely in their granularity and context sensitivity. Figure 2.4 shows a chart of some common types of acoustic models, and where they lie along these dimensions. As can be seen, models with larger granularity (such as *word* or *syllable* models) tend to have greater context sensitivity. Moreover, models with the greatest context sensitivity give the best word recognition accuracy —if those models are well trained. Unfortunately, the larger the granularity of a model, the poorer it will be trained, because fewer samples will be available for training it. For this reason, word and syllable models are rarely used in high-performance systems; much more common are *triphone* or *generalized triphone* models. Many systems also use *monophone* models (sometimes simply called *phoneme* models), because of their relative simplicity.

During training, the acoustic models are incrementally modified in order to optimize the overall performance of the system. During testing, the acoustic models are left unchanged.

- **Acoustic analysis and frame scores.** *Acoustic analysis* is performed by applying each acoustic model over each frame of speech, yielding a matrix of *frame scores*, as shown in Figure 2.5. Scores are computed according to the type of acoustic model that is being used. For template-based acoustic models, a score is typically the Euclidean distance between a template's frame and an unknown frame. For state-based acoustic models, a score represents an *emission probability*, i.e., the likelihood of the current state generating the current frame, as determined by the state's parametric or non-parametric function.
- **Time alignment.** Frame scores are converted to a word sequence by identifying a sequence of acoustic models, representing a valid word sequence, which gives the

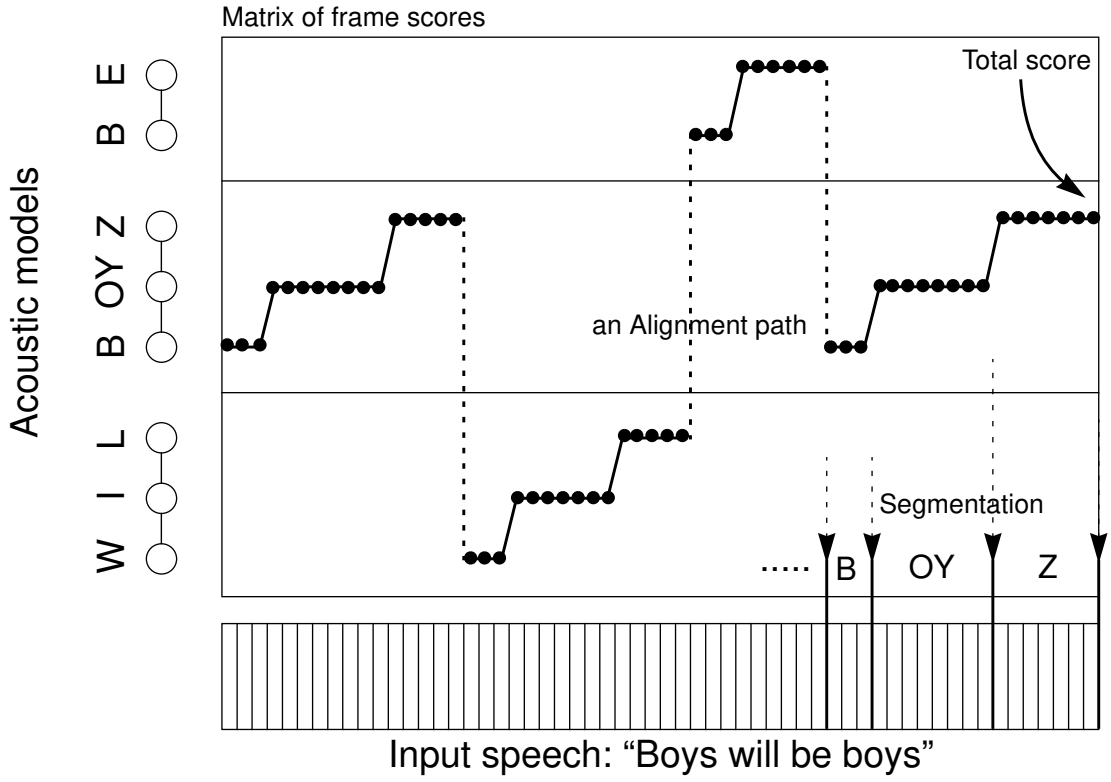

Figure 2.5: The alignment path with the best total score identifies the word sequence and segmentation.

best total score along an *alignment path* through the matrix[1], as illustrated in Figure 2.5. The process of searching for the best alignment path is called *time alignment*.

An alignment path must obey certain *sequential constraints* which reflect the fact that speech always goes forward, never backwards. These constraints are manifested both within and between words. Within a word, sequential constraints are implied by the sequence of frames (for template-based models), or by the sequence of states (for state-based models) that comprise the word, as dictated by the phonetic pronunciations in a dictionary, for example. Between words, sequential constraints are given by a grammar, indicating what words may follow what other words.

Time alignment can be performed efficiently by *dynamic programming*, a general algorithm which uses only local path constraints, and which has linear time and space requirements. (This general algorithm has two main variants, known as *Dynamic Time Warping* (DTW) and *Viterbi search*, which differ slightly in their local computations and in their optimality criteria.)

In a state-based system, the optimal alignment path induces a *segmentation* on the word sequence, as it indicates which frames are associated with each state. This

1. Actually, it is often better to evaluate a state sequence not by its single best alignment path, but by the composite score of all of its possible alignment paths; but we will ignore that issue for now.

segmentation can be used to generate labels for recursively training the acoustic models on corresponding frames.

- **Word sequence.** The end result of time alignment is a *word sequence* — the sentence hypothesis for the utterance. Actually it is common to return several such sequences, namely the ones with the highest scores, using a variation of time alignment called *N-best search* (Schwartz and Chow, 1990). This allows a recognition system to make two passes through the unknown utterance: the first pass can use simplified models in order to quickly generate an N-best list, and the second pass can use more complex models in order to carefully rescore each of the N hypotheses, and return the single best hypothesis.

2.2. Dynamic Time Warping

In this section we motivate and explain the *Dynamic Time Warping* algorithm, one of the oldest and most important algorithms in speech recognition (Vintsyuk 1971, Itakura 1975, Sakoe and Chiba 1978).

The simplest way to recognize an *isolated* word sample is to compare it against a number of stored word templates and determine which is the "best match". This goal is complicated by a number of factors. First, different samples of a given word will have somewhat different durations. This problem can be eliminated by simply normalizing the templates and the unknown speech so that they all have an equal duration. However, another problem is that the rate of speech may not be constant throughout the word; in other words, the optimal alignment between a template and the speech sample may be nonlinear. Dynamic Time Warping (DTW) is an efficient method for finding this optimal nonlinear alignment.

DTW is an instance of the general class of algorithms known as *dynamic programming*. Its time and space complexity is merely linear in the duration of the speech sample and the vocabulary size. The algorithm makes a single pass through a matrix of frame scores while computing locally optimized segments of the global alignment path. (See Figure 2.6.) If $D(x,y)$ is the Euclidean distance between frame x of the speech sample and frame y of the reference template, and if $C(x,y)$ is the cumulative score along an optimal alignment path that leads to (x,y), then

$$C(x, y) = \text{MIN}\left(C(x-1, y), C(x-1, y-1), C(x, y-1)\right) + D(x, y) \quad (1)$$

The resulting alignment path may be visualized as a low valley of Euclidean distance scores, meandering through the hilly landscape of the matrix, beginning at (0, 0) and ending at the final point (X, Y). By keeping track of backpointers, the full alignment path can be recovered by tracing backwards from (X, Y). An optimal alignment path is computed for each reference word template, and the one with the lowest cumulative score is considered to be the best match for the unknown speech sample.

There are many variations on the DTW algorithm. For example, it is common to vary the local path constraints, e.g., by introducing transitions with slope 1/2 or 2, or weighting the

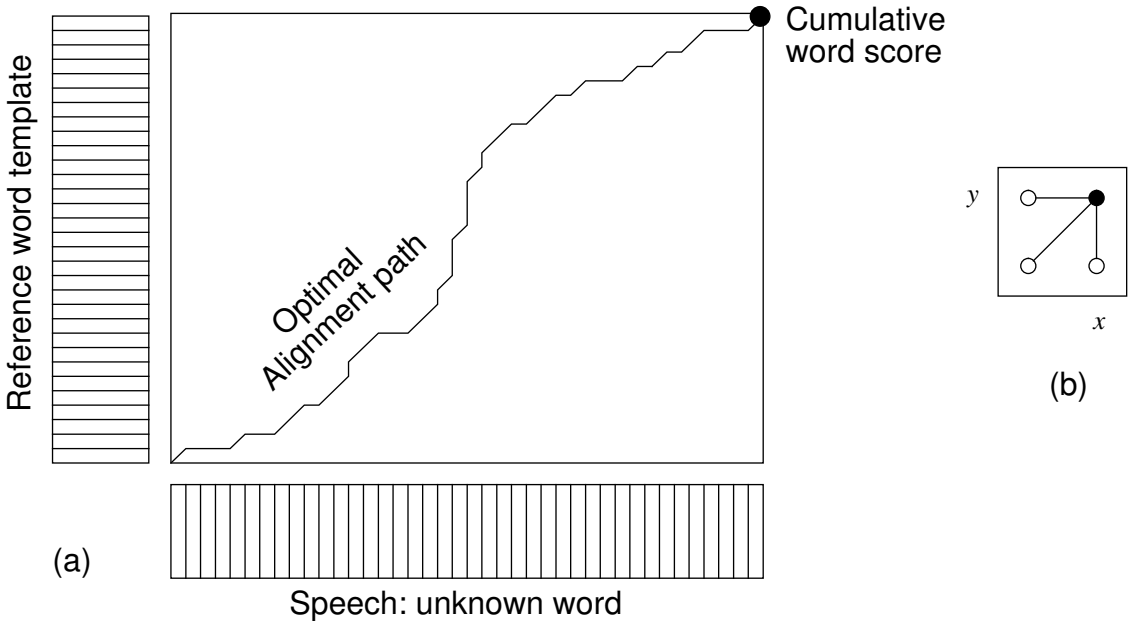

Figure 2.6: Dynamic Time Warping. (a) alignment path. (b) local path constraints.

transitions in various ways, or applying other kinds of slope constraints (Sakoe and Chiba 1978). While the reference word models are usually templates, they may be state-based models (as shown previously in Figure 2.5). When using states, vertical transitions are often disallowed (since there are fewer states than frames), and often the goal is to maximize the cumulative score, rather than to minimize it.

A particularly important variation of DTW is an extension from isolated to continuous speech. This extension is called the *One Stage DTW* algorithm (Ney 1984). Here the goal is to find the optimal alignment between the speech sample and the best sequence of reference words (see Figure 2.5). The complexity of the extended algorithm is still linear in the length of the sample and the vocabulary size. The only modification to the basic DTW algorithm is that at the beginning of each reference word model (i.e., its first frame or state), the diagonal path is allowed to point back to the end of all reference word models in the preceding frame. Local backpointers must specify the reference word model of the preceding point, so that the optimal word sequence can be recovered by tracing backwards from the final point (W, X, Y) of the word W with the best final score. Grammars can be imposed on continuous speech recognition by restricting the allowed transitions at word boundaries.

2.3. Hidden Markov Models

The most flexible and successful approach to speech recognition so far has been Hidden Markov Models (HMMs). In this section we will present the basic concepts of HMMs, describe the algorithms for training and using them, discuss some common variations, and review the problems associated with HMMs.

2.3.1. Basic Concepts

A Hidden Markov Model is a collection of states connected by transitions, as illustrated in Figure 2.7. It begins in a designated initial state. In each discrete time step, a transition is taken into a new state, and then one output symbol is generated in that state. The choice of transition and output symbol are both random, governed by probability distributions. The HMM can be thought of as a black box, where the sequence of output symbols generated over time is observable, but the sequence of states visited over time is hidden from view. This is why it's called a *Hidden* Markov Model.

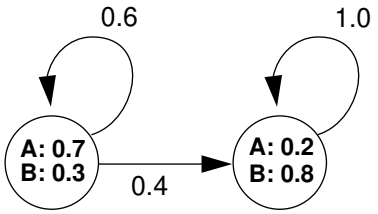

Figure 2.7: A simple Hidden Markov Model, with two states and two output symbols, A and B.

HMMs have a variety of applications. When an HMM is applied to speech recognition, the states are interpreted as acoustic models, indicating what sounds are likely to be heard during their corresponding segments of speech; while the transitions provide temporal constraints, indicating how the states may follow each other in sequence. Because speech always goes forward in time, transitions in a speech application always go forward (or make a self-loop, allowing a state to have arbitrary duration). Figure 2.8 illustrates how states and transitions in an HMM can be structured hierarchically, in order to represent phonemes, words, and sentences.

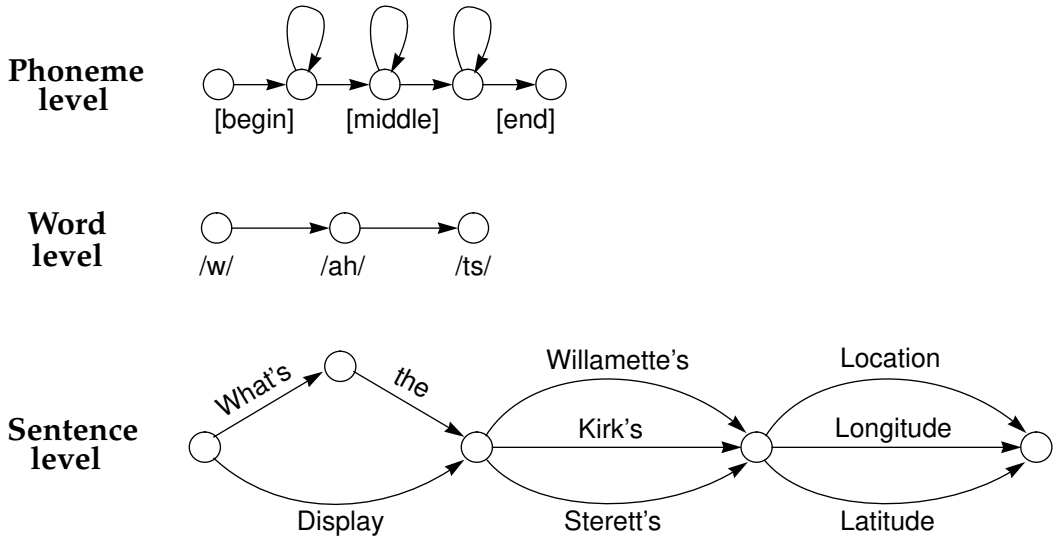

Figure 2.8: A hierarchically structured HMM.

Formally, an HMM consists of the following elements:

$\{s\}$ = A set of states.

$\{a_{ij}\}$ = A set of transition probabilities, where a_{ij} is the probability of taking the transition from state i to state j.

$\{b_i(u)\}$ = A set of emission probabilities, where b_i is the probability distribution over the acoustic space describing the likelihood of emitting[1] each possible sound u while in state i.

Since a and b are both probabilities, they must satisfy the following properties:

$$a_{ij} \geq 0, \quad b_i(u) \geq 0, \quad \forall i,j,u \tag{2}$$

$$\sum_j a_{ij} = 1, \quad \forall i \tag{3}$$

$$\sum_u b_i(u) = 1, \quad \forall i \tag{4}$$

In using this notation we implicitly confine our attention to First-Order HMMs, in which $\boldsymbol{a}$ and $\boldsymbol{b}$ depend only on the current state, independent of the previous history of the state sequence. This assumption, almost universally observed, limits the number of trainable parameters and makes the training and testing algorithms very efficient, rendering HMMs useful for speech recognition.

2.3.2. Algorithms

There are three basic algorithms associated with Hidden Markov Models:

- the *forward algorithm*, useful for isolated word recognition;
- the *Viterbi algorithm*, useful for continuous speech recognition; and
- the *forward-backward algorithm*, useful for training an HMM.

In this section we will review each of these algorithms.

2.3.2.1. The Forward Algorithm

In order to perform isolated word recognition, we must be able to evaluate the probability that a given HMM word model produced a given observation sequence, so that we can compare the scores for each word model and choose the one with the highest score. More formally: given an HMM model $\boldsymbol{M}$, consisting of $\{s\}$, $\{a_{ij}\}$, and $\{b_i(u)\}$, we must compute the probability that it generated the output sequence $y_1^T = (y_1, y_2, y_3, ..., y_T)$. Because every state i can generate each output symbol u with probability $b_i(u)$, every state sequence of length T

1. It is traditional to refer to $b_i(u)$ as an "emission" probability rather than an "observation" probability, because an HMM is traditionally a generative model, even though we are using it for speech recognition. The difference is moot.

contributes something to the total probability. A brute force algorithm would simply list all possible state sequences of length T, and accumulate their probabilities of generating y_1^T; but this is clearly an exponential algorithm, and is not practical.

A much more efficient solution is the *Forward Algorithm*, which is an instance of the class of algorithms known as *dynamic programming*, requiring computation and storage that are only linear in T. First, we define $\alpha_j(t)$ as the probability of generating the partial sequence y_1^t, ending up in state j at time t. $\alpha_j(t=0)$ is initialized to 1.0 in the initial state, and 0.0 in all other states. If we have already computed $\alpha_i(t-1)$ for all i in the previous time frame $t-1$, then $\alpha_j(t)$ can be computed recursively in terms of the incremental probability of entering state j from each i while generating the output symbol y_t (see Figure 2.9):

$$\alpha_j(t) = \sum_i \alpha_i(t-1)\, a_{ij} b_j(y_t) \tag{5}$$

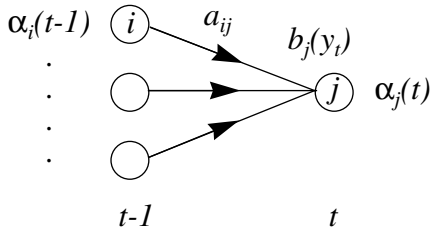

Figure 2.9: The forward pass recursion.

If F is the final state, then by induction we see that $\alpha_F(T)$ is the probability that the HMM generated the complete output sequence y_1^T.

Figure 2.10 shows an example of this algorithm in operation, computing the probability that the output sequence y_1^3 = (A, A, B) could have been generated by the simple HMM presented earlier. Each cell at (t,j) shows the value of $\alpha_j(t)$, using the given values of $\boldsymbol{a}$ and $\boldsymbol{b}$. The computation proceeds from the first state to the last state within a time frame, before proceeding to the next time frame. In the final cell, we see that the probability that this particular HMM generates the sequence (A, A, B) is .096.

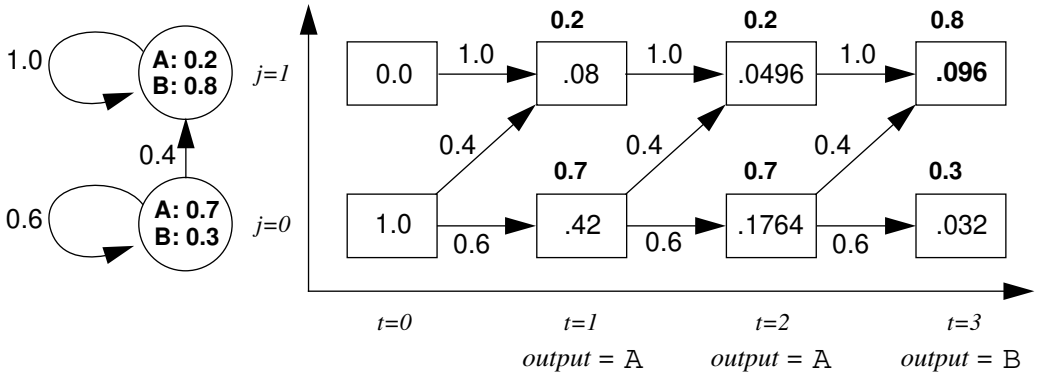

Figure 2.10: An illustration of the forward algorithm, showing the value of $\alpha_j(t)$ in each cell.

2.3.2.2. The Viterbi Algorithm

While the Forward Algorithm is useful for isolated word recognition, it cannot be applied to continuous speech recognition, because it is impractical to have a separate HMM for each possible sentence. In order to perform continuous speech recognition, we should instead infer the actual sequence of states that generated the given observation sequence; from the state sequence we can easily recover the word sequence. Unfortunately the actual state sequence is hidden (by definition), and cannot be uniquely identified; after all, any path could have produced this output sequence, with some small probability. The best we can do is to find the *one* state sequence that was *most likely* to have generated the observation sequence. As before, we could do this by evaluating all possible state sequences and reporting the one with the highest probability, but this would be an exponential and hence infeasible algorithm.

