

A System for Offline Cursive Handwritten Word Recognition

by

Marlon Abayan

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 8, 1997

© 1997 Marlon Abayan. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author
Electrical Engineering and Computer Science
July 22, 1997

Certified by
Professor Michael I. Jordan
Department of Brain and Cognitive Science
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Theses



A System for Offline Cursive Handwriting Recognition

by

Marlon Abayan

Submitted to the

Department of Electrical Engineering and Computer Science

July 22, 1997

in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Cursive handwriting recognition is a difficult problem because of large variations in handwritten words as well as overlaps and interconnections between neighboring characters. In this thesis we introduce a new approach to this problem, called Hidden Markov Model with Multiple Observation Sequences (HMMOS). A preprocessor extracts each word from a scanned-in document image and divides it into segments. A Neural Network (NN) classifier then finds the likelihoods of each possible character class given the segments and combinations of segments. These likelihoods, along with statistics computed from a lexicon, are used as input to a dynamic programming algorithm which recognizes the entire word. The dynamic programming algorithm can be viewed as a modified Viterbi algorithm for a Hidden Markov Model (HMM). Three types of Neural Networks are tried: a recurrent network, a Multilayer Perceptron (MLP), and a Hierarchical Mixture of Experts (HME) [Jordan & Jacobs 1994]. As an extension, the Viterbi algorithm for the HMM is implemented for a multiprocessor environment to speed up recognition.

Thesis Supervisor: Michael Jordan

Title: Professor, Department of Brain and Cognitive Science

Acknowledgments

This thesis would have been impossible without the help of my mentor at IBM Almaden Research Center, Dr. Jianchiang Mao. With his deep knowledge and keen insight about the topic, he guided me and served as an inspiration throughout the project. He taught me a lot about the process of doing research. And most importantly, he encouraged me to keep my focus and work hard every day.

I salute my thesis advisor, Prof. Michael Jordan, for his patience, input, and support.

Many colleagues at IBM made my task easier. Prasun Sinha gave very valuable input on my thesis drafts, and helped me at various stages in the project. My manager, K. M. Mohiuddin, was kind and always showed interest in my well-being and progress. An office-mate, Mary Mandl was a joy to have around. Others who gave encouragement and a helping hand are Sandeep Gopisetty, Raymond Lorrie, Andras Kornai, Tom Truong, and Peter DeSouza. Special thanks to Andras and Mao for giving me rides from work when I missed the shuttle.

I thank a friend from MIT, Beethoven Cheng, for helping me edit my drafts for grammar and style.

I am grateful to Allen Luniewsky, the 6-A liaison between IBM and MIT, and Prof. Jerome Saltzer, the 6-A advisor for IBM, who made sure I was plugged in and ready to work on my thesis.

Hurrah to my cheering squad of friends: Tom and Jane Letterman, Luong Tran, Jay Lulla, Yee Chuan Koh, and Chung-hsiu Ma.

Finally, I thank my parents, Perfecto and Carmencita Abayan, and my sister Maureen, for always being there for me. I dedicate this work to them.

Table of Contents

Introduction 10

- Problem Definition 10
- Background Literature 10
- Hidden Markov Model (HMM) Framework 12
- Variable Sequence Length HMM 13
- HMM with Multiple Observation Sequences 16
- Stochastic Context Free Grammars and Inside-Outside Algorithm 17
- Organization of this thesis 20

System Overview 22

- Architecture 22
- Training 23
- Oversegmentation 23
- Truthing 24
- Feature Extraction 24
- Neural Network character recognizer 24
- Transition Probabilities 25
- Recognition 25
- Software modules developed in this thesis 26

HMM with Multiple Observation Sequences 28

- HMM Framework 28
- Viterbi algorithm with multiple observation sequences. 31
- Original Viterbi Algorithm 31
- Modified Viterbi 32
- Correctness of Modified Viterbi Algorithm 36
- Asymptotic Performance 36
- Results. 38

Probability Estimation 42

- Overview 42
- Initial and Transition Probabilities 42
- Observation Likelihood Estimation (Character recognition) 42
- Multilayer Perceptron (MLP) 43
- Recurrent Neural Net 45
- Hierarchical Mixture of Experts 47
- Results 49

Rejection of bad samples to increase recognition rates 52

- Problem Definition 52
- Multilayer Perceptron Approach 54
- Features used 54
- Results 55

Improving recognition by verification and correction 58

- Introduction 58
- A Dynamic Programming Algorithm 59
- Correctness 65

Asymptotic Performance	65
Results	65
Extension: Parallel Implementation	68
Motivation and Problem Definition	68
The Model	69
Coarse grain parallelism	71
Fine grain parallelism	72
Forward-phase Viterbi	72
Pointer jumping algorithm for finding maximum	73
MLP Evaluations	74
Results	74
Conclusions and Future Work	76
Conclusions	76
Future Work	76
Compare the three neural classifiers.	76
Stochastic Context Free Grammars	77
References	78

List of Figures

Overall System Specification	10
Two-step approach	11
Sample oversegmented word	13
Modified Viterbi	14
Software Infrastructure	22
Oversegmentation examples.	23
Hidden Markov Model	28
Input/Output Relation for MLP used as Neural Classifier	30
Multilayer Perceptron	43
Node at layer h and row j	44
Right and middle parts of “w” vs. “u” when context is present.	46
A 2-level HME. To form a deeper tree, each expert is expanded recursively into a gating network, a nonterminal node, and a set of subexperts.	49
Two gaussians representing good and bad word images.	53
Error rate v.s Rejection rate for the verification algorithm	56
An oversegmented image of “word”.	60
Dynamic Programming Table y for “word”.	61
HMM with a π state and 26 character states	70
Coarse grain parallelism by alternating MLP evaluations	71
$n = 6$ nodes or states, $p = 3$ processors or processes	72
Speedup (y-axis) vs. number of HMM state processors (x-axis)	74

Chapter 1

Introduction

1.1 Problem Definition

This thesis proposes a solution to the off-line cursive handwritten word recognition problem. Handwritten word recognition (HWR) is the problem of interpreting a word image by assigning to it a particular sequence of characters called a “word”. The subproblems range from recognizing block-printed and disconnected characters to identifying cursively handwritten words. HWR can be done on-line or off-line, as differentiated in [Seiler, Schenkel, & Eggimann 1996]. When the recognition is done on-line, a user typically writes a word onto a computer interface. Temporal information, as well as the word image, is available to the recognition system. When it is done off-line, however, only the handwritten word image is available. Hence, off-line handwriting recognition is more difficult.

Examples of real world applications of off-line cursive HWR include personal check processing, mail sorting, and automatic data entry of business forms.

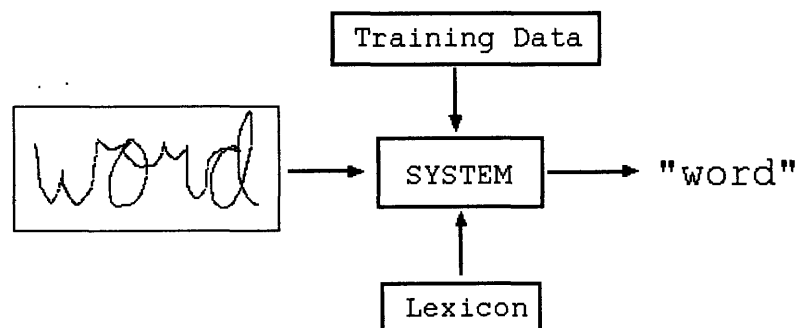


Figure 1.1: Overall System Specification

1.2 Background Literature

The strategies for off-line cursive handwritten word recognition can be roughly classified

into three categories. In the first category, the word is segmented into several characters and character recognition techniques are applied to each segment [Casey and Nagy, 1982]. This method heavily depends on the accuracy of the segmentation points found. Unfortunately, such an accurate segmentation technique is not yet available. In the second category, whole words are recognized without doing any kind of formal segmentation [Farag 1979] [Hull & Srihari 1986]. The global shape is analyzed to hypothesize the words. This is effective for small lexicons such as in check processing. The difference between the global shapes of “one”, “two”, and “dollar” can easily be captured by a simple classifier. As the lexicon grows, however, the differences between words become more minute and the classifier needs to be much more complex. Usually this strategy does not work for medium-sized and large lexicons. The third category is a compromise solution between these two schemes. It does a loose segmentation to find a number of potential segmentation points [Bozinovic & Srihari 1986] [He, Chen, Kundu 1996]. The final segmentation and word identity are determined later in the recognition stage with the help of a lexicon. Figure 1.2 illustrates this two step process.