A much more efficient solution is the *Viterbi Algorithm*, which is again based on dynamic programming. It is very similar to the Forward Algorithm, the main difference being that instead of evaluating a summation at each cell, we evaluate the maximum:

$$v_j(t) = MAX_i\left[v_i(t-1)\, a_{ij} b_j(y_t) \right] \tag{6}$$

This implicitly identifies the single best predecessor state for each cell in the matrix. If we explicitly identify that best predecessor state, saving a single backpointer in each cell in the matrix, then by the time we have evaluated $v_F(T)$ at the final state at the final time frame, we can retrace those backpointers from the final cell to reconstruct the whole state sequence. Figure 2.11 illustrates this process. Once we have the state sequence (i.e., an alignment path), we can trivially recover the word sequence.

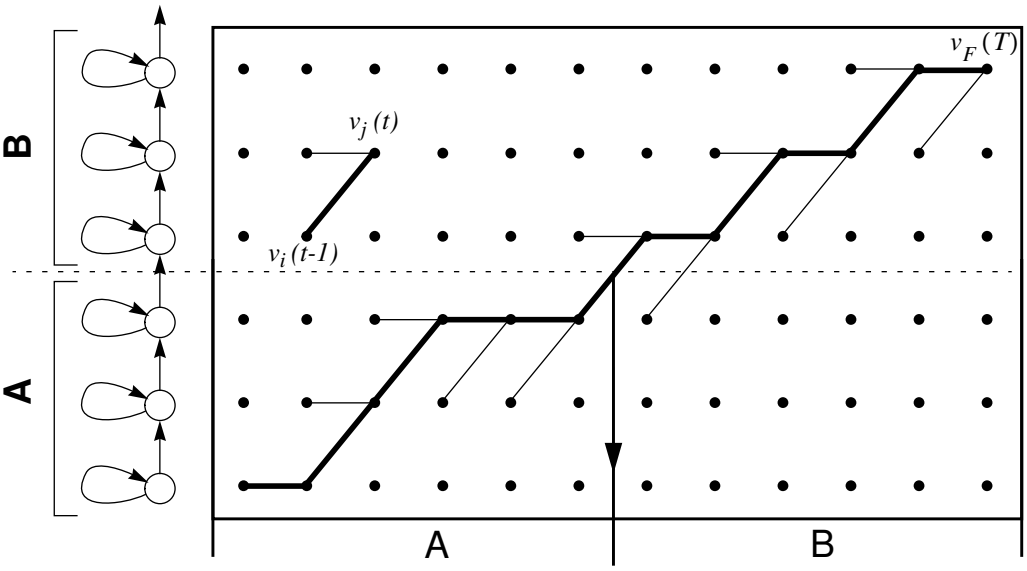

Figure 2.11: An example of backtracing.

2.3.2.3. The Forward-Backward Algorithm

In order to train an HMM, we must optimize ***a*** and ***b*** with respect to the HMM's likelihood of generating all of the output sequences in the training set, because this will maximize the HMM's chances of also correctly recognizing new data. Unfortunately this is a difficult problem; it has no closed form solution. The best that can be done is to start with some initial values for ***a*** and ***b***, and then to iteratively modify ***a*** and ***b*** by reestimating and improving them, until some stopping criterion is reached. This general method is called *Estimation-Maximization* (EM). A popular instance of this general method is the *Forward-Backward Algorithm* (also known as the *Baum-Welch Algorithm*), which we now describe.

Previously we defined $\alpha_j(t)$ as the probability of generating the partial sequence y_1^t and ending up in state j at time t. Now we define its mirror image, $\beta_j(t)$, as the probability of generating the remainder of the sequence y_{t+1}^T, starting from state j at time t. $\alpha_j(t)$ is called the *forward term*, while $\beta_j(t)$ is called the *backward term*. Like $\alpha_j(t)$, $\beta_j(t)$ can be computed recursively, but this time in a backward direction (see Figure 2.12):

$$\beta_j(t) = \sum_k a_{jk} b_k(y_{t+1}) \beta_k(t+1) \tag{7}$$

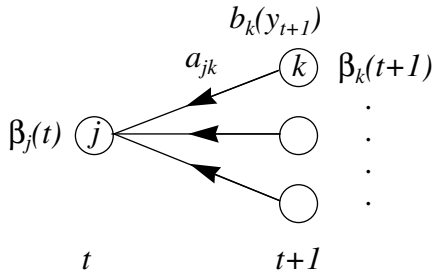

Figure 2.12: The backward pass recursion.

This recursion is initialized at time T by setting $\beta_k(T)$ to 1.0 for the final state, and 0.0 for all other states.

Now we define $\gamma_{ij}(t)$ as the probability of transitioning from state i to state j at time t, given that the whole output sequence y_1^T has been generated by the current HMM:

$$\gamma_{ij}(t) = P(i_t \to j | y_1^T) = \frac{P(i_t \to j, y_1^T)}{P(y_1^T)} = \frac{\alpha_i(t)\, a_{ij} b_j(y_{t+1}) \beta_j(t+1)}{\sum_k \alpha_k(T)} \tag{8}$$

The numerator in the final equality can be understood by consulting Figure 2.13. The denominator reflects the fact that the probability of generating y_1^T equals the probability of generating y_1^T while ending up in any of k final states.

Now let us define $N(i \to j)$ as the expected number of times that the transition from state i to state j is taken, from time 1 to T:

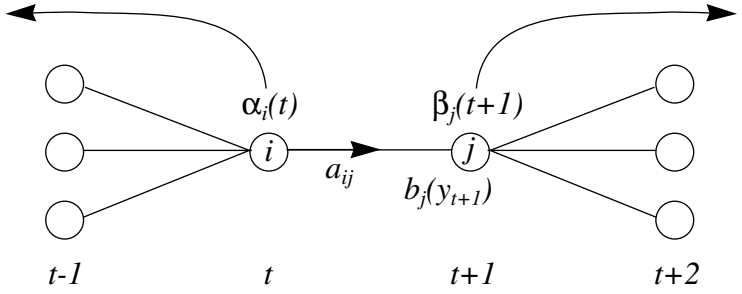

Figure 2.13: Deriving $\gamma_{ij}(t)$ in the Forward-Backward Algorithm.

$$N(i \rightarrow j) = \sum_t \gamma_{ij}(t) \tag{9}$$

Summing this over all destination states j, we obtain $N(i \rightarrow *)$, or $N(i)$, which represents the expected number of times that state i is visited, from time 1 to T:

$$N(i) = N(i \rightarrow *) = \sum_j \sum_t \gamma_{ij}(t) \tag{10}$$

Selecting only those occasions when state i emits the symbol u, we obtain $N(i, u)$:

$$N(i, u) = \sum_{t:(y_t=u)} \sum_j \gamma_{ij}(t) \tag{11}$$

Finally, we can reestimate the HMM parameters $\boldsymbol{a}$ and $\boldsymbol{b}$, yielding $\overline{\boldsymbol{a}}$ and $\overline{\boldsymbol{b}}$, by taking simple ratios between these terms:

$$\overline{a_{ij}} = P(i \rightarrow j) = \frac{N(i \rightarrow j)}{N(i \rightarrow *)} = \frac{\sum_t \gamma_{ij}(t)}{\sum_j \sum_t \gamma_{ij}(t)} \tag{12}$$

$$\overline{b_i(u)} = P(i, u) = \frac{N(i, u)}{N(i)} = \frac{\sum_{t:(y_t=u)} \sum_j \gamma_{ij}(t)}{\sum_t \sum_j \gamma_{ij}(t)} \tag{13}$$

It can be proven that substituting $\{\overline{\boldsymbol{a}}, \overline{\boldsymbol{b}}\}$ for $\{\boldsymbol{a}, \boldsymbol{b}\}$ will always cause $P(y_1^T)$ to increase, up to a local maximum. Thus, by repeating this procedure for a number of iterations, the HMM parameters will be optimized for the training data, and will hopefully generalize well to testing data.

2.3.3. Variations

There are many variations on the standard HMM model. In this section we discuss some of the more important variations.

2.3.3.1. Density Models

The states of an HMM need some way to model probability distributions in acoustic space. There are three popular ways to do this, as illustrated in Figure 2.14:

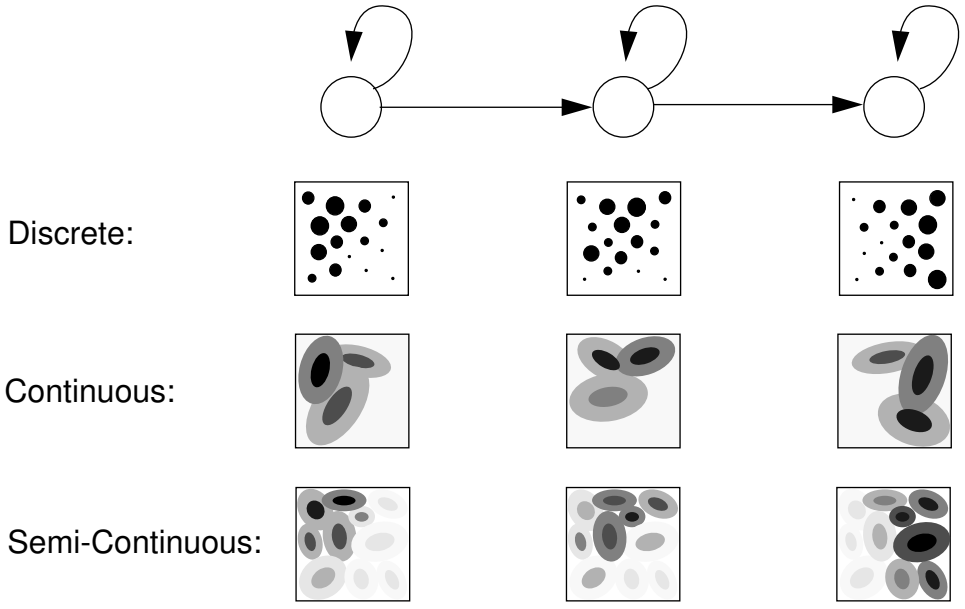

Figure 2.14: Density models, describing the probability density in acoustic space.

- **Discrete density model** (Lee 1988). In this approach, the entire acoustic space is divided into a moderate number (e.g., 256) of regions, by a clustering procedure known as Vector Quantization (VQ). The centroid of each cluster is represented by a scalar codeword, which is an index into a codebook that identifies the corresponding acoustic vectors. Each input frame is converted to a codeword by finding the nearest vector in the codebook. The HMM output symbols are also codewords. Thus, the probability distribution over acoustic space is represented by a simple histogram over the codebook entries. The drawback of this nonparametric approach is that it suffers from quantization errors if the codebook is too small, while increasing the codebook size would leave less training data for each codeword, likewise degrading performance.
- **Continuous density model** (Woodland et al, 1994). Quantization errors can be eliminated by using a continuous density model, instead of VQ codebooks. In this approach, the probability distribution over acoustic space is modeled directly, by assuming that it has a certain parametric form, and then trying to find those param-

eters. Typically this parametric form is taken to be a mixture of K Gaussians, i.e.,

$$b_j(y) = \sum_{k=1}^{K} c_{jk} G(y, \mu_{jk}, U_{jk}) \tag{14}$$

where c_{jk} is the weighting factor for each Gaussian G with mean μ_{jk} and covariance matrix U_{jk}, such that $\sum_k c_{jk} = 1$. During training, the reestimation of $\boldsymbol{b}$ then involves the reestimation of c_{jk}, μ_{jk}, and U_{jk}, using an additional set of formulas. The drawback of this approach is that parameters are not shared between states, so if there are many states in the whole system, then a large value of K may yield too many total parameters to be trained adequately, while decreasing the value of K may invalidate the assumption that the distribution can be well-modeled by a mixture of Gaussians.

- **Semi-Continuous density model** (Huang 1992), also called the **Tied-Mixture model** (Bellagarda and Nahamoo 1988). This is a compromise between the above two approaches. In a Semi-Continuous density model, as in the discrete model, there is a codebook describing acoustic clusters, shared by all states. But rather than representing the clusters as discrete centroids to which nearby vectors are collapsed, they are represented as continuous density functions (typically Gaussians) over the neighboring space, thus avoiding quantization errors. That is,

$$b_j(y) = \sum_{k=1}^{L} c_{jk} G(y, \mu_k, U_k) \tag{15}$$

where L is the number of codebook entries, and c_{jk} is the weighting factor for each Gaussian G with mean μ_k and covariance matrix U_k. As in the continuous case, the Gaussians are reestimated during training, hence the codebook is optimized jointly with the HMM parameters, in contrast to the discrete model in which the codebook remains fixed. This joint optimization can further improve the system's performance.

All three density models are widely used, although continuous densities seem to give the best results on large databases (while running up to 300 times slower, however).

2.3.3.2. Multiple Data Streams

So far we have discussed HMMs that assume a single data stream, i.e., input acoustic vectors. HMMs can be modified to use multiple streams, such that

$$b_j(u) = \prod_{i=1}^{N} b_j(u_i) \tag{16}$$

where u_i are the observation vectors of N independent data streams, which are modeled with separate codebooks or Gaussian mixtures. HMM based speech recognizers commonly[1] use up to four data streams, for example representing spectral coefficients, delta spectral coeffi-

cients, power, and delta power. While it is possible to concatenate each of these into one long vector, and to vector-quantize that single data stream, it is generally better to treat these separate data streams independently, so that each stream is more coherent and their union can be modeled with a minimum of parameters.

2.3.3.3. Duration modeling

If the self-transition probability $a_{ii} = p$, then the probability of remaining in state i for d frames is p^d, indicating that state duration in an HMM is modeled by exponential decay. Unfortunately this is a poor model of duration, as state durations actually have a roughly Poisson distribution. There are several ways to improve duration modeling in HMMs.

We can define $p_i(d)$ as the probability of remaining in state i for a duration of d frames, and create a histogram of $p_i(d)$ from the training data. To ensure that state duration is governed by $p_i(d)$, we must eliminate all self-loops (by setting a_{ii}=0), and modify the equations for α and β as well as all the reestimation formulas, to include summations over d (up to a maximum duration D) of terms with multiplicative factors that represent all possible durational contingencies. Unfortunately this increases memory requirements by a factor of D, and computational requirements by a factor of $D^2/2$. If D=25 frames (which is quite reasonable), this causes the application to run about 300 times slower. Another problem with this approach is that it may require more training parameters (adding about 25 per state) than can be adequately trained.

The latter problem can be mitigated by replacing the above nonparametric approach with a parametric approach, in which a Poisson, Gaussian, or Gamma distribution is assumed as a duration model, so that relatively few parameters are needed. However, this improvement causes the system to run even slower.

A third possibility is to ignore the precise shape of the distribution, and simply impose hard minimum and maximum duration constraints. One way to impose these constraints is by duplicating the states and modifying the state transitions appropriately. This approach has only moderate overhead, and gives fairly good results, so it tends to be the most favored approach to duration modeling.

2.3.3.4. Optimization criteria

The training procedure described earlier (the Forward-Backward Algorithm) implicitly uses an optimization criterion known as *Maximum Likelihood* (ML), which maximizes the likelihood that a given observation sequence Y is generated by the correct model M_c, without considering other models M_i. (For instance, if M_i represent word models, then only the correct word model will be updated with respect to Y, while all the competing word models are ignored.) Mathematically, ML training solves for the HMM parameters $\Lambda = \{\boldsymbol{a}, \boldsymbol{b}\}$, and specifically the subset Λ_c that corresponds to the correct model M_c, such that

$$\Lambda_{ML} = \underset{\Lambda}{\operatorname{argmax}}\ P(Y|\Lambda_c) \tag{17}$$

1. Although this is still common among semi-continuous HMMs, there is now a trend towards using a single data stream with LDA coefficients derived from these separate streams; this latter approach is now common among continuous HMMs.

If the HMM's modeling assumptions were accurate — e.g., if the probability density in acoustic space could be precisely modeled by a mixture of Gaussians, and if enough training data were available for perfectly estimating the distributions — then ML training would theoretically yield optimal recognition accuracy. But the modeling assumptions are always inaccurate, because acoustic space has a complex terrain, training data is limited, and the scarcity of training data limits the size and power of the models, so that they cannot perfectly fit the distributions. This unfortunate condition is called *model mismatch*. An important consequence is that ML is not guaranteed to be the optimal criterion for training an HMM.

An alternative criterion is *Maximum Mutual Information* (MMI), which enhances discrimination between competing models, in an attempt to squeeze as much useful information as possible out of the limited training data. In this approach, the correct model M_c is trained positively while all other models M_i are trained negatively on the observation sequence Y, helping to separate the models and improve their ability to discriminate during testing. Mutual information between an observation sequence Y and the correct model M_c is defined as follows:

$$\begin{aligned} I_\Lambda(Y, M_c) &= \log \frac{P(Y, M_c)}{P(Y)\,P(M_c)} = \log \frac{P(Y|M_c)}{P(Y)} = \log P(Y|M_c) - \log P(Y) \\ &= \log P(Y|M_c) - \log \sum_i P(Y|M_i)\,P(M_i) \end{aligned} \tag{18}$$

where the first term represents positive training on the correct model M_c (just as in ML), while the second term represents negative training on all other models M_i. Training with the MMI criterion then involves solving for the model parameters Λ that maximize the mutual information:

$$\Lambda_{MMI} = \operatorname*{argmax}_\Lambda I_\Lambda(Y, M_c) \tag{19}$$

Unfortunately, this equation cannot be solved by either direct analysis or reestimation; the only known way to solve it is by gradient descent, and the proper implementation is complex (Brown 1987, Rabiner 1989).

We note in passing that MMI is equivalent to using a *Maximum A Posteriori* (MAP) criterion, in which the expression to be maximized is $P(M_c|Y)$, rather than $P(Y|M_c)$. To see this, note that according to Bayes Rule,

$$P(M_c|Y) = \frac{P(Y|M_c)\,P(M_c)}{P(Y)} \tag{20}$$

Maximizing this expression is equivalent to maximizing $I_\Lambda(Y, M_c)$, because the distinguishing logarithm is monotonic and hence transparent, and the MAP's extra factor of $P(M_c)$ is transparent because it's only an additive constant (after taking logarithms), whose value is fixed by the HMM's topology and language model.

2.3.4. Limitations of HMMs

Despite their state-of-the-art performance, HMMs are handicapped by several well-known weaknesses, namely:

- The First-Order Assumption — which says that all probabilities depend solely on the current state — is false for speech applications. One consequence is that HMMs have difficulty modeling coarticulation, because acoustic distributions are in fact strongly affected by recent state history. Another consequence is that durations are modeled inaccurately by an exponentially decaying distribution, rather than by a more accurate Poisson or other bell-shaped distribution.
- The Independence Assumption — which says that there is no correlation between adjacent input frames — is also false for speech applications. In accordance with this assumption, HMMs examine only one frame of speech at a time. In order to benefit from the context of neighboring frames, HMMs must absorb those frames into the current frame (e.g., by introducing multiple streams of data in order to exploit delta coefficients, or using LDA to transform these streams into a single stream).
- The HMM probability density models (discrete, continuous, and semi-continuous) have suboptimal modeling accuracy. Specifically, discrete density HMMs suffer from quantization errors, while continuous or semi-continuous density HMMs suffer from model mismatch, i.e., a poor match between their a priori choice of statistical model (e.g., a mixture of K Gaussians) and the true density of acoustic space.
- The Maximum Likelihood training criterion leads to poor discrimination between the acoustic models (given limited training data and correspondingly limited models). Discrimination can be improved using the Maximum Mutual Information training criterion, but this is more complex and difficult to implement properly.

Because HMMs suffer from all these weaknesses, they can obtain good performance only by relying on context dependent phone models, which have so many parameters that they must be extensively shared — and this, in turn, calls for elaborate mechanisms such as senones and decision trees (Hwang et al, 1993b).

We will argue that neural networks mitigate each of the above weaknesses (except the First Order Assumption), while they require relatively few parameters, so that a neural network based speech recognition system can get equivalent or better performance with less complexity.

3. Review of Neural Networks

In this chapter we present a brief review of neural networks. After giving some historical background, we will review some fundamental concepts, describe different types of neural networks and training procedures (with special emphasis on backpropagation), and discuss the relationship between neural networks and conventional statistical techniques.

3.1. Historical Development

The modern study of neural networks actually began in the 19th century, when neurobiologists first began extensive studies of the human nervous system. Cajal (1892) determined that the nervous system is comprised of discrete neurons, which communicate with each other by sending electrical signals down their long *axons*, which ultimately branch out and touch the *dendrites* (receptive areas) of thousands of other neurons, transmitting the electrical signals through *synapses* (points of contact, with variable resistance). This basic picture was elaborated on in the following decades, as different kinds of neurons were identified, their electrical responses were analyzed, and their patterns of connectivity and the brain's gross functional areas were mapped out. While neurobiologists found it relatively easy to study the functionality of individual neurons (and to map out the brain's gross functional areas), it was extremely difficult to determine how neurons worked together to achieve high-level functionality, such as perception and cognition. With the advent of high-speed computers, however, it finally became possible to build working models of neural systems, allowing researchers to freely experiment with such systems and better understand their properties.

McCulloch and Pitts (1943) proposed the first computational model of a neuron, namely the *binary threshold unit*, whose output was either 0 or 1 depending on whether its net input exceeded a given threshold. This model caused a great deal of excitement, for it was shown that a system of such neurons, assembled into a finite state automaton, could compute any arbitrary function, given suitable values of weights between the neurons (see Minsky 1967). Researchers soon began searching for *learning procedures* that would automatically find the values of weights enabling such a network to compute any specific function. Rosenblatt (1962) discovered an iterative learning procedure for a particular type of network, the *single-layer perceptron*, and he proved that this learning procedure always converged to a set of weights that produced the desired function, as long as the desired function was potentially computable by the network. This discovery caused another great wave of excitement, as many AI researchers imagined that the goal of machine intelligence was within reach.

However, in a rigorous analysis, Minsky and Papert (1969) showed that the set of functions potentially computable by a single-layer perceptron is actually quite limited, and they expressed pessimism about the potential of multi-layer perceptrons as well; as a direct result, funding for connectionist research suddenly dried up, and the field lay dormant for 15 years.

Interest in neural networks was gradually revived when Hopfield (1982) suggested that a network can be analyzed in terms of an *energy function*, triggering the development of the *Boltzmann Machine* (Ackley, Hinton, & Sejnowski 1985) — a stochastic network that could be trained to produce any kind of desired behavior, from arbitrary pattern mapping to pattern completion. Soon thereafter, Rumelhart et al (1986) popularized a much faster learning procedure called *backpropagation*, which could train a multi-layer perceptron to compute any desired function, showing that Minsky and Papert's earlier pessimism was unfounded. With the advent of backpropagation, neural networks have enjoyed a third wave of popularity, and have now found many useful applications.

3.2. Fundamentals of Neural Networks

In this section we will briefly review the fundamentals of neural networks. There are many different types of neural networks, but they all have four basic attributes:

- A set of processing units;
- A set of connections;
- A computing procedure;
- A training procedure.

Let us now discuss each of these attributes.

3.2.1. Processing Units

A neural network contains a potentially huge number of very simple processing units, roughly analogous to neurons in the brain. All these units operate simultaneously, supporting massive parallelism. All computation in the system is performed by these units; there is no other processor that oversees or coordinates their activity[1]. At each moment in time, each unit simply computes a scalar function of its local inputs, and broadcasts the result (called the *activation value*) to its neighboring units.

The units in a network are typically divided into *input units*, which receive data from the environment (such as raw sensory information); *hidden units*, which may internally transform the data representation; and/or *output units*, which represent decisions or control signals (which may control motor responses, for example).

1. Except, of course, to the extent that the neural network may be simulated on a conventional computer, rather than implemented directly in hardware.

In drawings of neural networks, units are usually represented by circles. Also, by convention, input units are usually shown at the bottom, while the outputs are shown at the top, so that processing is seen to be "bottom-up".

The *state* of the network at each moment is represented by the set of activation values over all the units; the network's state typically varies from moment to moment, as the inputs are changed, and/or feedback in the system causes the network to follow a dynamic trajectory through state space.

3.2.2. Connections

The units in a network are organized into a given topology by a set of *connections*, or *weights*, shown as lines in a diagram. Each weight has a real value, typically ranging from $-\infty$ to $+\infty$, although sometimes the range is limited. The value (or *strength*) of a weight describes how much influence a unit has on its neighbor; a positive weight causes one unit to excite another, while a negative weight causes one unit to inhibit another. Weights are usually one-directional (from input units towards output units), but they may be two-directional (especially when there is no distinction between input and output units).

The values of all the weights predetermine the network's computational reaction to any arbitrary input pattern; thus the weights encode the *long-term memory*, or the *knowledge*, of the network. Weights can change as a result of training, but they tend to change slowly, because accumulated knowledge changes slowly. This is in contrast to activation patterns, which are transient functions of the current input, and so are a kind of *short-term memory.*

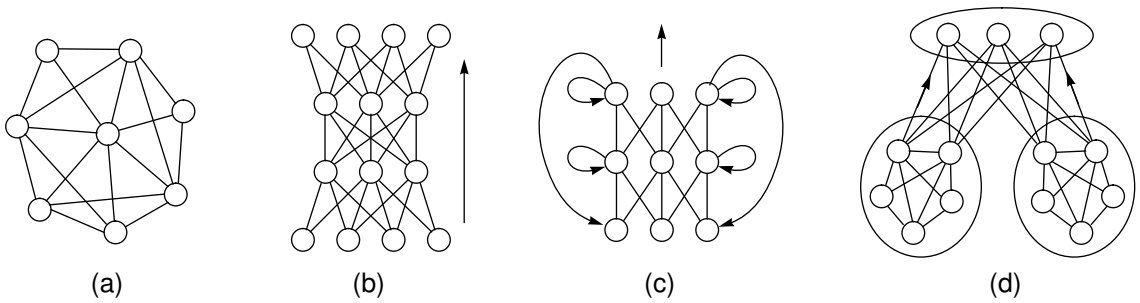

Figure 3.1: Neural network topologies: (a) unstructured, (b) layered, (c) recurrent, (d) modular.

A network can be connected with any kind of topology. Common topologies include unstructured, layered, recurrent, and modular networks, as shown in Figure 3.1. Each kind of topology is best suited to a particular type of application. For example:

- *unstructured networks* are most useful for pattern completion (i.e., retrieving stored patterns by supplying any part of the pattern);
- *layered networks* are useful for pattern association (i.e., mapping input vectors to output vectors);
- *recurrent networks* are useful for pattern sequencing (i.e., following sequences of

network activation over time); and

- *modular networks* are useful for building complex systems from simpler components.

Note that unstructured networks may contain cycles, and hence are actually recurrent; layered networks may or may not be recurrent; and modular networks may integrate different kinds of topologies. In general, unstructured networks use 2-way connections, while other networks use 1-way connections.

Connectivity between two groups of units, such as two layers, is often *complete* (connecting all to all), but it may also be *random* (connecting only some to some), or *local* (connecting one neighborhood to another). A completely connected network has the most degrees of freedom, so it can theoretically learn more functions than more constrained networks; however, this is not always desirable. If a network has too many degrees of freedom, it may simply memorize the training set without learning the underlying structure of the problem, and consequently it may generalize poorly to new data. Limiting the connectivity may help constrain the network to find economical solutions, and so to generalize better. Local connectivity, in particular, can be very helpful when it reflects topological constraints inherent in a problem, such as the geometric constraints that are present between layers in a visual processing system.

3.2.3. Computation

Computation always begins by presenting an input pattern to the network, or *clamping* a pattern of activation on the input units. Then the activations of all of the remaining units are computed, either *synchronously* (all at once in a parallel system) or *asynchronously* (one at a time, in either randomized or natural order), as the case may be. In unstructured networks, this process is called *spreading activation*; in layered networks, it is called *forward propagation*, as it progresses from the input layer to the output layer. In feedforward networks (i.e., networks without feedback), the activations will stabilize as soon as the computations reach the output layer; but in recurrent networks (i.e., networks with feedback), the activations may never stabilize, but may instead follow a dynamic trajectory through state space, as units are continuously updated.

A given unit is typically updated in two stages: first we compute the unit's *net input* (or internal activation), and then we compute its *output activation* as a function of the net input. In the standard case, as shown in Figure 3.2(a), the net input x_j for unit j is just the weighted sum of its inputs:

$$x_j = \sum_i y_i w_{ji} \tag{21}$$

where y_i is the output activation of an incoming unit, and w_{ji} is the weight from unit i to unit j. Certain networks, however, will support so-called *sigma-pi* connections, as shown in Figure 3.2(b), where activations are multiplied together (allowing them to gate each other) before being weighted. In this case, the net input is given by:

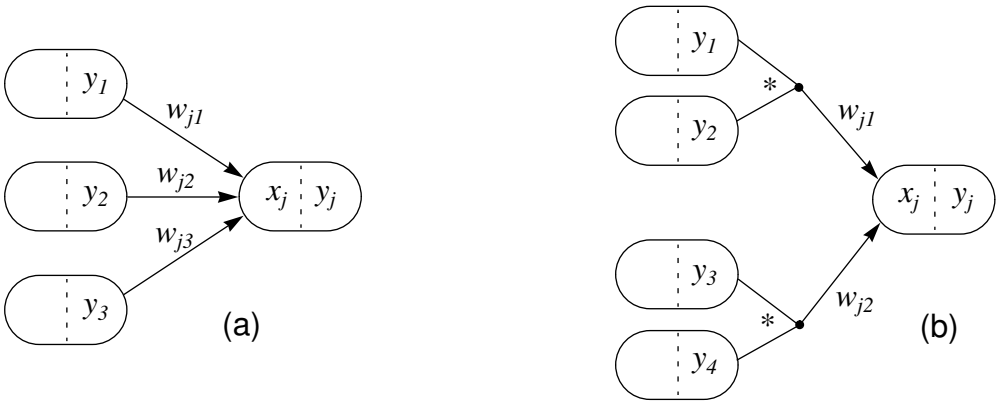

Figure 3.2: Computing unit activations: *x*=net input, *y*=activation. (a) standard unit; (b) sigma-pi unit.

$$x_j = \sum_i w_{ji} \prod_{k \in k(i)} y_k \tag{22}$$

from which the name "sigma-pi" is transparently derived.

In general, the net input is offset by a variable bias term, θ, so that for example Equation (21) is actually:

$$x_j = \sum_i y_i w_{ji} + \theta_j \tag{23}$$

However, in practice, this bias is usually treated as another weight w_{j0} connected to an invisible unit with activation $y_0 = 1$, so that the bias is automatically included in Equation (21) if the summation's range includes this invisible unit.

Once we have computed the unit's net input x_j, we compute the output activation y_j as a function of x_j. This *activation function* (also called a *transfer function*) can be either deterministic or stochastic, and either local or nonlocal.

Deterministic local activation functions usually take one of three forms — *linear, threshold,* or *sigmoidal* — as shown in Figure 3.3. In the linear case, we have simply $y = x$. This is not used very often because it's not very powerful: multiple layers of linear units can be collapsed into a single layer with the same functionality. In order to construct nonlinear functions, a network requires nonlinear units. The simplest form of nonlinearity is provided by the threshold activation function, illustrated in panel (b):

$$y = \begin{cases} 0 & \text{if } x \le 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{24}$$

This is much more powerful than a linear function, as a multilayered network of threshold units can theoretically compute any boolean function. However, it is difficult to train such a network because the discontinuities in the function imply that finding the desired set of weights may require an exponential search; a practical learning rule exists only for single-

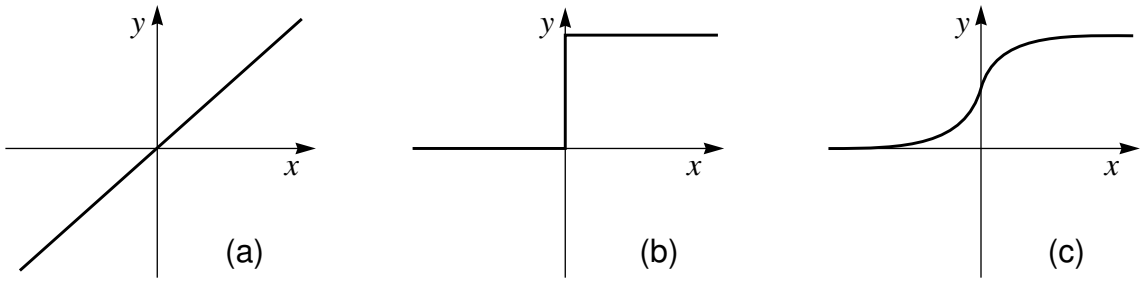

Figure 3.3: Deterministic local activation functions: (a) linear; (b) threshold; (c) sigmoidal.

layered networks of such units, which have limited functionality. Moreover, there are many applications where continuous outputs are preferable to binary outputs. Consequently, the most common function is now the sigmoidal function, illustrated in panel (c):

$$y = \frac{1}{1 + \exp(-x)} \qquad \text{or similarly} \qquad y = \tanh(x) \tag{25}$$

Sigmoidal functions have the advantages of nonlinearity, continuousness, and differentiability, enabling a multilayered network to compute any arbitrary real-valued function, while also supporting a practical training algorithm, backpropagation, based on gradient descent.

Nonlocal activation functions can be useful for imposing global constraints on the network. For example, sometimes it is useful to force all of the network's output activations to sum to 1, like probabilities. This can be performed by linearly normalizing the outputs, but a more popular approach is to use the *softmax* function:

$$y_j = \frac{\exp(x_j)}{\sum_i \exp(x_i)} \tag{26}$$

which operates on the net inputs directly. Nonlocal functions require more overhead and/or hardware, and so are biologically implausible, but they can be useful when global constraints are desired.

Nondeterministic activation functions, in contrast to deterministic ones, are probabilistic in nature. They typically produce binary activation values (0 or 1), where the probability of outputing a 1 is given by:

$$P(y = 1) = \frac{1}{1 + \exp(-x/T)} \tag{27}$$

Here T is a variable called the *temperature*, which commonly varies with time. Figure 3.4 shows how this probability function varies with the temperature: at infinite temperature we have a uniform probability function; at finite temperatures we have sigmoidal probability functions; and at zero temperature we have a binary threshold probability function. If the temperature is steadily decreased during training, in a process called *simulated annealing*, a

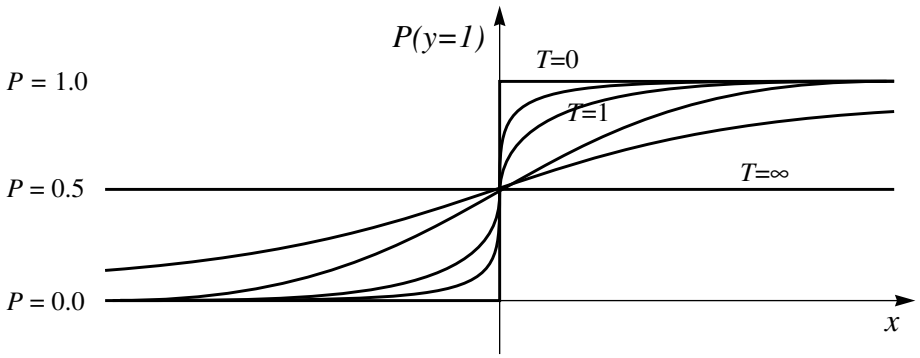

Figure 3.4: Nondeterministic activation functions: Probability of outputing 1 at various temperatures.

network may be able to escape local minima (which can trap deterministic gradient descent procedures like backpropagation), and find global minima instead.

Up to this point we have discussed units whose activation functions have the general form

$$y_j = f(x_j) \qquad \text{where} \qquad x_j = \sum_i y_i w_{ji} \tag{28}$$

This is the most common form of activation function. However, some types of networks — such as Learned Vector Quantization (LVQ) networks, and Radial Basis Function (RBF) networks — include units that are based on another type of activation function, with the general form:

$$y_j = f(x_j) \qquad \text{where} \qquad x_j = \sum_i (y_i - w_{ji})^2 \tag{29}$$

The difference between these two types of units has an intuitive geometric interpretation, illustrated in Figure 3.5. In the first case, x_j is the dot product between an input vector $\boldsymbol{y}$ and a weight vector $\boldsymbol{w}$, so x_j is the length of the projection of $\boldsymbol{y}$ onto $\boldsymbol{w}$, as shown in panel (a). This projection may point either in the same or the opposite direction as $\boldsymbol{w}$, i.e., it may lie either on one side or the other of a hyperplane that is perpendicular to $\boldsymbol{w}$. Inputs that lie on the same side will have $x_j > 0$, while inputs that lie on the opposite side will have $x_j < 0$. Thus, if $y_j = f(x_j)$ is a threshold function, as in Equation (24), then the unit will classify each input in terms of which side of the hyperplane it lies on. (This classification will be fuzzy if a sigmoidal function is used instead of a threshold function.)

By contrast, in the second case, x_j is the Euclidean distance between an input vector $\boldsymbol{y}$ and a weight vector $\boldsymbol{w}$. Thus, the weight represents the center of a spherical distribution in input space, as shown in panel (b). The distance function can be inverted by a function like $y_j = f(x_j) = \exp(-x_j)$, so that an input at the center of the cluster has an activation $y_j = 1$, while an input at an infinite distance has an activation $y_j = 0$.

In either case, such decision regions — defined by hyperplanes or hyperspheres, with either discontinuous or continuous boundaries — can be positioned anywhere in the input space, and used to "carve up" the input space in arbitrary ways. Moreover, a set of such

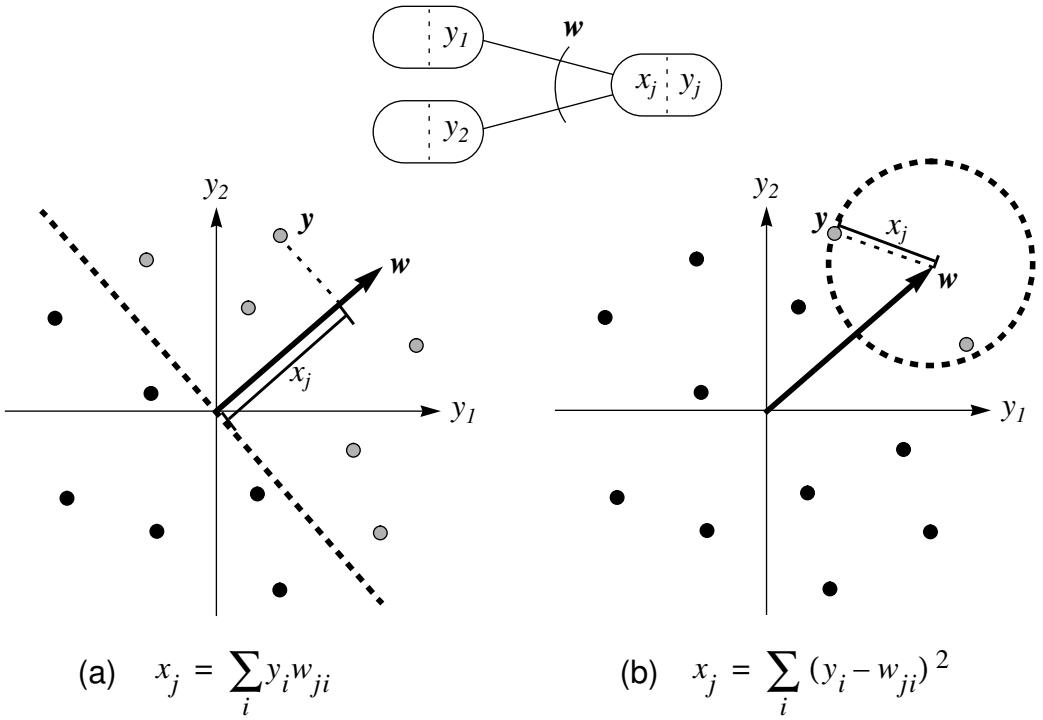

Figure 3.5: Computation of net input. (a) Dot product ⇒ hyperplane; (b) Difference ⇒ hypersphere.