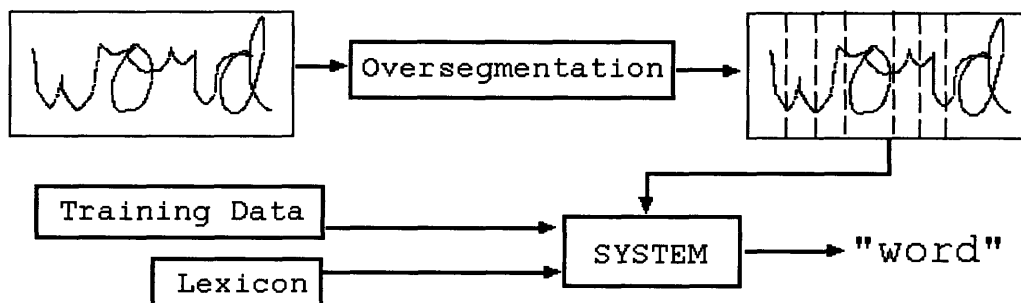


Figure 1.2: Two-step approach

The new Hidden Markov Model (HMM) approach to HWR described in this paper belongs to the third category.

1.3 Hidden Markov Model (HMM) Framework

Like most real world processes, handwriting produces observable outputs which can be characterized as sequential signals. These signals can be successfully coded by a stochastic model such as a Hidden Markov Model. It is quite natural to model handwriting as an embedded Markov chain in an HMM. For example, a word can be regarded as a Markov chain of individual characters.

An HMM has an underlying Markov process that is “hidden” or not directly observable, i.e. it does not produce the observation sequences directly. Another set of stochastic processes produces the sequence of observed symbols and stochastically relate the symbols to the “hidden” Markov states. They are called observation probability distributions.

HMMs are often called doubly stochastic processes because there are two sets of processes. They are more powerful than Markov models which allow one less degree of freedom. In handwriting, the writer translates the language to the written text with added ambiguities, e.g., variation in writing style, ambiguity caused by connected and overlapping letters, etc. These ambiguities can be accommodated by the observation probability distributions.

A particular difficulty in the use of HMMs for HWR is the following problem. The potential segmentation points found in the loose segmentation phase are a superset of the points that would partition the image into the underlying characters. It is therefore necessary to identify which subsegments, when joined, form a single character. [Chen et. al. 1995] successfully applied a Variable Duration Hidden Markov Model (VDHMM) to solve this problem. Each state is actually a “small HMM” which itself has four states to account for up to four segments. The small HMMs embody the variability in the duration (the number of segments) of each state. In order to run a Viterbi algorithm for them, there is a need to estimate duration statistics.

1.4 Variable Sequence Length HMM

[He et. al. 1996] discusses a similar approach which does not involve computing duration statistics, called Variable Sequence Length HMM (VSLHMM). Instead, it explores the possible combinations of character subsegments using a modified dynamic programming Viterbi algorithm. However, it is not a pure dynamic programming algorithm because it has a greedy component which arises from the following assumption:

“If one of the 4 block images consisting of either 1 or 2 or 3 or 4 consecutive segments (starting from the same segment) is an actual character image of a [character state], then if we compare the four images with that letter in the feature space, the actual character image always has the best match.”

The characters are oversegmented into at most four segments. However, for consistency with our work, assume that the oversegmented characters are composed of only up to three segments.

The next section explains how the assumption can break and the consequences of that happening. This section steps through how the algorithm works on a sample image (Figure 1.3).

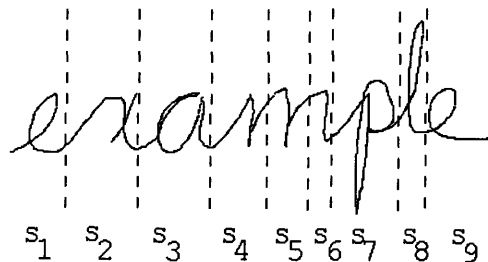


Figure 1.3: Sample oversegmented word

Figure 1.4 shows a dynamic programming table used to find an optimal word, i.e. state sequence, for Figure 1.3. Each row corresponds to a character state; each column, $k=1, 2, \dots, 8$, corresponds to the k^{th} successive letter in the state sequence. The entries of the table

consist of a pair: the starting segment number, and the number of segments used up by the character. Two paths through the table are shown, one dashed and the other solid. The solid is the correct state sequence: 'e', 'x', 'a', 'm', 'p', 'l', 'e'. It starts with an 'e' because the first column entry along the path is on the 'e' row. The entry, <1,1>, means that the starting segment is 1 and 'e' takes up exactly 1 segment. The second state is 'x' because the next entry along the path is on row 'x'. Similarly, the third state is 'a'. Column 4 shows that the fourth state is 'm'. This character starts at segment 4 and takes up 3 segments (see Figure 1.3). The remaining states are 'p', 'l', and 'e', each taking up one segment.

The dashed path is 'v', 'a', 'v', 'v', 'p', 'b'. It has six states and therefore uses up six columns. The first state, 'v', takes up the first two segments. The second state, 'a', uses up only the third segment, and so on along the path. Notice that both paths cross at row 'p' and column 5.

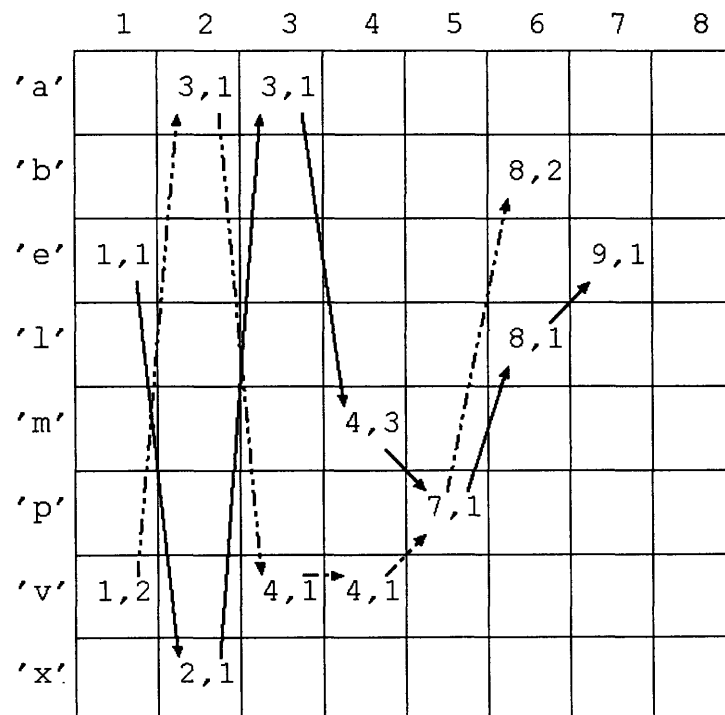


Figure 1.4: Modified Viterbi

The algorithm explores many possible paths through the table and the one with the highest probability is chosen. The state sequence $Q = \langle \pi, q_1, q_2, \dots, q_T \rangle$, implied by a path through the table, is mapped to a probability by the expression below.

$$a_{\pi, q_1} P(o_1 | q_1) a_{q_1, q_2} \dots a_{q_{T-1}, q_T} P(o_T | q_T)$$

Each o_i is the combination of adjacent segments derived from the table entry along the path at column i . Let us define any combination of up to three adjacent segments to be a *chunk*. Each o_i is therefore a chunk.

The algorithm starts by filling in the first column. All the rows (states) have to begin at the first segment so the first element of each table entry on the first column is always 1. The second element is determined by finding the number of segments which will optimize the probability $P(o|q)$ where q is the state and o is the chunk composed of the segments. This step arises directly from the assumption mentioned above. The number of segments “consumed” by a state at a given column is greedily set to the one that maximizes the probability at that point in the algorithm.

The second column is filled by finding the state in the first column, i^* , that maximizes the path probability up to that point,

$$i^* = \underset{i}{\operatorname{argmax}} \quad a_{\pi, q_i} P(o_1 | q_i) a_{q_i, q_j} P(o_2 | q_j)$$

where i is a previous state and j is the current state. Back-pointers to i^* for each j are also stored. The chunks o_1 and o_2 are selected by maximizing $P(o_1 | q_i)$ and $P(o_2 | q_j)$, respectively, for various combinations of segments. The algorithm proceeds in the same fashion for subsequent columns until all paths have used up all the segments. Then, the path with the highest probability is selected.

1.5 HMM with Multiple Observation Sequences

This thesis introduces an HMM architecture coined *HMM with Multiple Observation Sequences* (HMMMOS). It is based on VSLHMM, but relaxes the assumption stated in section 1.4 because it frequently breaks in practice and problems arise.

For example, if segment 2 in Figure 1.3 is always merged with segment 1 for all the states with high probability in column 1, it will never be recognized as a separate character 'x'. That is, if all table entries in the first column are of the form $\langle 1,2 \rangle$ or $\langle 1,3 \rangle$ then no path in the table will ever have a separate segment 2. For state 'e' the assumption that the best chunk will have the highest probability is violated. Instead, the chunk composed of segments 1 and 2, or 1 and 2 and 3, have the highest probability.

Similarly, if segment 3 is merged with segment 4 for state 'a' we can never separate them. The resulting misalignment eats into the 'm' character and causes more errors down the line.

These problems are solved by HMMMOS by having a more general dynamic programming Viterbi algorithm which gets rid of the greedy component. The algorithm is discussed in detail in Chapter 3.

Although HMMs and HMMMOS are very similar in that they have Viterbi-like procedures for assigning probabilities to candidates, there are some major differences. An HMMMOS is not trained using the canonical HMM training algorithm [Rabiner 1989]. Instead, the transition probabilities are estimated directly from a lexicon and the observation probability distributions are provided by a neural classifier which is trained separately. Furthermore, the units of inputs, the word segments, do not generally correspond to a single state: groups of inputs do. This causes the Viterbi algorithm to be more complicated.

In order to characterize HMMMOS more explicitly than as an extended HMM, the next section compares it with Stochastic Context Free Grammars (SCFG), and the associated training and recognition algorithm, the Inside-Outside Algorithm.

1.6 Stochastic Context Free Grammars and Inside-Outside Algorithm

HMMMOS and SCFGs are both stochastic models for sequential data. Their main similarity is in the training and the fact that segments are recombined into chunks, which looks like combining nonterminals in a grammar production.

A basic formulation of stochastic context-free grammars is given by [Fu 1976]. A context-free language is one in which a grammar in the Chomsky normal form [Chomsky 1959] can be found. I.e., productions can only be of the form

$$\begin{aligned}
 & i \rightarrow jk \\
 & \text{or} \\
 & i \rightarrow m
 \end{aligned}$$

where the symbols i , j , and k are nonterminals and m is a terminal symbol. A stochastic context-free grammar is a grammar for a context-free language in which all productions are associated with a probability. The probabilities for the grammar rules for a context-free grammar G are stored in matrices A and B with elements

$$\begin{aligned}
 a[i, j, k] &= P(i \rightarrow jk|G) \\
 b[i, m] &= P(i \rightarrow m|G)
 \end{aligned}$$

representing the probability of the productions of the forms $i \rightarrow jk$ and $i \rightarrow m$, respectively. Therefore, the parameters stored in the A matrix represent the *hidden* process while the parameters stored in the B matrix represent the *observable* process.

The Inside-Outside algorithm [Lari & Young 1991] is used to train parameters from sample data and recognize test sentences given the grammar. The algorithm defines *inner* (e) and *outer* (f) probability distributions as follows,

$$e(s, t, i) = P(i \Rightarrow O(s), \dots, O(t) | G)$$

$$f(s, t, i) = P(S \Rightarrow O(1), \dots, O(s-1), i, O(t+1), \dots, O(T) | G)$$

The symbol “ \Rightarrow ” means that the symbol on the left hand side eventually expands into the right hand side using zero or more productions. S is the start state of the grammar. The algorithm proceeds by computing the inner probabilities in a bottom-up fashion and the outer-probabilities in a top-down fashion.

An HMMOS can be interpreted as an SCFG in the following way. All the states of the HMMOS, the characters, are nonterminals in the context-free grammar. The image segments are terminal symbols. State sequences (words and subwords) are also nonterminals. Words are formed by expanding word nonterminals into pairs of nonterminals in productions of the form $i \rightarrow jk$, until the expression is expressed purely in terms of state nonterminals. The state nonterminals are then expanded into productions involving the terminal symbols, the segments. Here are some examples of productions for the state ‘w’:

$$w \rightarrow w_L w_R \quad (1)$$

$$w_R \rightarrow \alpha_1 \quad (2)$$

$$w_L \rightarrow w_{L_L} w_{L_R} \quad (3)$$

$$w_L \rightarrow \alpha_2 \quad (4)$$

$$w \rightarrow \alpha_3 \quad (5)$$

$$w_{L_L} \rightarrow \alpha_4 \quad (6)$$

$$w_{L_R} \rightarrow \alpha_5 \quad (7)$$

The α_i are nonterminal symbols corresponding to specific word segments.

There are many ways a letter, in this case ‘w’, can be formed. Each way has a different probability, with some of them being equal to zero. In production (1) the character is bro-

ken up into two nonterminals. The left nonterminal can be a nonterminal segment (4) or a combination of adjacent nonterminal segments (3), (6), and (7). The right nonterminal (1) can only be a segment (2). This makes the grammar for 'w' left recursive, though it need not be. Finally, the character could be formed by a single segment (5). Each of (2), (4), (5), (6), (7) actually represent a very large number of productions because the α_i are arbitrary bitmaps. If the size and resolution of the bitmaps are unconstrained, they correspond to an infinite number of productions. The probabilities for this class of productions are in the matrix B discussed above.

Using our segmentation algorithm, the character 'w' would most often be characterized by (1) and (3) because 'w' has three ascending spikes. The character 'n', on the other hand, would most often be characterized by (1) and (4) because it has two descending spikes, and 'a' by (5) because it has a roundish self-contained shape.

The main difference between the HMMMOS and SCFG arises when probabilities are assigned to the productions. HMMMOS does not assign probabilities to (2), (4), (6), and (7). Instead, probabilities are assigned directly to (1) and (5) based on the combined bitmap formed from all the segments involved. Hence, in contrast to SCFG, the intra-state productions of the form $i \rightarrow jk$ are not independent of the productions of the form $i \rightarrow m$. The Viterbi algorithm for HMMMOS uses a bottom-up dynamic programming algorithm to consider different combinations of adjacent segments into chunks which form the right hand side of (1) and (5).