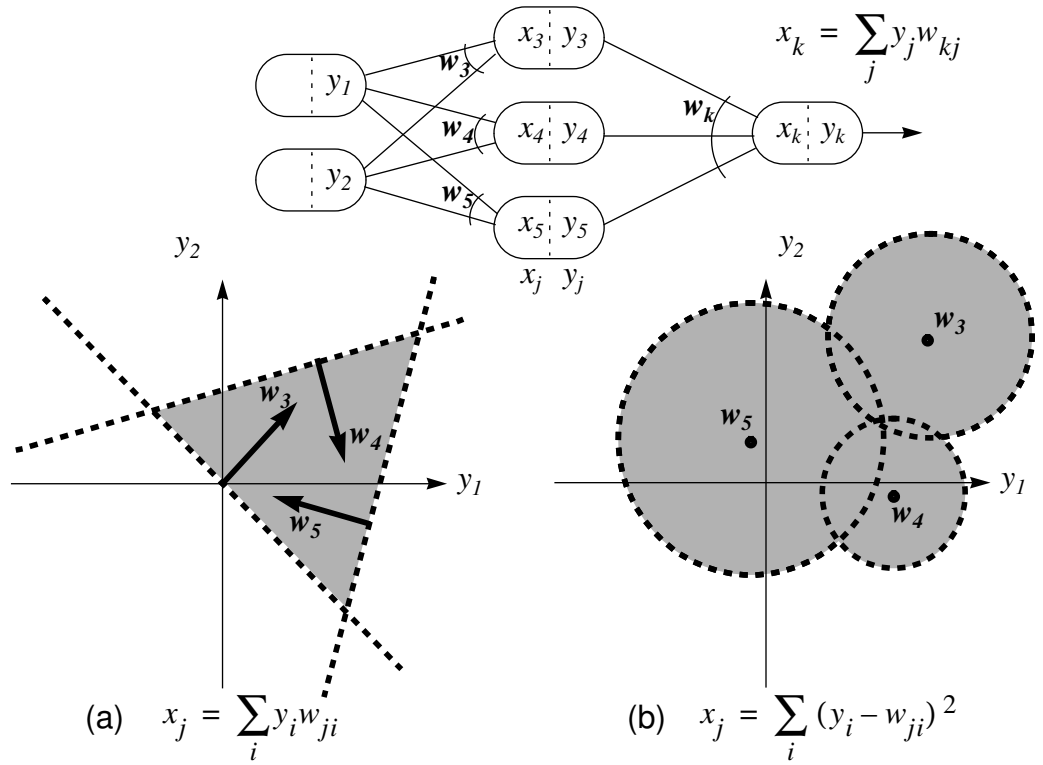

Figure 3.6: Construction of complex functions from (a) hyperplanes, or (b) hyperspheres.

decision regions can be overlapped and combined, to construct any arbitrarily complex function, by including at least one additional layer of threshold (or sigmoidal) units, as illustrated in Figure 3.6. It is the task of a training procedure to adjust the hyperplanes and/or hyperspheres to form a more accurate model of the desired function.

3.2.4. Training

Training a network, in the most general sense, means adapting its connections so that the network exhibits the desired computational behavior for all input patterns. The process usually involves modifying the weights (moving the hyperplanes/hyperspheres); but sometimes it also involves modifying the actual topology of the network, i.e., adding or deleting connections from the network (adding or deleting hyperplanes/hyperspheres). In a sense, weight modification is more general than topology modification, since a network with abundant connections can learn to set any of its weights to zero, which has the same effect as deleting such weights. However, topological changes can improve both generalization and the speed of learning, by constraining the class of functions that the network is capable of learning. Topological changes will be discussed further in Section 3.3.5; in this section we will focus on weight modification.

Finding a set of weights that will enable a given network to compute a given function is usually a nontrivial procedure. An analytical solution exists only in the simplest case of pattern association, i.e., when the network is linear and the goal is to map a set of orthogonal input vectors to output vectors. In this case, the weights are given by

$$w_{ji} = \sum_p \frac{y_i^p t_j^p}{\left\| y^p \right\|^2} \tag{30}$$

where $\boldsymbol{y}$ is the input vector, $\boldsymbol{t}$ is the target vector, and p is the pattern index.

In general, networks are nonlinear and multilayered, and their weights can be trained only by an iterative procedure, such as *gradient descent* on a global performance measure (Hinton 1989). This requires multiple passes of training on the entire training set (rather like a person learning a new skill); each pass is called an *iteration* or an *epoch*. Moreover, since the accumulated knowledge is distributed over all of the weights, the weights must be modified very gently so as not to destroy all the previous learning. A small constant called the *learning rate* (ε) is thus used to control the magnitude of weight modifications. Finding a good value for the learning rate is very important — if the value is too small, learning takes forever; but if the value is too large, learning disrupts all the previous knowledge. Unfortunately, there is no analytical method for finding the optimal learning rate; it is usually optimized empirically, by just trying different values.

Most training procedures, including Equation (30), are essentially variations of the *Hebb Rule* (Hebb 1949), which reinforces the connection between two units if their output activations are correlated:

$$\Delta w_{ji} = \varepsilon y_i y_j \tag{31}$$

By reinforcing the correlation between active pairs of units during training, the network is prepared to activate the second unit if only the first one is known during testing.

One important variation of the above rule is the *Delta Rule* (or the *Widrow-Hoff Rule*), which applies when there is a target value for one of the two units. This rule reinforces the connection between two units if there is a correlation between the first unit's activation y_i and the second unit's error (or potential for error reduction) relative to its target t_j:

$$\Delta w_{ji} = \varepsilon y_i (t_j - y_j) \tag{32}$$

This rule decreases the relative error if y_i contributed to it, so that the network is prepared to compute an output y_j closer to t_j if only the first unit's activation y_i is known during testing. In the context of binary threshold units with a single layer of weights, the Delta Rule is known as the *Perceptron Learning Rule*, and it is guaranteed to find a set of weights representing a perfect solution, if such a solution exists (Rosenblatt 1962). In the context of multilayered networks, the Delta Rule is the basis for the *backpropagation* training procedure, which will be discussed in greater detail in Section 3.4.

Yet another variation of the Hebb Rule applies to the case of spherical functions, as in LVQ and RBF networks:

$$\Delta w_{ji} = \varepsilon (y_i - w_{ji}) y_j \tag{33}$$

This rule moves the spherical center w_{ji} closer to the input pattern y_i if the output class y_j is active.

3.3. A Taxonomy of Neural Networks

Now that we have presented the basic elements of neural networks, we will give an overview of some different types of networks. This overview will be organized in terms of the learning procedures used by the networks. There are three main classes of learning procedures:

- **supervised learning**, in which a "teacher" provides output targets for each input pattern, and corrects the network's errors explicitly;
- **semi-supervised** (or **reinforcement**) **learning**, in which a teacher merely indicates whether the network's response to a training pattern is "good" or "bad"; and
- **unsupervised learning**, in which there is no teacher, and the network must find regularities in the training data by itself.

Most networks fall squarely into one of these categories, but there are also various anomalous networks, such as **hybrid networks** which straddle these categories, and **dynamic networks** whose architectures can grow or shrink over time.

3.3.1. Supervised Learning

Supervised learning means that a "teacher" provides output targets for each input pattern, and corrects the network's errors explicitly. This paradigm can be applied to many types of networks, both feedforward and recurrent in nature. We will discuss these two cases separately.

3.3.1.1. Feedforward Networks

Perceptrons (Rosenblatt 1962) are the simplest type of feedforward networks that use supervised learning. A perceptron is comprised of binary threshold units arranged into layers, as shown in Figure 3.7. It is trained by the Delta Rule given in Equation (32), or variations thereof.

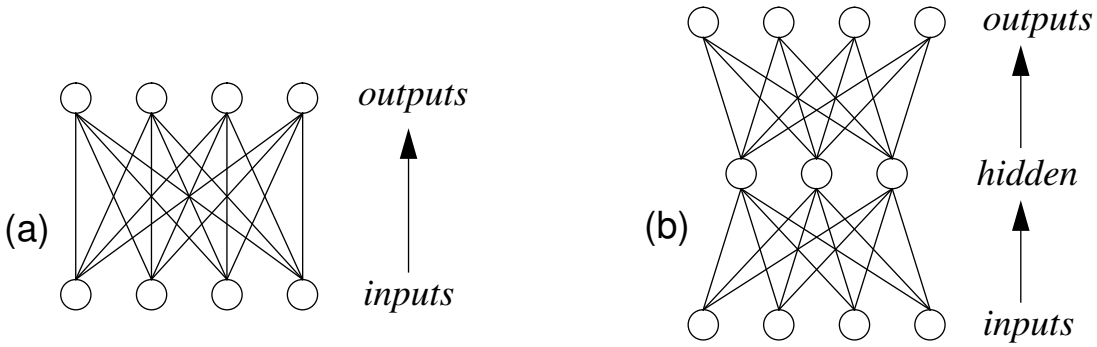

Figure 3.7: Perceptrons. (a) Single layer perceptron; (b) multi-layer perceptron.

In the case of a *single layer perceptron*, as shown in Figure 3.7(a), the Delta Rule can be applied directly. Because a perceptron's activations are binary, this general learning rule reduces to the *Perceptron Learning Rule*, which says that if an input is active ($y_i = 1$) and the output y_j is wrong, then w_{ji} should be either increased or decreased by a small amount ε, depending if the desired output is 1 or 0, respectively. This procedure is guaranteed to find a set of weights to correctly classify the patterns in any training set if the patterns are *linearly separable*, i.e., if they can be separated into two classes by a straight line, as illustrated in Figure 3.5(a). Most training sets, however, are not linearly separable (consider the simple XOR function, for example); in these cases we require multiple layers.

Multi-layer perceptrons (MLPs), as shown in Figure 3.7(b), can theoretically learn any function, but they are more complex to train. The Delta Rule cannot be applied directly to MLPs because there are no targets in the hidden layer(s). However, if an MLP uses continuous rather than discrete activation functions (i.e., sigmoids rather than threshold functions), then it becomes possible to use partial derivatives and the chain rule to derive the influence of any weight on any output activation, which in turn indicates how to modify that weight in order to reduce the network's error. This generalization of the Delta Rule is known as *back-propagation*; it will be discussed further in Section 3.4.

MLPs may have any number of hidden layers, although a single hidden layer is sufficient for many applications, and additional hidden layers tend to make training slower, as the terrain in weight space becomes more complicated. MLPs can also be architecturally constrained in various ways, for instance by limiting their connectivity to geometrically local areas, or by limiting the values of the weights, or tying different weights together.

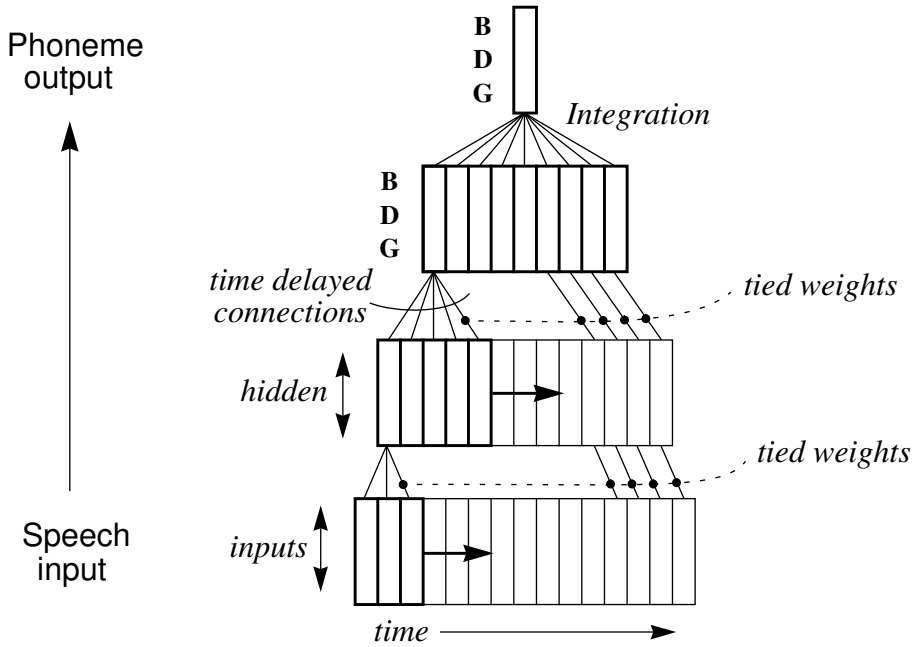

Figure 3.8: Time Delay Neural Network.

One type of constrained MLP which is especially relevant to this thesis is the *Time Delay Neural Network* (TDNN), shown in Figure 3.8. This architecture was initially developed for phoneme recognition (Lang 1989, Waibel et al 1989), but it has also been applied to handwriting recognition (Idan et al, 1992, Bodenhausen and Manke 1993), lipreading (Bregler et al, 1993), and other tasks. The TDNN operates on two-dimensional input fields, where the horizontal dimension is *time*[1]. Connections are "time delayed" to the extent that their connected units are temporally nonaligned. The TDNN has three special architectural features:

1. Its time delays are hierarchically structured, so that higher level units can integrate more temporal context and perform higher level feature detection.
2. Weights are *tied* along the time axis, i.e., corresponding weights at different temporal positions share the same value, so the network has relatively few free parameters, and it can generalize well.
3. The output units temporally integrate the results of local feature detectors distributed over time, so the network is shift invariant, i.e., it can recognize patterns no matter where they occur in time.

1. Assuming the task is speech recognition, or some other task in the temporal domain.

The TDNN is trained using standard backpropagation. The only unusual aspect of training a TDNN is that the tied weights are modified according to their averaged error signal, rather than independently.

Another network that can classify input patterns is a *Learned Vector Quantization* (LVQ) network (Kohonen 1989). An LVQ network is a single-layered network in which the outputs represent classes, and their weights from the inputs represent the centers of hyperspheres, as shown in Figure 3.5(b). Training involves moving the hyperspheres to cover the classes more accurately. Specifically[1], for each training pattern $\boldsymbol{x}$, if the best output y_1 is incorrect, while the second best output y_2 is correct, and if $\boldsymbol{x}$ is near the midpoint between the hyperspheres $\boldsymbol{w_1}$ and $\boldsymbol{w_2}$, then we move $\boldsymbol{w_1}$ toward $\boldsymbol{x}$, and $\boldsymbol{w_2}$ away from $\boldsymbol{x}$:

$$\begin{aligned} \Delta \boldsymbol{w_1} &= +\varepsilon\,(\boldsymbol{x} - \boldsymbol{w_1}) \\ \Delta \boldsymbol{w_2} &= -\varepsilon\,(\boldsymbol{x} - \boldsymbol{w_2}) \end{aligned} \tag{34}$$

3.3.1.2. Recurrent Networks

Hopfield (1982) studied neural networks that implement a kind of content-addressable associative memory. He worked with unstructured networks of binary threshold units with symmetric connections ($w_{ji} = w_{ij}$), in which activations are updated asynchronously; this type of recurrent network is now called a *Hopfield network*. Hopfield showed that if the weights in such a network were modified according to the Hebb Rule, then the training patterns would become *attractors* in state space. In other words, if the network were later presented with a corrupted version of one of the patterns, and the network's activations were updated in a random, asynchronous manner (using the previously trained weights), then the network would gradually reconstruct the whole activation pattern of the closest pattern in state space, and stabilize on that pattern. Hopfield's key insight was to analyze the network's dynamics in terms of a global *energy function*:

$$E = -\frac{1}{2}\sum_i \sum_{j \neq i} w_{ji} y_i y_j \tag{35}$$

which necessarily decreases (or remains the same) when any unit's activation is updated, and which reaches a minimum value for activation patterns corresponding to the stored memories. This implies that the network always settles to a stable state (although it may reach a local minimum corresponding to a spurious memory arising from interference between the stored memories).

A *Boltzmann Machine* (Ackley et al 1985) is a Hopfield network with hidden units, stochastic activations, and simulated annealing in its learning procedure. Each of these features contributes to its exceptional power. The hidden units allow a Boltzmann Machine to find higher order correlations in the data than a Hopfield network can find, so it can learn arbitrarily complex patterns. The stochastic (temperature-based) activations, as shown in Figure 3.4, allow a Boltzmann Machine to escape local minima during state evolution. Simulated annealing (i.e., the use of steadily decreasing temperatures during training) helps the

1. The training algorithm described here is known as LVQ2, an improvement over the original LVQ training algorithm.

network learn more efficiently than if it always used a low temperature, by vigorously "shaking" the network into viable neighborhoods of weight space during early training, and more gently jiggling the network into globally optimal positions during later training. Training a Boltzmann Machine involves modifying the weights so as to reduce the difference between two observed probability distributions:

$$\Delta w_{ji} = \frac{\varepsilon}{T}\left(p_{ij}^{+} - p_{ij}^{-} \right) \tag{36}$$

where T is the temperature, p_{ij}^{+} is the probability (averaged over all environmental inputs and measured at equilibrium) that the ith and jth units are both active when all of the visible units (inputs and/or outputs) have clamped values, and p_{ij}^{-} is the corresponding probability when the system is "free running", i.e., when nothing is clamped. Learning tends to be extremely slow in Boltzmann Machines, not only because it uses gradient descent, but also because at each temperature in the annealing schedule we must wait for the network to come to equilibrium, and then collect lots of statistics about its clamped and unclamped behavior. Nevertheless, Boltzmann Machines are theoretically very powerful, and they have been successfully applied to many problems.

Other types of recurrent networks have a layered structure with connections that feed back to earlier layers. Figure 3.9 shows two examples, known as the *Jordan network* (Jordan 1986) and the *Elman network* (Elman 1990). These networks feature a set of *context* units, whose activations are copied from either the outputs or the hidden units, respectively, and which are then fed forward into the hidden layer, supplementing the inputs. The context units give the networks a kind of decaying memory, which has proven sufficient for learning temporal structure over short distances, but not generally over long distances (Servan-Schreiber et al 1991). These networks can be trained with standard backpropagation, since all of the trainable weights are feedforward weights.

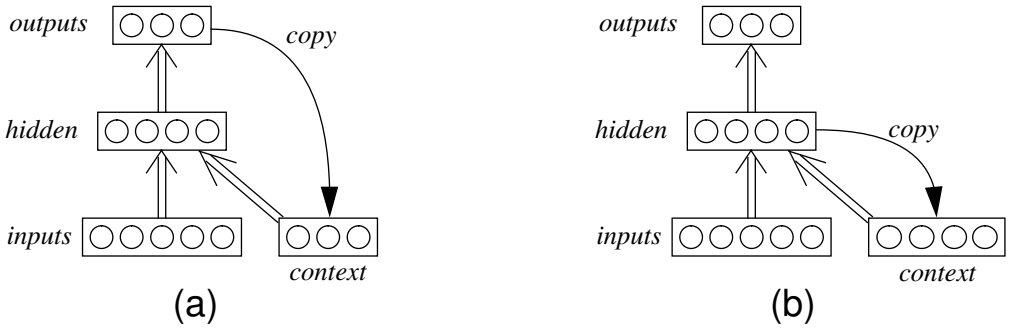

Figure 3.9: Layered recurrent networks. (a) Jordan network; (b) Elman network.

3.3.2. Semi-Supervised Learning

In semi-supervised learning (also called *reinforcement learning*), an external teacher does not provide explicit targets for the network's outputs, but only evaluates the network's behavior as "good" or "bad". Different types of semi-supervised networks are distinguished

not so much by their topologies (which are fairly arbitrary), but by the nature of their environment and their learning procedures. The environment may be either static or dynamic, i.e., the definition of "good" behavior may be fixed or it may change over time; likewise, evaluations may either be deterministic or probabilistic.

In the case of static environments (with either deterministic or stochastic evaluations), networks can be trained by the *associative reward-penalty algorithm* (Barto and Anandan 1985). This algorithm assumes stochastic output units (as in Figure 3.4) which enable the network to try out various behaviors. The problem of semi-supervised learning is reduced to the problem of supervised learning, by setting the training targets to be either the actual outputs or their negations, depending on whether the network's behavior was judged "good" or "bad"; the network is then trained using the Delta Rule, where the targets are compared against the network's mean outputs, and error is backpropagated through the network if necessary.