The transition probabilities in HMMMOS is the same as the production probabilities for contiguous characters in the context-free grammar. The observation probabilities are exactly the elements of matrix B. Matrix A in the SCFG is just the transition probabilities, along with the probabilities of productions of single characters comprised of chunks of

segments just described. Matrix B is the set of observation probabilities and a set of undefined and unused probabilities for productions of the form (2), (4), (6), and (7).

Training-wise, HMMMOS and SCFG are similar in that the A and B matrices are determined separately but use the same data. The exception, of course, is the probabilities in productions (1) and (3) in matrix A can only be determined after matrix B has been determined.

The HMMMOS Viterbi algorithm is similar to the Inside-Outside algorithm [Lari & Young 1991] in that it computes the probability in a dynamic programming manner. The Inside-Outside algorithm does a bottom-up for inner probabilities and top-down for outer probabilities. The Viterbi algorithm only does a bottom-up computation as described in detail in Chapter 3.

1.7 Organization of this thesis

An overview of the software system, the main modules, and the contributions of this thesis is given in Chapter 2. The underlying idea for this thesis, the HMM with Variable Observation Sequences, is described in Chapter 3. Chapter 4 explains how the different probabilities in the model are estimated. Chapter 5 shows a verification algorithm that ranks candidate words for a given oversegmented word image, and Chapter 6 discusses a neural classifier used to reject bad inputs and improve the recognition rate. Then, Chapter 7 extends the work by introducing a parallel implementation of a simplified version of the system. Each chapter gives the results of the algorithms it presents. Finally, Chapter 8 concludes and suggests future work.

Chapter 2

System Overview

2.1 Architecture

This chapter outlines the software architecture and gives an overview of the main modules. As is typical among supervised learning systems, there is a training phase and a testing (or recognition) phase. The data, which come in the form of word images, is divided into two sets: the training set, for training the neural network, and the test set, for evaluating the performance of the system. Figure 2.1 shows a data flow diagram.

The majority of the code is written in C for an RS/6000 machine running AIX.

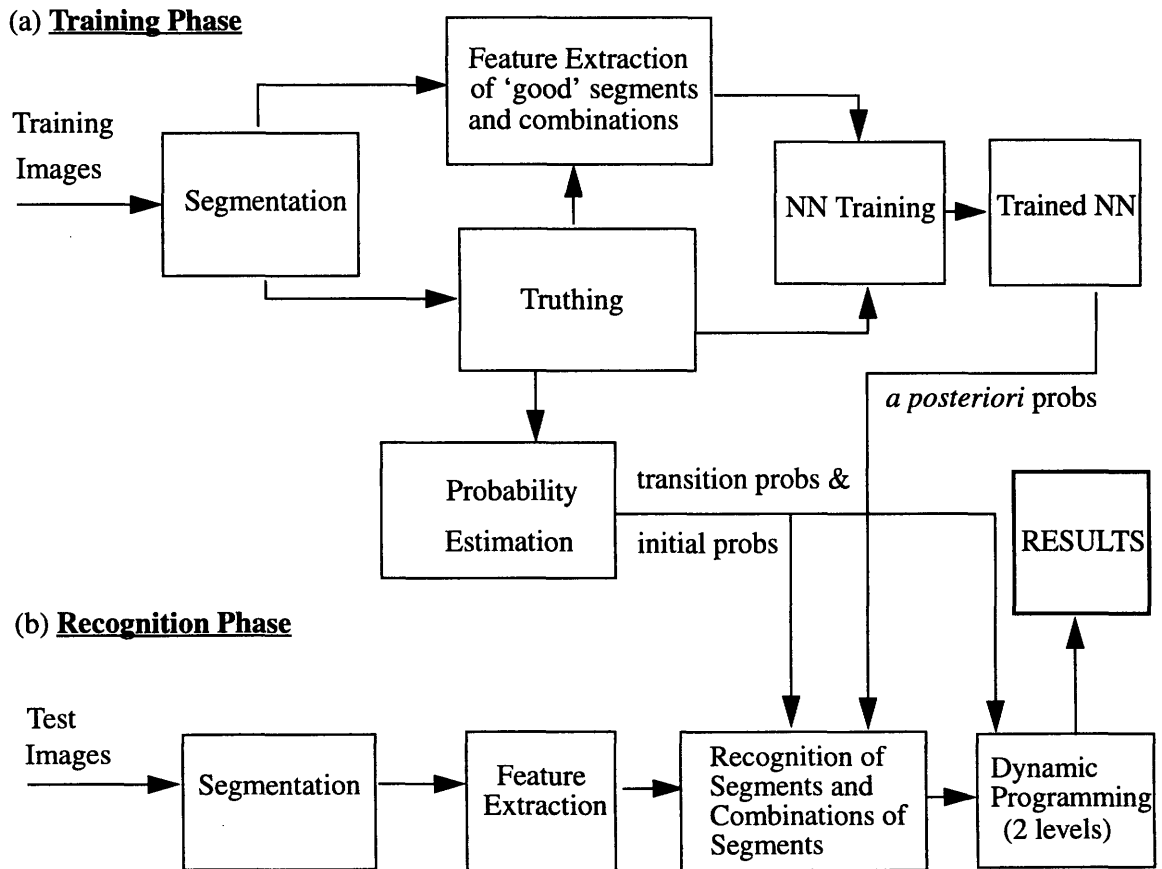


Figure 2.1: Software Infrastructure

2.2 Training

The purpose of the training phase is to train the neural network that generates the observation (character bitmap) probabilities and estimate state transition probabilities.

2.2.1 Oversegmentation

After word images are extracted from the input images¹ they are oversegmented. Continuous handwriting is cut into pieces by specifying the beginning and ending of the stroke. Pairs of points mark the top and the bottom pixels of these locations. Therefore, each segment has up to four cutting points. Segments that are at the beginning and end of a connected component only have two cutting points.

Each word is segmented into letters and parts of letters. A particular letter may have one, two, or up to three segments. All the letters in the word can be made up of segments or unions of adjacent segments. Figure 2.2 shows some examples of actual oversegmented word images.

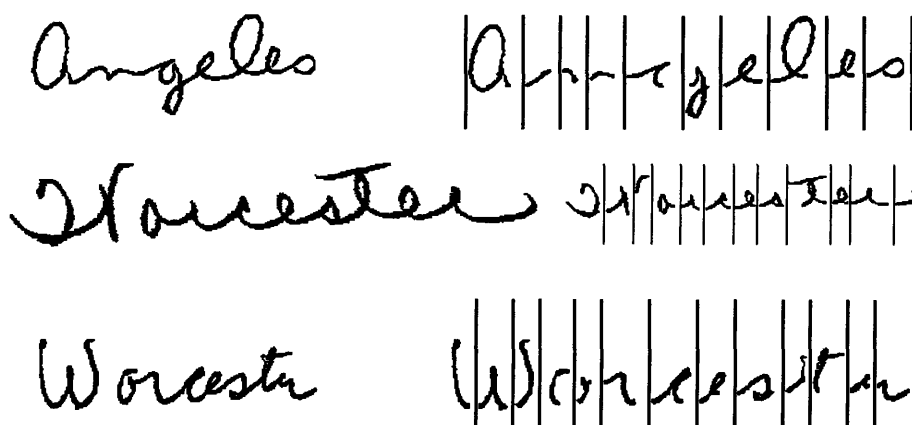


Figure 2.2: Oversegmentation examples.

1. In general, input images contain multiple words and even multiple lines of text.

2.2.2 Truthing

From segmented words, chunks that correspond to whole characters are manually chosen. They are manually labeled with the correct character states in a process called *truthing*. Only the truthed chunks are needed for training.

2.2.3 Feature Extraction

A feature vector composed of 108 features is extracted from each truthed chunk. We use 88 contour direction features [Mohiuddin & Mao 1994] and 20 cutting point features.

Contour direction features can be efficiently extracted by scanning the normalized 24x16 image by a 2x2 mask to detect primitive features when the number of black pixels in the mask is between one and three (neither all white nor all black). Histograms are computed by summing the number of primitive features detected along each of the horizontal, vertical, and two diagonal directions. Therefore, each detected direction is assigned four times according to its position in four projections. The vertical projection has four slices and the others have six slices (22 slices in total). This results in an 88-dimensional (22 slices x 4 directions) feature vector.

The relative positions of the cutting points in the bounding box of merged segments (previous segment with the current segment, the current segment with the next segment) are coded in 20 additional features, resulting in a 108-dimensional feature vector.

2.2.4 Neural Network character recognizer

The feature vectors and truth labels are used to train the neural classifier. The neural classifier provides the set of probability distributions that map observations¹ to the “hidden” states² of the HMM.

There are many character recognition techniques being used for off-line cursive HWR. The most popular ones are artificial neural networks. Among the different types of neural networks, we try three: a Multilayer Perceptron (MLP) [Rumelhart, Widrow, Lehr 1994], a Recurrent Network which is based on an MLP architecture, and a Hierarchical Mixture of Experts (HME) [Jordan & Jacobs 1994].

HME is a space partitioning scheme which was motivated by the fact that it can perform function decomposition and so learn separate sub-tasks within a large task more quickly and perhaps more effectively than a fully interconnected network [Jacobs, Jordan & Barto 1991]. The scheme has also been applied to the classification problem of speech recognition [Waterhouse 1993]. We hope to use it in our application in a similar manner to cope with high input variance.

2.2.5 Transition Probabilities

While the neural classifier is being trained, transition probabilities are estimated using the frequencies of transitions in the truth labels. The training data set is used as an implicit lexicon.

2.3 Recognition

In the recognition phase, test input or new text images are assigned ASCII representations.

1. These are segments or combinations of up to three adjacent segments, otherwise known as chunks.

2. The hidden states are actually character classes, e.g. ‘a’, ‘b’, in the model.

New images are segmented and extracted of features as described in the previous section. Then all the chunks are fed into the neural classifier. A dynamic programming Viterbi algorithm, described in Chapter 3, is then applied to the recognition results to find word hypotheses. These hypotheses are ranked according to probability using global context and lexicon constraints. Bad samples are discarded to improve recognition rates.

2.4 Software modules developed in this thesis

Some of the software modules needed in the system had already existed at the IBM Almaden Research Center. The new modules and that were constructed to complete the system, as well as extensions, are:

1. Two-level dynamic programming Viterbi algorithm for the HMM.
2. Neural Networks (attempts at HME and recurrent network, as well as training the MLP using only whole characters).
3. Transition and initial probability estimation.
4. Recognition of segments and combinations of segments in the recognition phase.
5. A verification algorithm that ranks candidate words.
6. Rejection of bad samples to increase recognition rates (uses an MLP).
7. Parallel implementation of simpler problem to explore ways to improve performance.

Chapter 3

HMM with Multiple Observation Sequences

3.1 HMM Framework

We propose to use the following Hidden Markov Model (HMM) framework. The model is a directed graph where the nodes are called states. A state, i , in the model corresponds to a character class. For example class 1 corresponds to the letter 'a', 2 to letter 'b', and so forth. The set of character classes $\{ 1, 2, \dots, N \}$ contains all English letters, numbers, and a subset of the punctuation marks. Every path through the model must start from a special state called π . A typical path looks like $Q = \langle \pi, q_1, q_2, \dots, q_T \rangle$, where $1 \leq q_i \leq N$. Paths can be interpreted as words where T is the number of letters in the word.

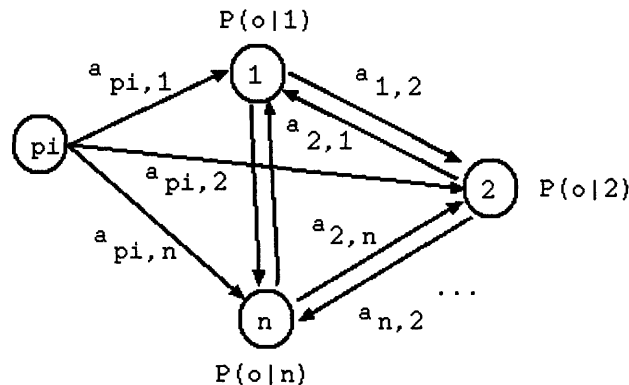


Figure 3.1: Hidden Markov Model

There are links from one state to another. Each link is associated with the probability of making a transition from the source¹ state to the destination state. These probabilities are called *transition probabilities*. They allow us to assign probabilities to entire words or paths through the graph. Consider a word $W = "c_1 \dots c_T"$ where c_i are characters for $i = 1$,

1. This is the state from which the arrow representing the directed edge starts. The destination state is where the arrow points.

..., T. Let q_c represent the state in the HMM corresponding to the character c . The word W can be represented as a state sequence Q

$$Q = \langle \pi, q_{c_1}, q_{c_2}, \dots, q_{c_T} \rangle$$

with probability

$$P(Q) = a_{\pi, q_1} a_{q_1, q_2} \dots a_{q_{T-1}, q_T}$$

In keeping with the laws of probability, the sum of the transition probabilities of all the outgoing links from any node must be equal to one.

$$\sum_{k=1}^N a_{q_i, q_k} = 1 \quad \forall (1 \leq i \leq N)$$

A link with probability equal to zero is equivalent to having no link at all. For example, all probabilities $a_{i, \pi}$ are equal to zero since no state can transition back to the π state.

The initial state probabilities, $a_{\pi, i}$, and transition probabilities, a_{ij} , can be estimated from the training data or from a lexicon.

So far we have described a simple Markov Model. We extend this by introducing observation probability distributions $P(o_i | i)$ at each state i . Here o is a chunk¹ of a word image. The joint probability of a path and a perfectly segmented bitmap $O = \langle o_1, o_2, \dots, o_T \rangle$ is

$$P(O, Q) = P(Q)P(O|Q) = (a_{\pi, q_1} a_{q_1, q_2} \dots a_{q_{T-1}, q_T}) [P(o_1 | q_1) P(o_2 | q_2) \dots P(o_T | q_T)]$$

1. Recall that a chunk is a combination of up to three contiguous segments.

The distributions $P(o|i)$ come from a neural classifier that process feature vectors that have been extracted from the bitmap image of the corresponding chunk. An example of a neural classifier is a the Multilayer Perceptron (MLP).

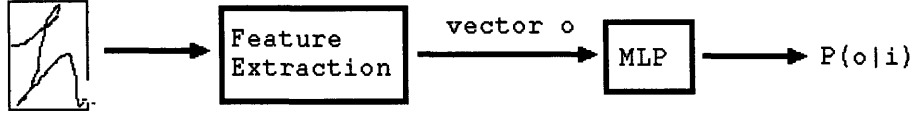


Figure 3.2: Input/Output Relation for MLP used as Neural Classifier

The problem of finding the word represented by an image S is therefore the exact same problem as identifying the state sequence, $Q = \langle \pi, q_1, q_2, \dots, q_T \rangle$, that maximizes the joint probability $P(Q,S)$.