Another approach, which can be applied to either static or dynamic environments, is to introduce an auxiliary network which tries to model the environment (Munro 1987). This auxiliary network maps environmental data (consisting of both the input and output of the first network) to a reinforcement signal. Thus, the problem of semi-supervised learning is reduced to two stages of supervised learning with known targets — first the auxiliary network is trained to properly model the environment, and then backpropagation can be applied through both networks, so that each output of the original network has a distinct error signal coming from the auxiliary network.

A similar approach, which applies only to dynamic environments, is to enhance the auxiliary network so that it becomes a *critic* (Sutton 1984), which maps environmental data plus the reinforcement signal to a prediction of the future reinforcement signal. By comparing the expected and actual reinforcement signal, we can determine whether the original network's performance exceeds or falls short of expectation, and we can then reward or punish it accordingly.

3.3.3. Unsupervised Learning

In unsupervised learning, there is no teacher, and a network must detect regularities in the input data by itself. Such *self-organizing* networks can be used for compressing, clustering, quantizing, classifying, or mapping input data.

One way to perform unsupervised training is to recast it into the paradigm of supervised training, by designating an artificial target for each input pattern, and applying backpropagation. In particular, we can train a network to reconstruct the input patterns on the output layer, while passing the data through a bottleneck of hidden units. Such a network learns to preserve as much information as possible in the hidden layer; hence the hidden layer becomes a compressed representation of the input data. This type of network is often called an *encoder*, especially when the inputs/outputs are binary vectors. We also say that this network performs *dimensionality reduction*.

Other types of unsupervised networks (usually without hidden units) are trained with Hebbian learning, as in Equation (31). Hebbian learning can be used, for example, to train a sin-

gle linear unit to recognize the familiarity of an input pattern, or by extension to train a set of M linear output units to project an input pattern onto the M principal components of the distribution, thus forming a compressed representation of the inputs on the output layer. With linear units, however, the standard Hebb Rule would cause the weights to grow without bounds, hence this rule must be modified to prevent the weights from growing too large. One of several viable modifications is Sanger's Rule (Sanger 1989):

$$\Delta w_{ji} = \varepsilon \cdot y_j \cdot \left(y_i - \sum_{k=1}^{j} y_k w_{ki} \right) \tag{37}$$

This can be viewed as a form of weight decay (Krogh and Hertz, 1992). This rule uses nonlocal information, but it has the nice property that the M weight vectors $\boldsymbol{w}_j$ converge to the first M principal component directions, in order, normalized to unit length.

Linsker (1986) showed that a modified Hebbian learning rule, when applied to a multilayered network in which each layer is planar and has geometrically local connections to the next layer (as in the human visual system), can automatically develop useful feature detectors, such as center-surround cells and orientation-selective cells, very similar to those found in the human visual system.

Still other unsupervised networks are based on *competitive learning*, in which one output unit is considered the "winner"; these are known as *winner-take-all* networks. The winning unit may be found by lateral inhibitory connections on the output units (which drive down the losing activations to zero), or simply by comparative inspection of the output activations. Competitive learning is useful for clustering the data, in order to classify or quantize input patterns. Note that if the weights $\boldsymbol{w}_i$ to each output unit i are normalized, such that $\|\boldsymbol{w}_i\| = 1$ for all i, then maximizing the net input $\boldsymbol{w}_i \cdot \boldsymbol{y}$ is equivalent to minimizing the difference $\|\boldsymbol{w}_i - \boldsymbol{y}\|$; hence the goal of training can be seen as moving the weight vectors to the centers of hyperspherical input clusters, so as to minimize this distance. The standard competitive learning rule is thus the one given in Equation (33); when outputs are truly winner-take-all, this learning rule simplifies to

$$\Delta w_{j'i} = \varepsilon \cdot (y_i - w_{j'i}) \tag{38}$$

which is applied only to the winning output j'. Unfortunately, with this learning procedure, some units may be so far away from the inputs that they never win, and therefore never learn. Such *dead units* can be avoided by initializing the weights to match actual input samples, or else by relaxing the winner-take-all constraint so that losers learn as well as winners, or by using any of a number of other mechanisms (Hertz, Krogh, & Palmer 1991).

Carpenter and Grossberg (1988) developed networks called ART1 and ART2 (*Adaptive Resonance Theory* networks for binary and continuous inputs, respectively), which support competitive learning in such a way that a new cluster is formed whenever an input pattern is sufficiently different from any existing cluster, according to a *vigilance* parameter. Clusters are represented by individual output units, as usual; but in an ART network the output units are reserved until they are needed. Their network uses a search procedure, which can be implemented in hardware.

Kohonen (1989) developed a competitive learning algorithm which performs *feature mapping*, i.e., mapping patterns from an input space to an output space while preserving topological relations. The learning rule is

$$\Delta w_{ji} = \varepsilon \cdot \Lambda(j, j') \cdot (y_i - w_{ji}) \tag{39}$$

which augments the standard competitive learning rule by a *neighborhood function* $\Lambda(j, j')$, measuring the topological proximity between unit j and the winning unit j', so that units near j' are strongly affected, while distant units are less affected. This can be used, for example, to map two input coefficients onto a 2-dimensional set of output units, or to map a 2-dimensional set of inputs to a different 2-dimensional representation, as occurs in different layers of visual or somatic processing in the brain.

3.3.4. Hybrid Networks

Some networks combine supervised and unsupervised training in different layers. Most commonly, unsupervised training is applied at the lowest layer in order to cluster the data, and then backpropagation is applied at the higher layer(s) to associate these clusters with the desired output patterns. For example, in a *Radial Basis Function* network (Moody and Darken 1989), the hidden layer contains units that describe hyperspheres (trained with a standard competitive learning algorithm), while the output layer computes normalized linear combinations of these receptive field functions (trained with the Delta Rule). The attraction of such hybrid networks is that they reduce the multilayer backpropagation algorithm to the single-layer Delta Rule, considerably reducing training time. On the other hand, since such networks are trained in terms of independent modules rather than as an integrated whole, they have somewhat less accuracy than networks trained entirely with backpropagation.

3.3.5. Dynamic Networks

All of the networks discussed so far have a static architecture. But there are also *dynamic networks*, whose architecture can change over time, in order to attain optimal performance. Changing an architecture involves either deleting or adding elements (weights and/or units) in the network; these opposite approaches are called *pruning* and *construction*, respectively. Of these two approaches, pruning tends to be simpler, as it involves merely ignoring selected elements; but constructive algorithms tend to be faster, since the networks are small for much of their lives.

Pruning of course requires a way to identify the least useful elements in the network. One straightforward technique is to delete the weights with the smallest magnitude; this can improve generalization, but sometimes it also eliminates the wrong weights (Hassibi and Stork 1993). A more complex but more reliable approach, called *Optimal Brain Damage* (Le Cun et al, 1990b), identifies the weights whose removal will cause the least increase in the network's output error function; this requires the calculation of second-derivative information.

Among constructive algorithms, the *Cascade Correlation* algorithm (Fahlman and Lebiere 1990) is one of the most popular and effective. This algorithm starts with no hidden units, but gradually adds them (in depth-first fashion) as long as they help to cut down any remaining output error. At each stage of training, all previously trained weights in the network are frozen, and a pool of new candidate units are connected to all existing non-output units; each candidate unit is trained to maximize the correlation between the unit's output and the network's residual error, and then the most effective unit is fully integrated into the network (while the other candidates are discarded), and its weights to the output layer are fine-tuned. This process is repeated until the network has acceptable performance. The Cascade Correlation algorithm can quickly construct compact, powerful networks that exhibit excellent performance.

Bodenhausen (1994) has developed a constructive algorithm called *Automatic Structure Optimization*, designed for spacio-temporal tasks such as speech recognition and online handwriting recognition, especially given limited training data. The ASO algorithm starts with a small network, and adds more resources (including connections, time delays, hidden units, and state units) in a class-dependent way, under the guidance of confusion matrices obtained by cross-validation during training, in order to minimize the overall classification error. The ASO algorithm automatically optimized the architecture of MS-TDNNs, achieving results that were competitive with state-of-the-art systems that had been optimized by hand.

3.4. Backpropagation

Backpropagation, also known as *Error Backpropagation* or the *Generalized Delta Rule*, is the most widely used supervised training algorithm for neural networks. Because of its importance, we will discuss it in some detail in this section. We begin with a full derivation of the learning rule.

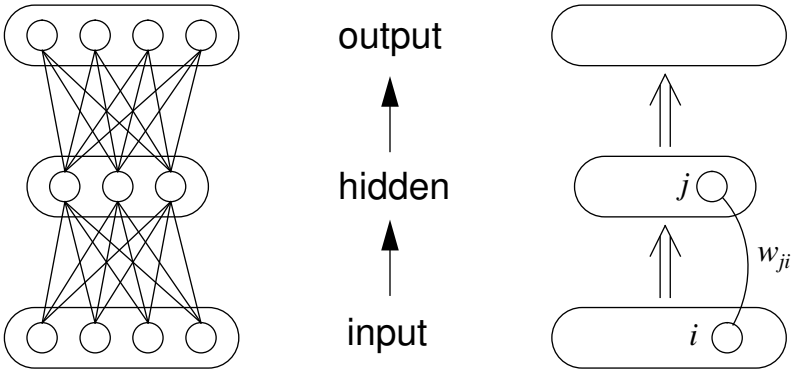

Figure 3.10: A feedforward neural network, highlighting the connection from unit i to unit j.

Suppose we have a multilayered feedforward network of nonlinear (typically sigmoidal) units, as shown in Figure 3.10. We want to find values for the weights that will enable the network to compute a desired function from input vectors to output vectors. Because the units compute nonlinear functions, we cannot solve for the weights analytically; so we will instead use a gradient descent procedure on some global error function E.

Let us define i, j, and k as arbitrary unit indices, O as the set of output units, p as training pattern indices (where each training pattern contains an input vector and output target vector), x_j^p as the net input to unit j for pattern p, y_j^p as the output activation of unit j for pattern p, w_{ji} as the weight from unit i to unit j, t_j^p as the target activation for unit j in pattern p (for $j \in O$), E^p as the global output error for training pattern p, and E as the global error for the entire training set. Assuming the most common type of network, we have

$$x_j^p = \sum_i w_{ji} y_i^p \tag{40}$$

$$y_j^p = \sigma(x_j^p) = \frac{1}{1 + e^{-x_j^p}} \tag{41}$$

It is essential that this activation function $y_j^p = \sigma(x_j^p)$ be differentiable, as opposed to non-differentiable as in a simple threshold function, because we will be computing its gradient in a moment.

The choice of error function is somewhat arbitrary[1]; let us assume the Sum Squared Error function

$$E^p = \frac{1}{2}\sum_j (y_j^p - t_j^p)^2 \qquad \text{where} \qquad j \in O \tag{42}$$

and

$$E = \sum_p E^p \tag{43}$$

We want to modify each weight w_{ji} in proportion to its influence on the error E, in the direction that will reduce E:

$$\Delta^p w_{ji} = -\varepsilon \cdot \frac{\partial E^p}{\partial w_{ji}} \tag{44}$$

where ε is a small constant, called the *learning rate*.

1. Popular choices for the global error function include Sum Squared Error: $E = \frac{1}{2}\sum_j (y_j - t_j)^2$; Cross Entropy: $E = -\sum_j (t_j \log y_j) + (1 - t_j) \log (1 - y_j)$; McClelland Error: $E = -\sum_j \log (1 - (t_j - y_j)^2)$; and the Classification Figure of Merit: $E = f(d)$ where $d = y_{\bar{c}} - y_c$ = the difference between the best incorrect output and the correct output, for example $E = (d + 1)^2$.

By the Chain Rule, and from Equations (41) and (40), we can expand this as follows:

$$\frac{\partial E^p}{\partial w_{ji}} = \frac{\partial E^p}{\partial y_j^p} \cdot \frac{\partial y_j^p}{\partial x_j^p} \cdot \frac{\partial x_j^p}{\partial w_{ji}} \qquad (45)$$

$$\overset{def.}{\downarrow} \quad \downarrow \quad \downarrow$$

$$= \gamma_j^p \cdot \sigma'(x_j^p) \cdot y_i^p$$

The first of these three terms, which introduces the shorthand definition $\gamma_j^p = \partial E^p / \partial y_j^p$, remains to be expanded. Exactly how it is expanded depends on whether j is an output unit or not. If j is an output unit, then from Equation (42) we have

$$j \in O \quad \Rightarrow \quad \gamma_j^p = \frac{\partial E^p}{\partial y_j^p} = y_j^p - t_j^p \qquad (46)$$

But if j is not an output unit, then it directly affects a set of units $k \in \text{out}(j)$, as illustrated in Figure 3.11, and by the Chain Rule we obtain

$$j \notin O \quad \Rightarrow \quad \gamma_j^p = \frac{\partial E^p}{\partial y_j^p} = \sum_{k \in \text{out}(j)} \frac{\partial E^p}{\partial y_k^p} \cdot \frac{\partial y_k^p}{\partial x_k^p} \cdot \frac{\partial x_k^p}{\partial y_j^p} \qquad (47)$$

$$\downarrow \quad \downarrow \quad \downarrow$$

$$= \sum_{k \in \text{out}(j)} \gamma_k^p \cdot \sigma'(x_k^p) \cdot w_{kj}$$

The recursion in this equation, in which γ_j^p refers to γ_k^p, says that the γ's (and hence Δw's) in each layer can be derived directly from the γ's in the next layer. Thus, we can derive all the γ's in a multilayer network by starting at the output layer (using Equation 46) and working our way backwards towards the input layer, one layer at a time (using Equation 47). This learning procedure is called "backpropagation" because the error terms (γ's) are propagated through the network in this backwards direction.

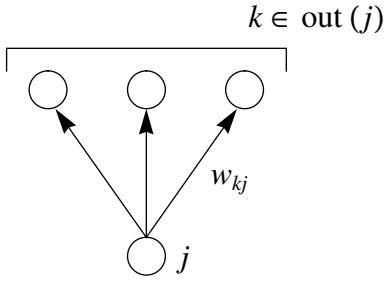

Figure 3.11: If unit *j* is not an output unit, then it directly affects some units *k* in the next layer.

To summarize the learning rule, we have

$$\Delta^p w_{ji} = -\varepsilon \cdot \gamma_j^p \cdot \sigma'(x_j^p) \cdot y_i^p \tag{48}$$

where

$$\gamma_j^p = \frac{\partial E^p}{\partial y_j^p} = \begin{cases} (y_j^p - t_j^p) & \text{if} \quad j \in O \\ \sum_{k \in \text{out}(j)} \gamma_k^p \cdot \sigma'(x_k^p) \cdot w_{kj} & \text{if} \quad j \notin O \end{cases} \tag{49}$$

Or, equivalently, if we wish to define

$$\delta_j^p = \frac{\partial E^p}{\partial x_j^p} = \gamma_j^p \cdot \sigma'(x_j^p) \tag{50}$$

then we have

$$\Delta^p w_{ji} = -\varepsilon \cdot \delta_j^p \cdot y_i^p \tag{51}$$

where

$$\delta_j^p = \frac{\partial E^p}{\partial x_j^p} = \begin{cases} (y_j^p - t_j^p) \cdot \sigma'(x_j^p) & \text{if} \quad j \in O \\ \sum_{k \in \text{out}(j)} (\delta_k^p \cdot w_{kj}) \cdot \sigma'(x_j^p) & \text{if} \quad j \notin O \end{cases} \tag{52}$$

Backpropagation is a much faster learning procedure than the Boltzmann Machine training algorithm, but it can still take a long time for it to converge to an optimal set of weights. Learning may be accelerated by increasing the learning rate ε, but only up to a certain point, because when the learning rate becomes too large, weights become excessive, units become saturated, and learning becomes impossible. Thus, a number of other heuristics have been developed to accelerate learning. These techniques are generally motivated by an intuitive image of backpropagation as a gradient descent procedure. That is, if we envision a hilly landscape representing the error function E over weight space, then backpropagation tries to find a local minimum value of E by taking incremental steps Δw_{ji} down the current hillside, i.e., in the direction $-\partial E^p / \partial w_{ji}$. This image helps us see, for example, that if we take too large of a step, we run the risk of moving so far down the current hillside that we find ourselves shooting up some other nearby hillside, with possibly a higher error than before.

Bearing this image in mind, one common heuristic for accelerating the learning process is known as *momentum* (Rumelhart et al 1986), which tends to push the weights further along in the most recently useful direction:

$$\Delta w_{ji}(t) = \left(-\varepsilon \cdot \frac{\partial E^p}{\partial w_{ji}} \right) + \left(\alpha \cdot \Delta w_{ji}(t-1) \right) \tag{53}$$

where α is the momentum constant, usually between 0.50 and 0.95. This heuristic causes the step size to steadily increase as long as we keep moving down a long gentle valley, and also to recover from this behavior when the error surface forces us to change direction. A more elaborate and more powerful heuristic is to use second-derivative information to estimate how far down the hillside to travel; this is used in techniques such as *conjugate gradient* (Barnard 1992) and *quickprop* (Fahlman 1988).

Ordinarily the weights are updated after each training pattern (this is called *online training*. But sometimes it is more effective to update the weights only after accumulating the gradients over a whole batch of training patterns (this is called *batch training*), because by superimposing the error landscapes for many training patterns, we can find a direction to move which is best for the whole group of patterns, and then confidently take a larger step in that direction. Batch training is especially helpful when the training patterns are uncorrelated (for then it eliminates the waste of Brownian motion), and when used in conjunction with aggressive heuristics like quickprop which require accurate estimates of the landscape's surface.

Because backpropagation is a simple gradient descent procedure, it is unfortunately susceptible to the problem of local minima, i.e., it may converge upon a set of weights that are locally optimal but globally suboptimal. Experience has shown that local minima tend to cause more problems for artificial domains (as in boolean logic) than for real domains (as in perceptual processing), reflecting a difference in terrain in weight space. In any case, it is possible to deal with the problem of local minima by adding noise to the weight modifications.

3.5. Relation to Statistics

Neural networks have a close relationship to many standard statistical techniques. In this section we discuss some of these commonalities.

One of the most important tasks in statistics is the classification of data. Suppose we want to classify an input vector x into one of two classes, c_1 and c_2. Obviously our decision should correspond to the class with the highest probability of being correct, i.e., we should decide on class c_1 if $P(c_1|x) > P(c_2|x)$. Normally these posterior probabilities are not known, but the "inverse" information, namely the probability distributions $P(x|c_1)$ and $P(x|c_2)$, may be known. We can convert between posterior probabilities and distributions using Bayes Rule:

$$P(c_i|x) = \frac{P(x|c_i) \cdot P(c_i)}{P(x)} \qquad \text{where} \qquad P(x) = \sum_i P(x|c_i) P(c_i) \tag{54}$$

It follows directly that we should choose class c_1 if

$$P(x|c_1)\,P(c_1) > P(x|c_2)\,P(c_2) \tag{55}$$

This criterion is known as the *Bayes Decision Rule*. Given perfect knowledge of the distributions $P(x|c_i)$ and priors $P(c_i)$, this decision rule is guaranteed to minimize the classification error rate.

Typically, however, the distributions, priors, and posteriors are all unknown, and all we have is a collection of sample data points. In this case, we must analyze and model the data in order to classify new data accurately. If the existing data is labeled, then we can try to estimate either the posterior probabilities $P(c|x)$, or the distributions $P(x|c)$ and priors $P(c)$, so that we can use Bayes Decision Rule; alternatively, we can try to find boundaries that separate the classes, without trying to model their probabilities explicitly. If the data is unlabeled, we can first try to cluster it, in order to identify meaningful classes. Each of the above tasks can be performed either by a statistical procedure or by a neural network.

For example, if we have labeled data, and we wish to perform Bayesian classification, there are many statistical techniques available for modeling the data (Duda and Hart 1973). These include both parametric and nonparametric approaches. In the parametric approach, we assume that a distribution $P(x|c)$ has a given parametric form (e.g., a gaussian density), and then try to estimate its parameters; this is commonly done by a procedure called *Maximum Likelihood* estimation, which finds the parameters that maximize the likelihood of having generated the observed data. In the non-parametric approach, we may use a volumetric technique called *Parzen windows* to estimate the local density of samples at any point x; the robustness of this technique is often improved by scaling the local volume around x so that it always contains k samples, in a variation called *k-nearest neighbor estimation*. (Meanwhile, the priors $P(c)$ can be estimated simply by counting.) Alternatively, the posterior probability $P(c|x)$ can also be estimated using nonparametric techniques, such as the *k-nearest neighbor rule*, which classifies x in agreement with the majority of its k nearest neighbors.