A word image S can be represented by a sequence of segments, $S = \langle s_1, s_2, \dots, s_R \rangle$. If we are lucky, S will be perfectly segmented. There is a one-to-one correspondence between the segments s_i and the states q_i . So $R = T$. A Viterbi algorithm to find the underlying word, $Q_{opt} = \langle \pi, q_1, q_2, \dots, q_T \rangle$, that maximizes the likelihood of the word constructed by putting each segment in its own chunk, i.e.

$$\begin{aligned}
 Q_{opt} &= \underset{q_1, q_2, \dots, q_T}{\operatorname{argmax}} P(Q, S) \\
 &= \underset{q_1, q_2, \dots, q_T}{\operatorname{argmax}} (a_{\pi, q_1} P(s_1|q_1) a_{q_1, q_2} P(s_2|q_2) a_{q_2, q_3} \dots a_{q_{T-1}, q_T} P(s_T|q_T))
 \end{aligned}$$

This optimizes $P(S,Q) = P(S|Q)P(Q)$. It is equivalent to optimizing the *a posteriori* probability $P(Q|S) = P(S,Q)/P(S)$ since the word image is fixed.

In reality there is at present no segmentation algorithm that perfectly segments word images into the constituent characters. The word images in our system are usually over-segmented. At least one segment and at most three segments correspond to each character in virtually all cases.

To handle this general case, we incorporate in our Viterbi algorithm a search for the optimal grouping of the segments into characters. This search also uses a dynamic programming algorithm like the one used for the Viterbi algorithm. Therefore, the resulting modified Viterbi algorithm is a two-level dynamic programming algorithm.

3.2 Viterbi algorithm with multiple observation sequences.

This section describes the original Viterbi algorithm [Rabiner 1989] and the two-level dynamic programming “modified Viterbi algorithm”. The latter is original work for this thesis.

3.2.1 Original Viterbi Algorithm

Let us consider the simple case where we would like to identify the single best word that would account for a perfectly segmented word image S . This word is the path Q through the HMM that maximizes $P(S|Q)$. As shown above, this is the same path that maximizes $P(Q|S)$ after making some assumptions. To find this path the Viterbi algorithm [Viterbi 1967] [Rabiner 1989], which is a dynamic programming algorithm, is used.

To reiterate, we would like to find the best state sequence, $Q = \langle \pi, q_1, q_2, \dots, q_T \rangle$, for the given observation sequence $S = \langle s_1, s_2, \dots, s_T \rangle$.

Let us define the quantity

$$\delta_t(i) = \max_{q_1, q_2, \dots, q_{t-1}} P[q_1, q_2, \dots, q_{t-1}, i, s_1, s_2, \dots, s_t]$$

I.e. $\delta_t(i)$ is the highest probability along a single path, at time t , which accounts for the first t observations and ends in state c_i . By induction we have

$$\delta_{t+1}(j) = \max_i \delta_t(i) a_{ij} P(s_{t+1}|j)$$

To actually retrieve the state sequence, we need to keep track of the argument which maximized the equation above, for each t and j . We do this via the array $\psi_t(j)$. The complete procedure for finding the best state sequence can be stated as follows:

1) Initialization:

$$\begin{aligned}\delta_1(i) &= a_{\pi, i} P(s_1 | c_i) & 1 \leq i \leq N \\ \psi_1(i) &= 0\end{aligned}$$

2) Recursion:

$$\begin{aligned}\delta_t(j) &= \max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij} \\ \psi_t(j) &= \operatorname{argmax}_{1 \leq i \leq N} [\delta_{t-1}(i) a_{ij}] \\ \text{for } & 1 \leq j \leq N, 2 \leq t \leq T\end{aligned}$$

3) Termination:

$$\begin{aligned}\bar{P} &= \max_{1 \leq i \leq N} \delta_T(i) \\ \bar{q}_T &= \operatorname{argmax}_{1 \leq i \leq N} \delta_T(i)\end{aligned}$$

4) Path (state sequence) backtracking:

$$\bar{q}_t = \psi_{t+1}(\bar{q}_{t+1}) \quad t = T-1, T-2, \dots, 1$$

3.2.2 Modified Viterbi

Let us now relax the constraint that each segment corresponds to exactly one state. Because of the oversegmentation each state in a path through the HMM may now correspond to one, two, or three segments.

Let $s_{ij} = \langle s_i, s_{i+1}, s_{i+2}, \dots, s_j \rangle$ be a word image subsequence. If a word image has R segments s_1, s_2, \dots, s_R , the whole word can be written down as the image subsequence s_{1R} .

The subsequence s_{22} is just the sequence composed of only the second segment. We can regard each s_{ij} as a *subword* that itself needs to be identified as a sequence of states. Denote the optimal state sequence for s_{ij} as $Q_{ij} = \langle q_1, q_2, \dots, q_T \rangle$, with probability $P(Q_{ij})$. $T \leq j-i+1$ in general, since word images are oversegmented. Recall that the states $q_i \in \{1, 2, \dots, N\}$, where N is the number of possible states.

Let $\alpha_m(s_{ij})$ be the optimal state sequence for s_{ij} that *begins* with state m . Similarly, let $\omega_n(s_{ij})$ be the optimal state sequence for s_{ij} that *ends* with state n , and let $\rho_{mn}(s_{ij})$ be the optimal state sequence that begins with m and ends with n . The optimal state sequence Q_{ij} is therefore $\rho_{mn}(s_{ij})$ for some states m and n .

Suppose we know the optimal state sequences $\omega_n(s_{1k})$ and $\alpha_m(s_{k+1,R})$ for all $1 \leq m \leq N$, $1 \leq n \leq N$, and $k=1, 2, \dots, R$. The optimal state sequence Q_{1R} for the word s_{1R} is $\omega_{\bar{n}}(s_{1\bar{k}})\alpha_{\bar{m}}(s_{\bar{k}+1,R})$ where \bar{k} , \bar{n} , and \bar{m} are chosen by maximizing the probability

$$P(Q_{1R}) = \max_{1 \leq k \leq R, 1 \leq n \leq N, 1 \leq m \leq N} P(\omega_n(s_{1k})) a_{nm} P(\alpha_m(s_{k+1,R}))$$

We recursively solve a similar optimization problem for $s_{1\bar{k}}$ and for $s_{\bar{k}+1,R}$. In fact we can do this recursively for all s_{ij} . The recursion bottoms out when $j-i \leq 2$. If $i=j$, $P(Q_{ij}) = P(s_{ij}|c)$ from some class c . That is, the optimal state sequences of the segment subsequence s_{ij} is just the neural network likelihood output. We have a similar case for $j-i=1$ or $j-i=2$, except that the new segment formed by the merged segments is just one of the possible paths.

To avoid duplicating work done during the recursion we do a bottom-up tabular approach. We first find $\rho_{mn}(s_{ij})$ and $P(\rho_{mn}(s_{ij}))$ for $j-i=0$, then for $j-i=1, j-i=2, \dots, j-i=R-1$. When $j-i=R-1$, we get Q_{1R} and we are done.

The algorithm is laid out more formally below. Remember that $P(\rho_{mn}(s_{ij}))$ is the probability of a sequence and $P(s_{ij}|c)$ is a neural network output.

Initialization:

We find the values of $\rho_{mn}(s_{ij})$ and $P(\rho_{mn}(s_{ij}))$ for chunks s_{ij} where $j=i$, $j=i+1$, and $j=i+2$. These are chunks composed of one, two, or three contiguous segments, respectively. They are special because they can potentially represent single characters. Sequences of more than three contiguous segments can only be combinations of multiple characters.

The initialization, i.e. computation of $\rho_{mn}(s_{ij})$ and $P(\rho_{mn}(s_{ij}))$ for the smaller chunks, is more complicated than finding the same values for the larger subsequences because it involves neural network outputs.