A neural network also supports Bayesian classification by forming a model of the training data. More specifically, when a multilayer perceptron is asymptotically trained as a 1-of-N classifier using the mean squared error (MSE) or similar error function, its output activations learn to approximate the posterior probability $P(c|x)$, with an accuracy that improves with the size of the training set. A proof of this important fact can be found in Appendix B.

Another way to use labeled training data is to find boundaries that separate the classes. In statistics, this can be accomplished by a general technique called *discriminant analysis*. An important instance of this is *Fisher's linear discriminant*, which finds a line that gives the best discrimination between classes when data points are projected onto this line. This line is equivalent to the weight vector of a single layer perceptron with a single output that has been trained to discriminate between the classes, using the Delta Rule. In either case, the classes will be optimally separated by a hyperplane drawn perpendicular to the line or the weight vector, as shown in Figure 3.5(a).

Unlabeled data can be clustered using statistical techniques — such as *nearest-neighbor clustering, minimum squared error clustering,* or *k-means clustering* (Krishnaiah and Kanal

1982) — or alternatively by neural networks that are trained with competitive learning. In fact, k-means clustering is exactly equivalent to the standard competitive learning rule, as given in Equation (38), when using batch updating (Hertz et al 1991).

When analyzing high-dimensional data, it is often desirable to reduce its dimensionality, i.e., to project it into a lower-dimensional space while preserving as much information as possible. Dimensionality reduction can be performed by a statistical technique called *Principal Components Analysis* (PCA), which finds a set of M orthogonal vectors that account for the greatest variance in the data (Jolliffe 1986). Dimensionality reduction can also be performed by many types of neural networks. For example, a single layer perceptron, trained by an unsupervised competitive learning rule called *Sanger's Rule* (Equation 37), yields weights that equal the principal components of the training data, so that the network's outputs form a compressed representation of any input vector. Similarly, an encoder network — i.e., a multilayer perceptron trained by backpropagation to reproduce its input vectors on its output layer — forms a compressed representation of the data in its hidden units.

It is sometimes claimed that neural networks are simply a new formulation of old statistical techniques. While there is considerable overlap between these two fields, neural networks are attractive in their own right because they offer a general, uniform, and intuitive framework which can be applied equally well in statistical and non-statistical contexts.

4. Related Research

4.1. Early Neural Network Approaches

Because speech recognition is basically a pattern recognition problem, and because neural networks are good at pattern recognition, many early researchers naturally tried applying neural networks to speech recognition. The earliest attempts involved highly simplified tasks, e.g., classifying speech segments as voiced/unvoiced, or nasal/fricative/plosive. Success in these experiments encouraged researchers to move on to phoneme classification; this task became a proving ground for neural networks as they quickly achieved world-class results. The same techniques also achieved some success at the level of word recognition, although it became clear that there were scaling problems, which will be discussed later.

There are two basic approaches to speech classification using neural networks: static and dynamic, as illustrated in Figure 4.1. In static classification, the neural network sees all of the input speech at once, and makes a single decision. By contrast, in dynamic classification, the neural network sees only a small window of the speech, and this window slides over the input speech while the network makes a series of local decisions, which have to be integrated into a global decision at a later time. Static classification works well for phoneme recognition, but it scales poorly to the level of words or sentences; dynamic classification scales better. Either approach may make use of recurrent connections, although recurrence is more often found in the dynamic approach.

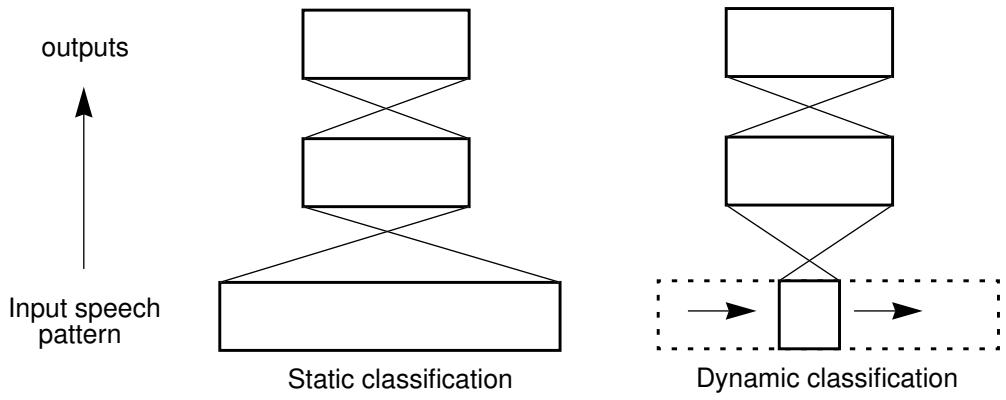

Figure 4.1: Static and dynamic approaches to classification.

In the following sections we will briefly review some representative experiments in phoneme and word classification, using both static and dynamic approaches.

4.1.1. Phoneme Classification

Phoneme classification can be performed with high accuracy by using either static or dynamic approaches. Here we review some typical experiments using each approach.

4.1.1.1. Static Approaches

A simple but elegant experiment was performed by Huang & Lippmann (1988), demonstrating that neural networks can form complex decision surfaces from speech data. They applied a multilayer perceptron with only 2 inputs, 50 hidden units, and 10 outputs, to Peterson & Barney's collection of vowels produced by men, women, & children, using the first two formants of the vowels as the input speech representation. After 50,000 iterations of training, the network produced the decision regions shown in Figure 4.2. These decision regions are nearly optimal, resembling the decision regions that would be drawn by hand, and they yield classification accuracy comparable to that of more conventional algorithms, such as k-nearest neighbor and Gaussian classification.

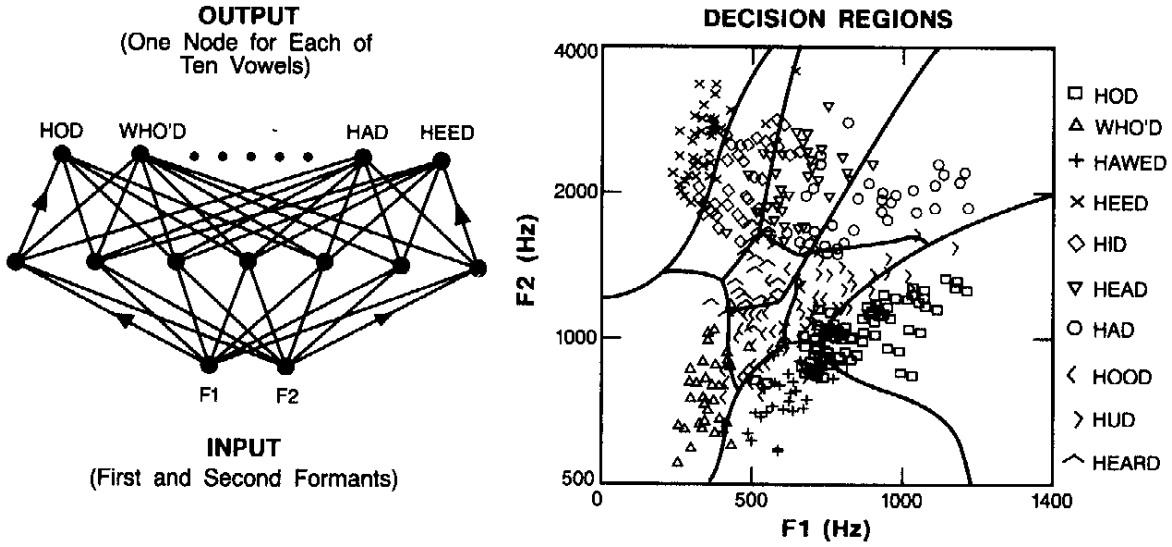

Figure 4.2: Decision regions formed by a 2-layer perceptron using backpropagation training and vowel formant data. (From Huang & Lippmann, 1988.)

In a more complex experiment, Elman and Zipser (1987) trained a network to classify the vowels /a,i,u/ and the consonants /b,d,g/ as they occur in the utterances ba,bi,bu; da,di,du; and ga,gi,gu. Their network input consisted of 16 spectral coefficients over 20 frames (covering an entire 64 msec utterance, centered by hand over the consonant's voicing onset); this was fed into a hidden layer with between 2 and 6 units, leading to 3 outputs for either vowel or consonant classification. This network achieved error rates of roughly 0.5% for vowels and 5.0% for consonants. An analysis of the hidden units showed that they tend to be fea-

ture detectors, discriminating between important classes of sounds, such as consonants versus vowels.

Among the most difficult of classification tasks is the so-called E-set, i.e., discriminating between the rhyming English letters "B, C, D, E, G, P, T, V, Z". Burr (1988) applied a static network to this task, with very good results. His network used an input window of 20 spectral frames, automatically extracted from the whole utterance using energy information. These inputs led directly to 9 outputs representing the E-set letters. The network was trained and tested using 180 tokens from a single speaker. When the early portion of the utterance was oversampled, effectively highlighting the disambiguating features, recognition accuracy was nearly perfect.

4.1.1.2. Dynamic Approaches

In a seminal paper, Waibel et al (1987=1989) demonstrated excellent results for phoneme classification using a Time Delay Neural Network (TDNN), shown in Figure 4.3. This architecture has only 3 and 5 delays in the input and hidden layer, respectively, and the final output is computed by integrating over 9 frames of phoneme activations in the second hidden layer. The TDNN's design is attractive for several reasons: its compact structure economizes on weights and forces the network to develop general feature detectors; its hierarchy of delays optimizes these feature detectors by increasing their scope at each layer; and its temporal integration at the output layer makes the network shift invariant (i.e., insensitive to the exact positioning of the speech). The TDNN was trained and tested on 2000 samples of /b,d,g/ phonemes manually excised from a database of 5260 Japanese words. The TDNN achieved an error rate of 1.5%, compared to 6.5% achieved by a simple HMM-based recognizer.

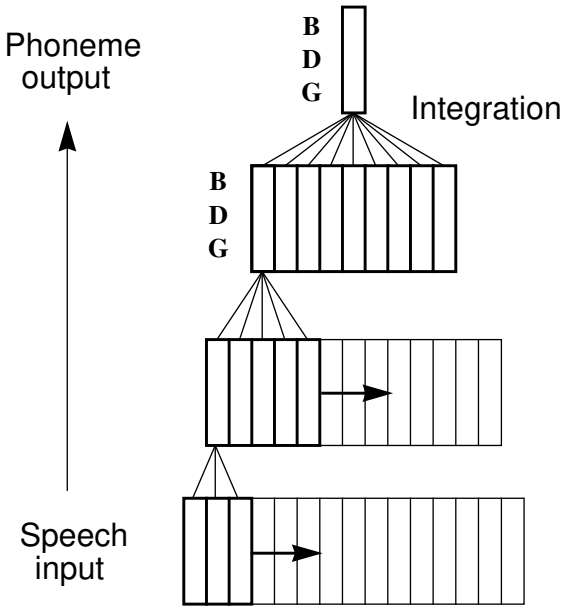

Figure 4.3: Time Delay Neural Network.

In later work (Waibel 1989a), the TDNN was scaled up to recognize all 18 Japanese consonants, using a modular approach which significantly reduced training time while giving slightly better results than a simple TDNN with 18 outputs. The modular approach consisted of training separate TDNNs on small subsets of the phonemes, and then combining these networks into a larger network, supplemented by some "glue" connections which received a little extra training while the primary modules remained fixed. The integrated network achieved an error rate of 4.1% on the 18 phonemes, compared to 7.3% achieved by a relatively advanced HMM-based recognizer.

McDermott & Katagiri (1989) performed an interesting comparison between Waibel's TDNN and Kohonen's LVQ2 algorithm, using the same /b,d,g/ database and similar conditions. The LVQ2 system was trained to quantize a 7-frame window of 16 spectral coefficients into a codebook of 150 entries, and during testing the distance between each input window and the nearest codebook vector was integrated over 9 frames, as in the TDNN, to produce a shift-invariant phoneme hypothesis. The LVQ2 system achieved virtually the same error rate as the TDNN (1.7% vs. 1.5%), but LVQ2 was much faster during training, slower during testing, and more memory-intensive than the TDNN.

In contrast to the feedforward networks described above, recurrent networks are generally trickier to work with and slower to train; but they are also theoretically more powerful, having the ability to represent temporal sequences of unbounded depth, without the need for artificial time delays. Because speech is a temporal phenomenon, many researchers consider recurrent networks to be more appropriate than feedforward networks, and some researchers have actually begun applying recurrent networks to speech.

Prager, Harrison, & Fallside (1986) made an early attempt to apply Boltzmann machines to an 11-vowel recognition task. In a typical experiment, they represented spectral inputs with 2048 binary inputs, and vowel classes with 8 binary outputs; their network also had 40 hidden units, and 7320 weights. After applying simulated annealing for many hours in order to train on 264 tokens from 6 speakers, the Boltzmann machine attained a multi-speaker error rate of 15%. This and later experiments suggested that while Boltzmann machines can give good accuracy, they are impractically slow to train.

Watrous (1988) applied recurrent networks to a set of basic discrimination tasks. In his system, framewise decisions were temporally integrated via recurrent connections on the output units, rather than by explicit time delays as in a TDNN; and his training targets were Gaussian-shaped pulses, rather than constant values, to match the ramping behavior of his recurrent outputs. Watrous obtained good results on a variety of discrimination tasks, after optimizing the non-output delays and sizes of his networks separately for each task. For example, the classification error rate was 0.8% for the consonants /b,d,g/, 0.0% for the vowels /a,i,u/, and 0.8% for the word pair "rapid/rabid".

Robinson and Fallside (1988) applied another kind of recurrent network, first proposed by Jordan (1986), to phoneme classification. In this network, output activations are copied to a "context" layer, which is then fed back like additional inputs to the hidden layer (as shown in Figure 3.9). The network was trained using "back propagation through time", an algorithm first suggested by Rumelhart et al (1986), which unfolds or replicates the network at each moment of time. Their recurrent network outperformed a feedforward network with

comparable delays, achieving 22.7% versus 26.0% error for speaker-dependent recognition, and 30.8% versus 40.8% error for multi-speaker recognition. Training time was reduced to a reasonable level by using a 64-processor array of transputers.

4.1.2. Word Classification

Word classification can also be performed with either static or dynamic approaches, although dynamic approaches are better able to deal with temporal variability over the duration of a word. In this section we review some experiments with each approach.

4.1.2.1. Static Approaches

Peeling and Moore (1987) applied MLPs to digit recognition with excellent results. They used a static input buffer of 60 frames (1.2 seconds) of spectral coefficients, long enough for the longest spoken word; briefer words were padded with zeros and positioned randomly in the 60-frame buffer. Evaluating a variety of MLP topologies, they obtained the best performance with a single hidden layer with 50 units. This network achieved accuracy near that of an advanced HMM system: error rates were 0.25% versus 0.2% in speaker-dependent experiments, or 1.9% versus 0.6% for multi-speaker experiments, using a 40-speaker database of digits from RSRE. In addition, the MLP was typically five times faster than the HMM system.

Kammerer and Kupper (1988) applied a variety of networks to the TI 20-word database, finding that a single-layer perceptron outperformed both multi-layer perceptrons and a DTW template-based recognizer in many cases. They used a static input buffer of 16 frames, into which each word was linearly normalized, with 16 2-bit coefficients per frame; performance improved slightly when the training data was augmented by temporally distorted tokens. Error rates for the SLP versus DTW were 0.4% versus 0.7% in speaker-dependent experiments, or 2.7% versus 2.5% for speaker-independent experiments.

Lippmann (1989) points out that while the above results seem impressive, they are mitigated by evidence that these small-vocabulary tasks are not really very difficult. Burton et al (1985) demonstrated that a simple recognizer based on whole-word vector quantization, without time alignment, can achieve speaker-dependent error rates as low as 0.8% for the TI 20-word database, or 0.3 for digits. Thus it is not surprising that simple networks can achieve good results on these tasks, in which temporal information is not very important.

Burr (1988) applied MLPs to the more difficult task of alphabet recognition. He used a static input buffer of 20 frames, into which each spoken letter was linearly normalized, with 8 spectral coefficients per frame. Training on three sets of the 26 spoken letters and testing on a fourth set, an MLP achieved an error rate of 15% in speaker-dependent experiments, matching the accuracy of a DTW template-based approach.

4.1.2.2. Dynamic Approaches

Lang et al (1990) applied TDNNs to word recognition, with good results. Their vocabulary consisting of the highly confusable spoken letters “B, D, E, V”. In early experiments,

training and testing were simplified by representing each word by a 144 msec segment centered on its vowel segment, where the words differed the most from each other. Using such pre-segmented data, the TDNN achieved a multispeaker error rate of 8.5%. In later experiments, the need for pre-segmentation was avoided by classifying a word according to the output that received the highest activation at any position of the input window relative to the whole utterance; and training used 216 msec segments roughly centered on vowel onsets according to an automatic energy-based segmentation technique. In this mode, the TDNN achieved an error rate of 9.5%. The error rate fell to 7.8% when the network received additional negative training on counter examples randomly selected from the background "E" sounds. This system compared favorably to an HMM which achieved about 11% error on the same task (Bahl et al 1988).

Tank & Hopfield (1987) proposed a "Time Concentration" network, which represents words by a weighted sum of evidence that is delayed, with proportional dispersion, until the end of the word, so that activation is concentrated in the correct word's output at the end of the utterance. This system was inspired by research on the auditory processing of bats, and a working prototype was actually implemented in parallel analog hardware. Unnikrishnan et al (1988) reported good results for this network on simple digit strings, although Gold (1988) obtained results no better than a standard HMM when he applied a hierarchical version of the network to a large speech database.

Among the early studies using recurrent networks, Prager, Harrison, & Fallside (1986) configured a Boltzmann machine to copy the output units into "state" units which were fed back into the hidden layer, as in a so-called Jordan network, thereby representing a kind of first-order Markov model. After several days of training, the network was able to correctly identify each of the words in its two training sentences. Other researchers have likewise obtained good results with Boltzmann machines, but only after an exorbitant amount of training.

Franzini, Witbrock, & Lee (1989) compared the performance of a recurrent network and a feedforward network on a digit recognition task. The feedforward network was an MLP with a 500 msec input window, while the recurrent network had a shorter 70 msec input window but a 500 msec state buffer. They found no significant difference in the recognition accuracy of these systems, suggesting that it's important only that a network have some form of memory, regardless of whether it's represented as a feedforward input buffer or a recurrent state layer.

4.2. The Problem of Temporal Structure

We have seen that phoneme recognition can easily be performed using either static or dynamic approaches. We have also seen that word recognition can likewise be performed with either approach, although dynamic approaches now become preferable because the wider temporal variability in a word implies that invariances are localized, and that local features should be temporally integrated. Temporal integration itself can easily be performed by a network (e.g., in the output layer of a TDNN), as long as the operation can be

described statically (to match the network's fixed resources); but as we consider larger chunks of speech, with greater temporal variability, it becomes harder to map that variability into a static framework. As we continue scaling up the task from word recognition to sentence recognition, temporal variability not only becomes more severe, but it also acquires a whole new dimension — that of compositional structure, as governed by a grammar.

The ability to compose structures from simpler elements — implying the usage of some sort of variables, binding, modularity, and rules — is clearly required in any system that claims to support natural language processing (Pinker and Prince 1988), not to mention general cognition (Fodor and Pylyshyn 1988). Unfortunately, it has proven very difficult to model compositionality within the pure connectionist framework, although a number of researchers have achieved some early, limited success along these lines. Touretzky and Hinton (1988) designed a distributed connectionist production system, which dynamically retrieves elements from working memory and uses their components to contruct new states. Smolensky (1990) proposed a mechanism for performing variable binding, based on tensor products. Servan-Schreiber, Cleeremans, and McClelland (1991) found that an Elman network was capable of learning some aspects of grammatical structure. And Jain (1992) designed a modular, highly structured connectionist natural language parser that compared favorably to a standard LR parser.

But each of these systems is exploratory in nature, and their techniques are not yet generally applicable. It is clear that connectionist research in temporal and compositional modeling is still in its infancy, and it is premature to rely on neural networks for temporal modeling in a speech recognition system.

4.3. NN-HMM Hybrids

We have seen that neural networks are excellent at acoustic modeling and parallel implementations, but weak at temporal and compositional modeling. We have also seen that Hidden Markov Models are good models overall, but they have some weaknesses too. In this section we will review ways in which researchers have tried to combine these two approaches into various hybrid systems, capitalizing on the strengths of each approach. Much of the research in this section was conducted at the same time that this thesis was being written.

4.3.1. NN Implementations of HMMs

Perhaps the simplest way to integrate neural networks and Hidden Markov Models is to simply implement various pieces of HMM systems using neural networks. Although this does not improve the accuracy of an HMM, it does permit it to be parallelized in a natural way, and incidentally showcases the flexibility of neural networks.

Lippmann and Gold (1987) introduced the *Viterbi Net*, illustrated in Figure 4.4, which is a neural network that implements the Viterbi algorithm. The input is a temporal sequence of speech frames, presented one at a time, and the final output (after T time frames) is the

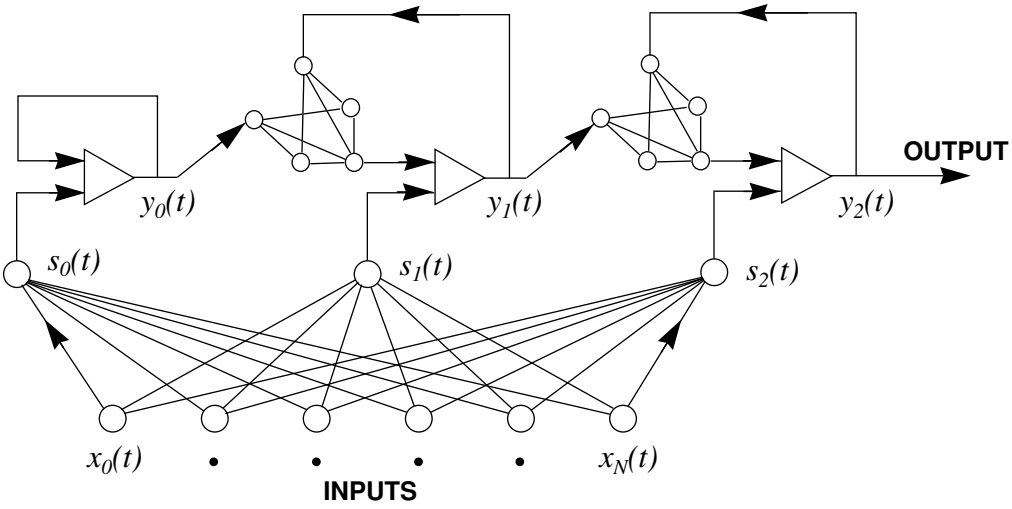

Figure 4.4: Viterbi Net: a neural network that implements the Viterbi algorithm.

cumulative score along the Viterbi alignment path, permitting isolated word recognition via subsequent comparison of the outputs of several Viterbi Nets running in parallel. (The Viterbi Net cannot be used for continuous speech recognition, however, because it yields no backtrace information from which the alignment path could be recovered.) The weights in the lower part of the Viterbi Net are preassigned in such a way that each node s_i computes the local score for state i in the current time frame, implementing a Gaussian classifier. The knotlike upper networks compute the maximum of their two inputs. The triangular nodes are threshold logic units that simply sum their two inputs (or output zero if the sum is negative), and delay the output by one time frame, for synchronization purposes. Thus, the whole network implements a left-to-right HMM with self-transitions, and the final output $y_F(T)$ represents the cumulative score in state F at time T along the optimal alignment path. It was tested on 4000 word tokens from the 9-speaker 35-word Lincoln Stress-Style speech database, and obtained results essentially identical with a standard HMM (0.56% error).

In a similar spirit, Bridle (1990) introduced the *AlphaNet*, which is a neural network that computes $\alpha_j(t)$, i.e., the forward probability of an HMM producing the partial sequence y_1^t and ending up in state j, so that isolated words can be recognized by comparing their final scores $\alpha_F(T)$. Figure 4.5 motivates the construction of an AlphaNet. The first panel illustrates the basic recurrence, $\alpha_j(t) = \sum_i \alpha_i(t-1)\, a_{ij}$. The second panel shows how this recurrence may be implemented using a recurrent network. The third panel shows how the additional term $b_j(y_t)$ can be factored into the equation, using sigma-pi units, so that the AlphaNet properly computes $\alpha_j(t) = \sum_i \alpha_i(t-1)\, a_{ij} b_j(y_t)$.

4.3.2. Frame Level Training

Rather than simply reimplementing an HMM using neural networks, most researchers have been exploring ways to enhance HMMs by designing hybrid systems that capitalize on the respective strengths of each technology: temporal modeling in the HMM and acous-

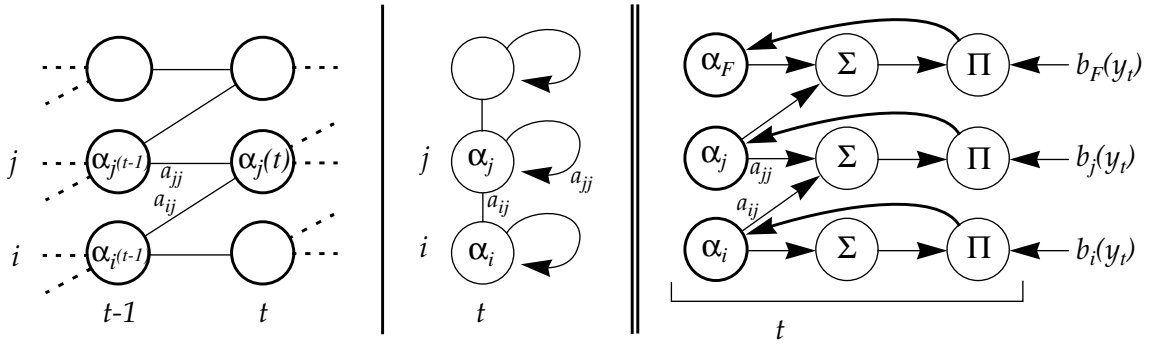

Figure 4.5: Construction of an AlphaNet (final panel).

tic modeling in neural networks. In particular, neural networks are often trained to compute emission probabilities for HMMs. Neural networks are well suited to this mapping task, and they also have a theoretical advantage over HMMs, because unlike discrete density HMMs, they can accept continuous-valued inputs and hence don't suffer from quantization errors; and unlike continuous density HMMs, they don't make any dubious assumptions about the parametric shape of the density function. There are many ways to design and train a neural network for this purpose. The simplest is to map frame inputs directly to emission symbol outputs, and to train such a network on a frame-by-frame basis. This approach is called *Frame Level Training*.

Frame level training has been extensively studied by researchers at Philips, ICSI, and SRI. Initial work by Bourlard and Wellekens (1988=1990) focused on the theoretical links between Hidden Markov Models and neural networks, establishing that neural networks estimate posterior probabilities which should be divided by priors in order to yield likelihoods for use in an HMM. Subsequent work at ICSI and SRI (Morgan & Bourlard 1990, Renals et al 1992, Bourlard & Morgan 1994) confirmed this insight in a series of experiments leading to excellent results on the Resource Management database. The simple MLPs in these experiments typically used an input window of 9 speech frames, 69 phoneme output units, and hundreds or even thousands of hidden units (taking advantage of the fact that more hidden units always gave better results); a parallel computer was used to train millions of weights in a reasonable amount of time. Good results depended on careful use of the neural networks, with techniques that included online training, random sampling of the training data, cross-validation, step size adaptation, heuristic bias initialization, and division by priors during recognition. A baseline system achieved 12.8% word error on the RM database using speaker-independent phoneme models; this improved to 8.3% by adding multiple pronunciations and cross-word modeling, and further improved to 7.9% by interpolating the likelihoods obtained from the MLP with those from SRI's DECIPHER system (which obtained 14.0% by itself under similar conditions). Finally, it was demonstrated that when using the same number of parameters, an MLP can outperform an HMM (e.g., achieving 8.3% vs 11.0% word error with 150,000 parameters), because an MLP makes fewer questionable assumptions about the parameter space.

Franzini, Lee, & Waibel (1990) have also studied frame level training. They started with an HMM, whose emission probabilities were represented by a histogram over a VQ codebook, and replaced this mechanism by a neural network that served the same purpose; the targets for this network were continuous probabilities, rather than binary classes as used by Bourlard and his colleagues. The network's input was a window containing seven frames of speech (70 msec), and there was an output unit for each probability distribution to be modeled[1]. Their network also had two hidden layers, the first of which was recurrent, via a buffer of the past 10 copies of the hidden layer which was fed back into that same hidden layer, in a variation of the Elman Network architecture. (This buffer actually represented 500 msec of history, because the input window was advanced 5 frames, or 50 msec, at a time.) The system was evaluated on the TI/NBS Speaker-Independent Continuous Digits Database, and achieved 98.5% word recognition accuracy, close to the best known result of 99.5%.

4.3.3. Segment Level Training

An alternative to frame-level training is segment-level training, in which a neural network receives input from an entire segment of speech (e.g., the whole duration of a phoneme), rather than from a single frame or a fixed window of frames. This allows the network to take better advantage of the correlation that exists among all the frames of the segment, and also makes it easier to incorporate segmental information, such as duration. The drawback of this approach is that the speech must first be segmented before the neural network can evaluate the segments.

The TDNN (Waibel et al 1989) represented an early attempt at segment-level training, as its output units were designed to integrate partial evidence from the whole duration of a phoneme, so that the network was purportedly trained at the phoneme level rather than at the frame level. However, the TDNN's input window assumed a constant width of 15 frames for all phonemes, so it did not truly operate at the segment level; and this architecture was only applied to phoneme recognition, not word recognition.

Austin et al (1992) at BBN explored true segment-level training for large vocabulary continuous speech recognition. A *Segmental Neural Network* (SNN) was trained to classify phonemes from variable-duration segments of speech; the variable-duration segments were linearly downsampled to a uniform width of five frames for the SNN. All phonemic segmentations were provided by a state-of-the-art HMM system. During training, the SNN was taught to correctly classify each segment of each utterance. During testing, the SNN was given the segmentations of the N-best sentence hypotheses from the HMM; the SNN produced a composite score for each sentence (the product of the scores and the duration probabilities[2] of all segments), and these SNN scores and HMM scores were combined to identify the single best sentence. This system achieved 11.6% word error on the RM database. Later, performance improved to 9.0% error when the SNN was also trained negatively

1. In this HMM, output symbols were emitted during transitions rather than in states, so there was actually one output unit per transition rather than per state.
2. Duration probabilities were provided by a smoothed histogram over all durations obtained from the training data.

on incorrect segments from N-best sentence hypotheses, thus preparing the system for the kinds of confusions that it was likely to encounter in N-best lists during testing.

4.3.4. Word Level Training

A natural extension to segment-level training is word-level training, in which a neural network receives input from an entire word, and is directly trained to optimize word classification accuracy. Word level training is appealing because it brings the training criterion still closer to the ultimate testing criterion of sentence recognition accuracy. Unfortunately the extension is nontrivial, because in contrast to a simple phoneme, a word cannot be adequately modeled by a single state, but requires a sequence of states; and the activations of these states cannot be simply summed over time as in a TDNN, but must first be segmented by a dynamic time warping procedure (DTW), identifying which states apply to which frames. Thus, word-level training requires that DTW be embedded into a neural network.

This was first achieved by Sakoe et al (1989), in an architecture called the *Dynamic programming Neural Network* (DNN). The DNN is a network in which the hidden units represent states, and the output units represent words. For each word unit, an alignment path between its states and the inputs is established by DTW, and the output unit integrates the activations of the hidden units (states) along the alignment path. The network is trained to output 1 for the correct word unit, and 0 for all incorrect word units. The DTW alignment path may be static (established before training begins) or dynamic (reestablished during each iteration of training); static alignment is obviously more efficient, but dynamic alignment was shown to give better results. The DNN was applied to a Japanese database of isolated digits, and achieved 99.3% word accuracy, outperforming pure DTW (98.9%).

Haffner (1991) similarly incorporated DTW into the high-performance TDNN architecture, yielding the *Multi-State Time Delay Neural Network* (MS-TDNN), as illustrated in Figure 4.6. In contrast to Sakoe's system, the MS-TDNN has an extra hidden layer and a hierarchy of time delays, so that it may form more powerful feature detectors; and its DTW path accumulates one score per frame rather than one score per state, so it is more easily extended to continuous speech recognition (Ney 1984). The MS-TDNN was applied to a database of spoken letters, and achieved an average of 93.6% word accuracy, compared to 90.0% for Sphinx[1]. The MS-TDNN benefitted from some novel techniques, including "transition states" between adjacent phonemic states (e.g., `B-IY` between the `B` and `IY` states, set to a linear combination of the activations of `B` and `IY`), and specially trained "boundary detection units" (BDU), which allowed word transitions only when the BDU activation exceeded a threshold value.

Hild and Waibel (1993) improved on Haffner's MS-TDNN, achieving 94.8% word accuracy on the same database of spoken letters, or 92.0% on the Resource Management spell mode database. Their improvements included (a) free alignment across word boundaries, i.e., using DTW on a segment of speech wider than the word to identify the word's boundaries dynamically during training; (b) word duration modeling, i.e., penalizing words by add-

1. In this comparison, Sphinx also had the advantage of using context-dependent phoneme models, while the MS-TDNN used context-independent models.

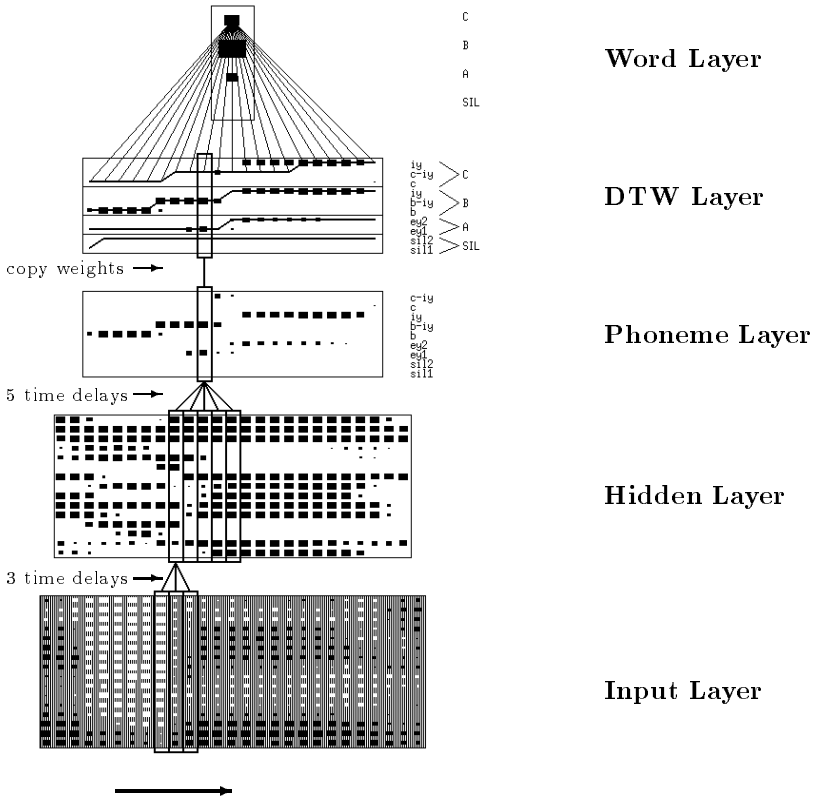

Figure 4.6: MS-TDNN recognizing the word "B". Only the activations for the words "SIL", "A", "B", and "C" are shown. (From Hild & Waibel, 1993).

ing the logarithm of their duration probabilities, derived from a histogram and scaled by a factor that balances insertions and deletions; and (c) sentence level training, i.e., training positively on the correct alignment path and training negatively on incorrect parts of an alignment path that is obtained by testing.

Tebelskis (1993) applied the MS-TDNN to large vocabulary continuous speech recognition. This work is detailed later in this thesis.

4.3.5. Global Optimization

The trend in NN-HMM hybrids has been towards global optimization of system parameters, i.e., relaxing the rigidities in a system so its performance is less handicapped by false assumptions. Segment-level training and word-level training are two important steps towards global optimization, as they bypass the rigid assumption that frame accuracy is correlated with word accuracy, making the training criterion more consistent with the testing criterion.

Another step towards global optimization, pursued by Bengio et al (1992), is the joint optimization of the input representation with the rest of the system. Bengio proposed a NN-HMM hybrid in which the speech frames are produced by a combination of signal analysis

and neural networks; the speech frames then serve as inputs for an ordinary HMM. The neural networks are trained to produce increasingly useful speech frames, by backpropagating an error gradient that derives from the HMM's own optimization criterion, so that the neural networks and the HMM are optimized simultaneously. This technique was evaluated on the task of speaker independent plosive recognition, i.e., distinguishing between the phonemes /b,d,g,p,t,k,dx,other/. When the HMM was trained separately from the neural networks, recognition accuracy was only 75%; but when it was trained with global optimization, recognition accuracy jumped to 86%.

4.3.6. Context Dependence

It is well known that the accuracy of an HMM improves with the context sensitivity of its acoustic models. In particular, context dependent models (such as triphones) perform better than context independent models (such as phonemes). This has led researchers to try to improve the accuracy of hybrid NN-HMM systems by likewise making them more context sensitive. Four ways to achieve this are illustrated in Figure 4.7.

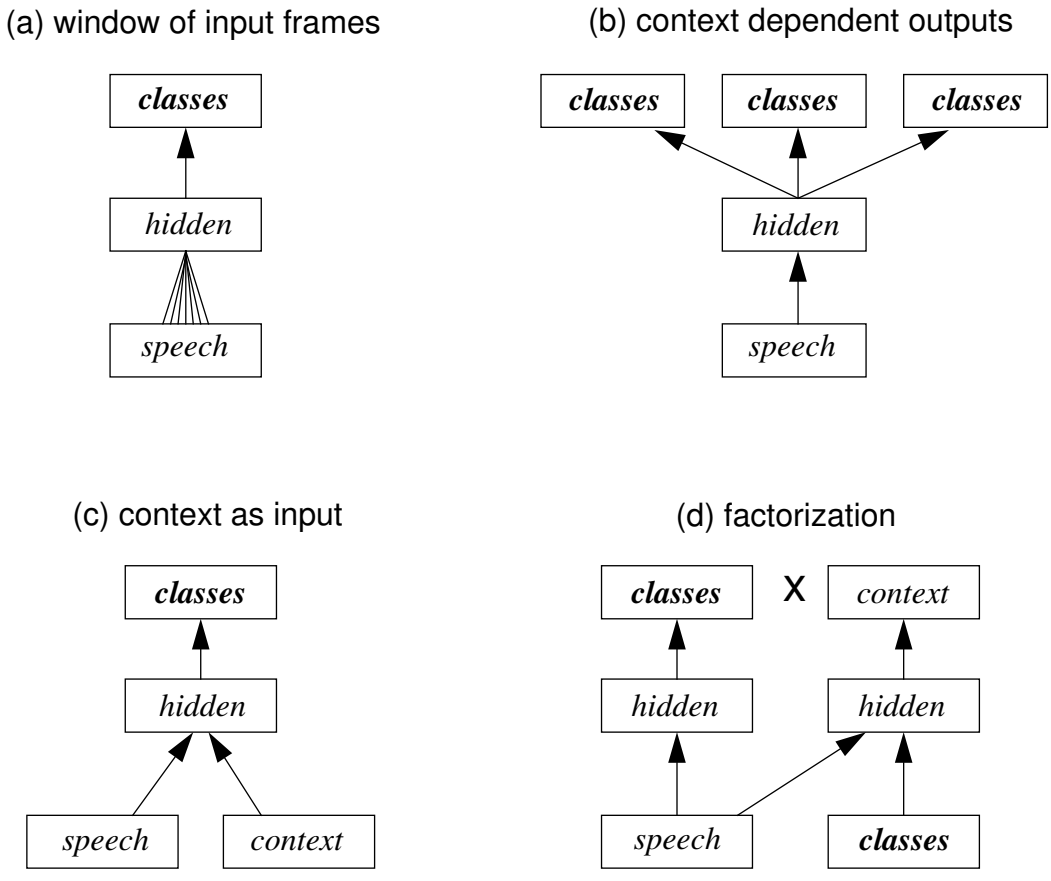

Figure 4.7: Four approaches to context dependent modeling.