For chunks with only one segment we have,

$$\rho_{m,n}(s_{i,i}) = \begin{cases} \langle m \rangle & m = n \\ nil & otherwise \end{cases}$$

$$P(\rho_{m,n}(s_{i,i})) = \begin{cases} P(s_{i,i}|m) & m = n \\ 0 & otherwise \end{cases}$$

for $1 \leq i \leq R, 1 \leq m \leq N, 1 \leq n \leq N$

For chunks with two segments we have,

$$\rho_{m,n}(s_{i,i+1}) = \begin{cases} \langle m \rangle & (m = n) \wedge (P(s_{i,i+1}|m) > P(s_{ii}|m)a_{mm}P(s_{i+1,i+1}|m)) \\ \langle m, m \rangle & (m = n) \wedge (P(s_{i,i+1}|m) \leq P(s_{ii}|m)a_{mm}P(s_{i+1,i+1}|m)) \\ \langle m, n \rangle & otherwise \end{cases}$$

$$P(\rho_{m,n}(s_{i,i+1})) = \begin{cases} P(s_{i,i+1}|m) & \rho_{m,n}(s_{i,i+1}) = \langle m \rangle \\ P(s_{ii}|m)a_{mn}P(s_{i+1,i+1}|n) & otherwise \end{cases}$$

for $1 \leq i \leq R, 1 \leq m \leq N, 1 \leq n \leq N$

For chunks with three segments, the possible $\rho_{mn}(s_{i,i+2})$ are of the form $\langle m \rangle$, $\langle m, n \rangle$, or $\langle m, l, n \rangle$ where $1 \leq l \leq N$. To simplify the expression for $\rho_{mn}(s_{i,i+2})$, let us first define the probabilities of each of these candidates.

$$\begin{aligned}
P(\langle m \rangle | s_{i,i+2}) &= P(s_{i,i+2} | m) \\
P(\langle m, n \rangle | s_{i,i+2}) &= \max(P(s_{ii} | m) a_{mn} P(s_{i+1,i+2} | n), P(s_{i,i+1} | m) a_{mn} P(s_{i+2,i+2} | n)) \\
P(\langle m, l, n \rangle | s_{i,i+2}) &= P(s_{ii} | m) a_{ml} P(s_{i+1,i+1} | l) a_{ln} P(s_{i+2,i+2} | n)
\end{aligned}$$

Now the optimal state sequence can be simply defined as

$$\rho_{m,n}(s_{i,i+2}) = \underset{\substack{\rho = \langle m \rangle, \langle m, n \rangle, \langle m, l, n \rangle \\ 1 \leq l, n, m \leq N}}{\text{argmax}} \{P(\rho | s_{i,i+2})\}$$

The probability of this path is therefore

$$P(\rho_{m,n}(s_{i,i+2})) = P(\rho_{m,n}(s_{i,i+2}) | s_{i,i+2})$$

Recursive case:

$$\begin{aligned}
\rho_{m,n}(s_{i,j}) &= \underset{\substack{i \leq k \leq j \\ 1 \leq x \leq N \\ 1 \leq y \leq N}}{\text{argmax}} \rho_{m,x}(s_{i,k}) a_{xy} \rho_{y,n}(s_{k+1,j}) \\
&\text{for } 1 \leq m \leq N, 1 \leq n \leq N
\end{aligned}$$

It is important to note that in order to avoid duplication of computations, we should do a bottom up approach where the current computation of $\rho_{mn}(s_{i,j})$ depends only on values that have already been computed. First fill in all the values for the case where $j-i = 3$ since all the ρ 's for $j-i < 3$ have been computed in the initialization. Then fill in the values for $j-i = 4$, and so on.

Termination:

When we have filled in the table for $j-i=R-1$ from the recursive case and have found the $\rho_{mn}(s_{i,R})$ paths, i.e. the optimal paths for the entire word s_{1R} , it suffices to find the optimal choices of m and n to find the single best path,

$$Q_{opt} = \max_{1 \leq n \leq N, 1 \leq m \leq N} P(\rho_{m,n}(s_{1,R}))$$

3.2.3 Correctness of Modified Viterbi Algorithm

This section sketches a proof of correctness for the modified Viterbi Algorithm. We need to show partial and total correctness. The algorithm has partial correctness if, assuming it terminates, it comes up with the correct result. Total correctness is partial correctness combined with the fact that the algorithm terminates.

For partial correctness, we need to show that, if it terminates, the algorithm finds a state sequence that optimizes $P(\rho_{\mathbf{mn}}(s_{1,R}))$. To do this, we define the algorithm in terms of a generic dynamic programming algorithm and show that it has the optimal substructure property. This property arises from the fact that all the ways of splitting a subword s_{ij} into two parts are themselves optimization problems. If they do not give optimal solutions, i.e. there exist solutions that have higher probability, then those solutions would be more optimal, thus posing a contradiction.

To show total correctness, it suffices to show that the algorithm terminates. The algorithm fills in the rows of the table, starting from row $r = 1$ to row $r = R$. For each r there are at most $R-r$ table entries to fill in, and for each table entry, there are at most N^2 combinations to consider. Since all these steps take a bounded number of computations, the entire algorithm itself take a bounded number of steps. Therefore, it terminates.

3.2.4 Asymptotic Performance

Let $T(r)$ be the time to find the paths $\rho_{mn}(s_{i,i+r-1})$ and their corresponding probabilities. Because we use a bottom-up approach, we can assume that the values for all subsequences of $s_{i,i+r-1}$ are already known.

$$\rho_{m,n}(s_{i,j}) = \underset{\substack{i \leq k \leq j \\ 1 \leq x \leq N \\ 1 \leq y \leq N}}{\operatorname{argmax}} \rho_{m,x}(s_{i,k}) a_{xy} \rho_{y,n}(s_{k+1,j})$$

There are $r-1$ ways of choosing k in the optimization equation shown above. There are N ways to choose x and N ways to choose y . Therefore, $T(r)$ is $O(rN^2)$.

The computation of the probabilities can be absorbed into this running time because we can store the values we find as we search for the argmax . In the base case where $r=1$, the computation is also $O(rN^2) = O(N^2)$ because this is the Neural Network output.

The algorithm goes through $r=1$, then $r=2$, and so on until $r=R$. There are R subsequence sets in the row for $r=1$, $R-1$ subsequence sets for $r=2$, and only one for $r=R$. Thus, the number of subsequence sets for row r is $R-r+1$. Hence the total runtime must be

$$\begin{aligned} & \sum_{r=1}^R (R-r+1) O(rN^2) \\ &= \left((R-1) \sum_{r=1}^R O(rN^2) - \sum_{r=1}^R O(r^2N^2) \right) \\ &= \left((R-1) O\left(\left[\frac{R(R+1)}{2}\right]N^2\right) - O\left(\left[\frac{R(R+1)(2R+1)}{6}\right]N^2\right) \right) \\ &= O\left(N^2 \left[\frac{(R+1)(R+1)R}{2} - \frac{R(R+1)(2R+1)}{6} \right]\right) \\ &= O(R^3N^2) \end{aligned}$$

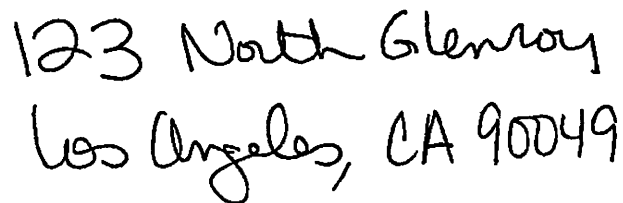
In any particular instance of the algorithm, the number of states N is kept constant so the asymptotic running time is $O(R^3)$.

In practice, we speed up the algorithm by not keeping all N^2 candidate subsequences. Many of the candidates have very low probabilities and are not worth keeping around.

Instead, only a short list is stored. This could be fixed in length or of variable length with a threshold for the probabilities for the elements on the list¹.