The first technique is simply to provide a window of speech frames, rather than a single frame, as input to the network. The arbitrary width of the input window is constrained only by computational requirements and the diminishing relevance of distant frames. This technique is so trivial and useful for a neural network that it is used in virtually all NN-HMM hybrids; it can also be used in combination with the remaining techniques in this section. By contrast, in a standard HMM, the Independence Assumption prevents the system from taking advantage of neighboring frames directly. The only way an HMM can exploit the correlation between neighboring frames is by artificially absorbing them into the current frame (e.g., by defining multiple simultaneous streams of data to impart the frames and/or their deltas, or by using LDA to transform these streams into a single stream).

A window of input frames provides context sensitivity, but not context dependence. Context dependence implies that there is a separate model for each context, e.g., a model for /A/ when embedded in the context "kab", a separate model for /A/ when embedded in the context "tap", etc. The following techniques support true context dependent modeling in NN-HMM hybrids.

In technique (b), the most naive approach, there is a separate output unit for each context-dependent model. For example, if there are 50 phonemes, then it will require 50x50 = 2500 outputs in order to model diphones (phonemes in the context of their immediate neighbor), or 50x50x50 = 125000 outputs to model triphones (phonemes in the context of both their left and right neighbor). An obvious problem with this approach, shared by analogous HMMs, is that there is unlikely to be enough training data to adequately train all of the parameters of the system. Consequently, this approach has rarely been used in practice.

A more economical approach (c) is to use a single network that accepts a description of the context as part of its input, as suggested by Petek et al (1991). Left-phoneme context dependence, for example, could be implemented by a boolean localist representation of the left phoneme; or, more compactly, by a binary encoding of its linguistic features, or by its principal components discovered automatically by an encoder network. Note that in order to model triphones instead of diphones, we only need to double the number of context units, rather than using 50 times as many models. Training is efficient because the full context is available in the training sentences; however, testing may require many forward passes with different contextual inputs, because the context is not always known. Petek showed that

Figure 4.8: Contextual inputs. Left: standard implementation. Right: efficient implementation.

these forward passes can be made more efficient by heuristically splitting the hidden layer as shown in Figure 4.8, such that the speech and the context feed into independent parts, and each context effectively contributes a different bias to the output units; after training is complete, these contextual output biases can be precomputed, reducing the family of forward passes to a family of output sigmoid computations. Contextual inputs helped to increase the absolute word accuracy of Petek's system from 60% to 72%.

Bourlard et al (1992) proposed a fourth approach to context dependence, based on factorization (d). When a neural network is trained as a phoneme classifier, it estimates $P(q|x)$, where q is the phoneme class and x is the speech input. To introduce context dependence, we would like to estimate $P(q,c|x)$, where c is the phonetic context. This can be decomposed as follows:

$$P(q, c|x) = P(q|x) \cdot P(c|q,x) \tag{56}$$

This says that the context dependent probability is equal to the product of two terms: $P(q|x)$ which is the output activation of a standard network, and $P(c|q,x)$ which is the output activation of an auxiliary network whose inputs are speech as well as the current phoneme class, and whose outputs range over the contextual phoneme classes, as illustrated in Figure 4.7(d). The resulting context dependent posterior can then be converted to a likelihood by Bayes Rule:

$$P(x|q,c) = \frac{P(q, c|x) \cdot P(x)}{P(q, c)} \tag{57}$$

where $P(x)$ can be ignored during recognition because it's a constant in each frame, and the prior $P(q,c)$ can be evaluated directly from the training set.

This factorization approach can easily be extended to triphone modeling. For triphones, we want to estimate $P(q,c_l,c_r|x)$, where c_l is the left phonetic context and c_r is right phonetic context. This can be decomposed as follows:

$$P(q, c_l, c_r|x) = P(q|x) \cdot P(c_l|q,x) \cdot P(c_r|c_l,q,x) \tag{58}$$

Similarly,

$$P(q, c_l,c_r) = P(q) \cdot P(c_l|q) \cdot P(c_r|c_l,q) \tag{59}$$

These six terms can be estimated by neural networks whose inputs and outputs correspond to each i and o in $P(o|i)$; in fact some of the terms in Equation (59) are so simple that they can be evaluated directly from the training data. The posterior in Equation (58) can be converted to a likelihood by Bayes Rule:

$$P(x|q,c_l,c_r) = \frac{P(q, c_l,c_r|x) \cdot P(x)}{P(q, c_l,c_r)} \tag{60}$$

$$= \frac{P(q|x) \cdot P(c_l|q,x) \cdot P(c_r|c_l,q,x)}{P(q) \cdot P(c_l|q) \cdot P(c_r|c_l,q)} \cdot P(x) \quad (61)$$

where $P(x)$ can again be ignored during recognition, and the other six terms can be taken from the outputs of the six neural networks. This likelihood can be used for Viterbi alignment.

As in approach (c), a family of forward passes during recognition can be reduced to a family of output sigmoid computations, by splitting the hidden layer and caching the effective output biases from the contextual inputs. Preliminary experiments showed that splitting the hidden layer in this way did not degrade the accuracy of a network, and triphone models were rendered only 2-3 times slower than monophone models.

4.3.7. Speaker Independence

Experience with HMMs has shown that speaker independent systems typically make 2-3 times as many errors as speaker dependent systems (Lee 1988), simply because there is greater variability between speakers than within a single speaker. HMMs typically deal with this problem by merely increasing the number of context-dependent models, in the hope of better covering the variabilities between speakers.

NN-HMM hybrids suffer from a similar gap in performance between speaker dependence and speaker independence. For example, Schmidbauer and Tebelskis (1992), using an LVQ-based hybrid, obtained an average of 14% error on speaker-dependent data, versus 32% error when the same network was applied to speaker-independent data. Several techniques aimed at closing this gap have been developed for NN-HMM hybrids. Figure 4.9 illustrates the baseline approach of training a standard network on the data from all speakers (panel a), followed by three improvements upon this (b,c,d).

The first improvement, shown as technique (b), is a **mixture of speaker-dependent models**, resembling the Mixture of Experts paradigm promoted by Jacobs et al (1991). In this approach, several networks are trained independently on data from different speakers, while a "speaker ID" network is trained to identify the corresponding speaker; during recognition, speech is presented to all networks in parallel, and the outputs of the speaker ID network specify a linear combination of the speaker-dependent networks, to yield an overall result. This approach makes it easier to classify phones correctly, because it separates and hence reduces the overlap of distributions that come from different speakers. It also yields multi-speaker accuracy[1] close to speaker-dependent accuracy, the only source of degradation being imperfect speaker identification. Among the researchers who have studied this approach:

- Hampshire and Waibel (1990) first used this approach in their *Meta-Pi network*, which consisted of six speaker-dependent TDNNs plus a speaker ID network containing one unit per TDNN, all trained by backpropagation. This network obtained 98.4% phoneme accuracy in multi-speaker mode, significantly outperforming a

1. "Multi-speaker" evaluation means testing on speakers who were in the training set.

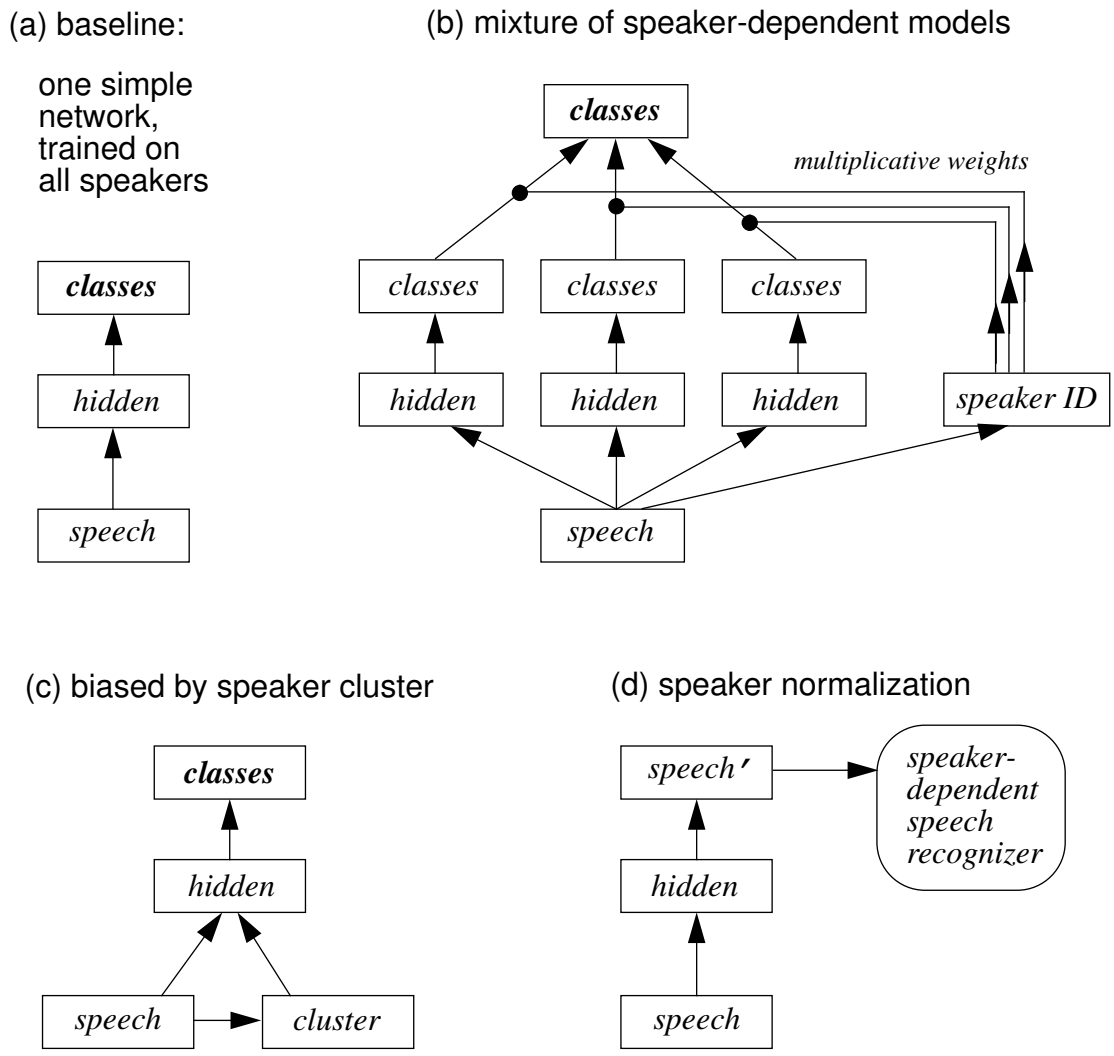

Figure 4.9: Four approaches to speaker independent modeling.

baseline TDNN which obtained only 95.9% accuracy. Remarkably, one of the speakers (MHT) obtained 99.8% phoneme accuracy, even though the speaker ID network failed to recognize him and thus ignored the outputs of MHT's own TDNN network, because the system had formed a robust linear combination of other speakers whom he resembled.

- Kubala and Schwartz (1991) adapted this approach to a standard HMM system, mixing their speaker-dependent HMMs with fixed weights instead of a speaker ID network. They found that only 12 speaker-dependent HMMs were needed in order to attain the same word recognition accuracy as a baseline system trained on 109 speakers (using a comparable amount of total data in each case). Because of this, and because it's cheaper to collect a large amount of data from a few speakers than

to collect a small amount of data from many speakers, Kubala and Schwartz concluded that this technique is also valuable for reducing the cost of data collection.

- Schmidbauer and Tebelskis (1992) incorporated this approach into an LVQ-HMM hybrid for continuous speech recognition. Four speaker-*biased* phoneme models (for pooled males, pooled females, and two individuals) were mixed using a correspondingly generalized speaker ID network, whose activations for the 40 separate phonemes were established using five "rapid adaptation" sentences. The rapid adaptation bought only a small improvement over speaker-independent results (59% vs. 55% word accuracy), perhaps because there were so few speaker-biased models in the system. Long-term adaptation, in which all system parameters received additional training on correctly recognized test sentences, resulted in a greater improvement (to 73%), although still falling short of speaker-dependent accuracy (82%).
- Hild and Waibel (1993) performed a battery of experiments with MS-TDNNs on spelled letter recognition, to determine the best level of speaker and parameter specificity for their networks, as well as the best way to mix the networks together. They found that segregating the speakers is always better than pooling everyone together, although some degree of parameter sharing between the segregated networks is often helpful (given limited training data). In particular, it was often best to mix only their lower layers, and to use shared structure at higher layers. They also found that mixing the networks according to the results of a brief adaptation phase (as in Schmidbauer and Tebelskis) is generally more effective than using an instantaneous speaker ID network, although the latter technique gives comparable results in multi-speaker testing. Applying their best techniques to the speaker-independent Resource Management spell mode database, they obtained 92.0% word accuracy, outperforming Sphinx (90.4%).

Another way to improve speaker-independent accuracy is to **bias the network** using extra inputs that characterize the speaker, as shown in Figure 4.9(c). The extra inputs are determined automatically from the input speech, hence they represent some sort of cluster to which the speaker belongs. Like the Mixture of Experts approach, this technique improves phoneme classification accuracy by separating the distributions of different speakers, reducing their overlap and hence their confusability. It has the additional advantage of adapting very quickly to a new speaker's voice, typically requiring only a few words rather than several whole sentences. Among the researchers who have studied this approach:

- Witbrock and Haffner (1992) developed the *Speaker Voice Code network* (SVC-net), a system that learns to quickly identify where a speaker's voice lies in a space of possible voices. An SVC is a 2 unit code, derived as the bottleneck of an encoder network that is trained to reproduce a speaker's complete set of phoneme pronunciation codes (PPCs), each of which is a 3-unit code that was likewise derived as the bottleneck of an encoder network that was trained to reproduce the acoustic patterns associated with that particular phoneme. The SVC code varied considerably between speakers, yet proved remarkably stable for any given speaker, regardless of the phonemes that were available for its estimation in only a few words of speech. When the SVC code was provided as an extra input to an

MS-TDNN, the word accuracy on a digit recognition task improved from 1.10% error to 0.99% error.

- Konig and Morgan (1993) experimented with the *Speaker Cluster Neural Network* (SCNN), a continuous speech recognizer in which an MLP's inputs were supplemented by a small number of binary units describing the speaker cluster. When two such inputs were used, representing the speaker's gender (as determined with 98.3% accuracy by a neural network that had received supervised training), performance on the Resource Management database improved from 10.6% error to 10.2% error. Alternatively, when speakers were clustered in an unsupervised fashion, by applying k-means clustering to the acoustic centroids of each speaker (for k = 2 through 5 clusters), performance improved to an intermediate level of 10.4% error.

A final way to improve speaker-independent accuracy is through **speaker normalization**, as shown in Figure 4.9(d). In this approach, one speaker is designated as the reference speaker, and a speaker-dependent system is trained to high accuracy on his voice; then, in order to recognize speech from a new speaker (say, a female), her acoustic frames are mapped by a neural network into corresponding frames in the reference speaker's voice, which can then be fed into the speaker-dependent system.

- Huang (1992a) explored speaker normalization, using a conventional HMM for speaker-dependent recognition (achieving 1.4% word error on the reference speaker), and a simple MLP for nonlinear frame normalization. This normalization network was trained on 40 adaptation sentences for each new speaker, using DTW to establish the correspondence between input frames (from the new speaker) and output frames (for the reference speaker). The system was evaluated on the speaker-dependent portion of the Resource Management database; impressively, speaker normalization reduced the cross-speaker error rate from 41.9% error to 6.8% error. The error rate was further reduced to 5.0% by using eight codeword-dependent neural networks instead of a single monolithic network, as the task of each network was considerably simplified. This final error rate is comparable to the error rate of speaker-independent systems on this database; hence Huang concluded that speaker normalization can be useful in situations where large amounts of training data are available only for one speaker and you want to recognize other people's speech.

4.3.8. Word Spotting

Continuous speech recognition normally assumes that every spoken word should be correctly recognized. However, there are some applications where in fact only very few vocabulary words (called *keywords*) carry any significance, and the rest of an utterance can be ignored. For example, a system might prompt the user with a question, and then only listen for the words "yes" or "no", which may be embedded within a long response. For such applications, a *word spotter*, which listens for and flags only these keywords, may be more useful than a full-blown continuous speech recognition system. Several researchers have

recently designed word spotting systems that incorporate both neural networks and HMMs. Among these systems, there have been two basic strategies for deploying a neural network:

1. A neural network may serve as a secondary system that reevaluates the putative hits identified by a primary HMM system. In this case, the network's architecture can be rather simple, because an already-detected keyword candidate can easily be normalized to a fixed duration for the network's input.

2. A neural network may serve as the primary word spotter. In this case, the network's architecture must be more complex, because it must automatically warp the utterance while it scans for keywords.

David Morgan et al (1991) explored the first strategy, using a primary word spotter that was based on DTW rather than HMMs. When this system detected a keyword candidate, its speech frames were converted to a fixed-length representation (using either a Fourier transform, a linear compression of the speech frames, a network-generated compression, or a combination of these); and then this fixed-length representation was reevaluated by an appropriately trained neural network (either an RCE network[1], a probabilistic RCE network, or a modularized hierarchy of these), so that the network could decide whether to reject the candidate as a "false alarm". This system was evaluated on the "Stonehenge X" database. One rather arcane combination of the above techniques eliminated 72% of the false alarms generated by the primary system, while only rejecting 2% of the true keywords (i.e., word spotting accuracy declined from 80% to 78%).

Zeppenfeld and Waibel (1992,1993) explored the second strategy, using an MS-TDNN as a primary word spotter. This system represented keywords with unlabeled state models rather than shared phoneme models, due to the coarseness of the database. The MS-TDNN produced a score for each keyword in every frame, derived from the keyword's best DTW score in a range of frames beginning in the current frame. The system was first bootstrapped with state-level training on a forced linear alignment within each keyword, and then trained with backpropagation from the word level; positive and negative training were carefully balanced in both phases. It achieved a Figure of Merit[2] of 82.5% on the Road Rally database. Subsequent improvements — which included adding noise to improve generalization, subtracting spectral averages to normalize different databases, using duration constraints, grouping and balancing the keywords by their frequency of occurrence, extending short keywords into their nearby context, and modeling variant suffixes — contributed to a Figure of Merit of 72.2% on the official Stonehenge database, or 50.9% on the official Switchboard database.

Lippmann and Singer (1993) explored both of the above strategies. First, they used a high-performance tied-mixture HMM as a primary word spotter, and a simple MLP as a secondary tester. Candidate keywords from the primary system were linearly normalized to a fixed width for the neural network. The network reduced the false alarm rate by 16.4% on the Stonehenge database. This network apparently suffered from a poverty of training data;

1. Restricted Coloumb Energy network. RCE is a trademark of Nestor, Inc.
2. Figure of Merit summarizes a tradeoff between detection rate and false alarm rate. It is computed as the average detection rate for system configurations that achieve between 0 and 10 false alarms per keyword per hour.

attempts were made to augment the training set with false alarms obtained from an independent database, but this failed to improve the system's performance because the databases were too different, and hence too easily discriminable. The second strategy was then explored, using a primary network closely resembling Zeppenfeld's MS-TDNN, except that the hidden layer used radial basis functions instead of sigmoidal units. This enabled new RBF units to be added dynamically, as their Gaussians could be automatically centered on false alarms that arose in training, to simplify the goal of avoiding such mistakes in the future.

4.4. Summary

The field of speech recognition has seen tremendous activity in recent years. Hidden Markov Models still dominate the field, but many researchers have begun to explore ways in which neural networks can enhance the accuracy of HMM-based systems. Researchers into NN-HMM hybrids have explored many techniques (e.g., frame level training, segment level training, word level training, global optimization), many issues (e.g., temporal modeling, parameter sharing, context dependence, speaker independence), and many tasks (e.g., isolated word recognition, continuous speech recognition, word spotting). These explorations have especially proliferated since 1990, when this thesis was proposed, hence it is not surprising that there is a great deal of overlap between this thesis and concurrent developments in the field. The remainder of this thesis will present the results of my own research in the area of NN-HMM hybrids.