3.3 Results.

Examples of input/output pairs of the modified Viterbi algorithm for HMMMOS are shown below. The hypothesis words are separated by vertical bars, the first being the top choice for the word and so on. Bear in mind that these are raw results which have not been used to generate candidates that actually belong to the lexicon. Furthermore, no context information has been used. Chapter 6 (results in section 6.5) introduces a scheme which uses context information and chooses hypotheses that can be found in the lexicon.



123 North Glenroy
Los Angeles, CA 90049

{ 123 | a | w } { wenglenwy | naaglenwy | wenghiny }

{ 6o | bo | la } { agle, | asle, | aso7 } { ca | la | ea } { 9ma | ama | ana }

Analysis: The first word, the number “123”, is correctly recognized as the first hypothesis. The words “North Glenroy” are combined into a single word by the algorithm because of their proximity. The “glen” part is captured by all three hypotheses, as is the “y” at the end. The “ro” is always recognized as a “w”, and this is consistent with a human inspection of the letters as drawn. “North” is not satisfactorily recognized although it is hoped that given the context of the other letters, the words “North Glenroy” becomes one of the candidate words. In the second line, “los” is recognized as “60”, “b0”, or “la”

1. To keep the algorithm running, at least one candidate is put in the list even if all the candidates do not exceed the threshold probability.

because of the shape. The hypothesis “agle” captures four of the letters in “angeles” correctly and “CA” is correctly recognized. Due to the cursive manner in which the zip code is written, it is recognized as a sequence of alphabetic characters instead of numeric digits.

345 Dr. Pine St.
Reedsburg, Wis. 53959

{ 345 | 34s | b45 } { a | w | an } { ane | pine | ave } { stu | wn | stn }

{ wnan | wwang | awiling } { wan | sana | sano } { o3937 | o3957 | o3959 }

Analysis: The first word “345” is accurately recognized. The second word, which the author cannot read himself, is hypothesized to be “a”, “w”, or “an”. The correct reading of the next word, “pine”, is actually the second hypothesis for that word. “st” is interpreted as “stu” because in this instance of the HMMMOS “.” is not a character. In the second line, “Reedsburg” is not satisfactorily recognized, and the author cannot read the state. The last zip code hypothesis has the correct digits except for the leading “5”.

1000 N. Market St.
Milwaukee, WI
53202

{ 1000 | 100 | ,m } { n7 | nn | w7 } { nundsta | nalatsta | nundsto }

{ nllwadia, | nllwadale, | mllwadia, } { wi | lwi | wt }

{ 5 | s | 3 } { 32o2 | ao2 | 22o2 }

P.O. BOX 354
WAUPACA, WI 54981

{ pa | ra | 8o } { box | bor | 8ox } { 35 | 3s | 3a } { 4 | t | 7 }

{ wau | wav | wal } { paca, | nca, | raca, } { vi | wi | ui } { 549 | 54g | 5t9 } { 81 | al | pl }

7457 Edmond Ave.
Middleton, WI 53562

{ 74 | 14 | ta } { 57 | a | w } { ewinanaven | ewintwaven | ewinawaven }

{ misileton | wisileton | mosileton } { , | l | i } { wi | w, | o51 } { 53o2 | 533o | 535o }

5509 Maywood Rd
Monona, WI
53716

{ 5 | br | 52 } { 4o | 5o | ho } { y | 9 | g } { may | my | ney } { 1oo4 | mnt, | ano4 } { 4 | a | al }

{ mononn | monoun | monann } { , | l | r } { wi | wl | win } { 53716 | o7716 | ay6 }

P.O. Box 428
Glen Head, LI NY
11545

{ pna | rna | rda } { 428 | 423o | pa }

{ bla | blan | ma } { acad, | mad, | wad, } { n | h | w } { ny | cy | wy }

{ 11545 | 115457 | 11575 }

P.O. Box 568
Honesdale, PA 18431

{ 8oxn | 8o8ox | poxn } { 5kf | 5hf | shf }

{ nannd, | nannn, | nanwa, } { pa | ra | ta } { 16431 | 18431 | inn }

917 Williamson St.
Madison, WI 53703-3549

{ 917 | 947 | 91n } { willimon | willianon | willianson } { st | st, | sti }

{ madgonlwi | madigonlwi | madgonlwl } { 537o3-3549 | 53o3-3549 | mo3-3549 }

315 N Henry St
Madison, WI 53703

{ 3 | 37 | 57 } { 15 | li | i5 } { n | w | v } { heng | heno | heny } { st | a | w }

{ nadson, | madson, | nadsoni } { wi | we | all } { 537o3 | 5323 | 53233 }

Chapter 4

Probability Estimation

4.1 Overview

Two kinds of probability distributions are used in the Hidden Markov Model. There are transition probabilities for the underlying Markov process and there are observation probabilities that map chunks of word images to the “hidden” states. The estimation of both types of distributions are discussed in the sections that follow.

4.2 Initial and Transition Probabilities

The probability that any word will start at a particular character or state i is called the initial probability $a_{\pi,i}$. A transition probability a_{ij} is the probability of going from state i to state j in the word sequence. I.e., the probability that character i is immediately to the left of character j in any word sequence. Initial and transition probabilities can be estimated from a lexicon. In the absence of a separate lexicon, the training data could be used.

4.3 Observation Likelihood Estimation (Character recognition)

We try three techniques for estimating the or observation probability distributions: a Multilayer Perceptron (MLP), a recurrent network, and a Hierarchical Mixture of Experts (HME).

Neural Networks approximate *a posteriori* probabilities $P(j|s)$. However, we need a neural network to estimate likelihoods $P(s|j)$. A conversion from a posteriori probabilities to likelihoods is given by Bayes rule. If s is the segment image and j is the character class being considered, then

$$P(s|j) = \frac{P(j|s)P(s)}{P(j)}$$

For a particular segment s , the distribution of likelihoods for all j is proportional to the following expression.

$$P(s|j) \approx \frac{P(j|s)}{P(j)}$$

The relative magnitudes of these likelihoods are preserved in the approximation above. The actual likelihoods can therefore be computed by normalizing the estimates. Note that $P(j)$ can be readily estimated from the training data.

4.3.1 Multilayer Perceptron (MLP)

The multilayer perceptron, trained using backpropagation, is a popular neural network architecture for this particular application. The appeal of this technique is in its simplicity in implementation and the fact that it has been successfully applied to various handwriting recognition problems in the past.

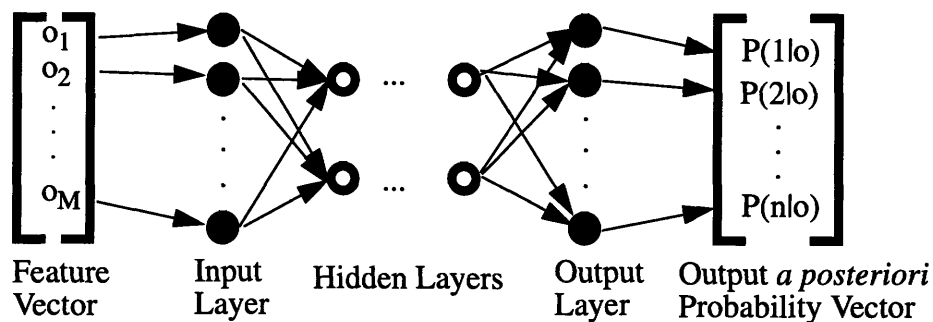


Figure 4.1: Multilayer Perceptron

MLPs consist of simple calculation elements, called *neurons*, and weighted connections between them. In a feedforward Multilayer Perceptron (MLP) the neurons are

arranged in layers. A neuron from one layer is fully connected with the neurons of the next layer.

The first and last layers are the input and output layers, respectively. The layers between them are called *hidden layers*. Feature vectors are given to the neurons in the input layer; the results come out of the output layer. The outputs of the input neurons are propagated through the hidden layers of the net. The figure below shows graphically the algorithm that each neuron performs.

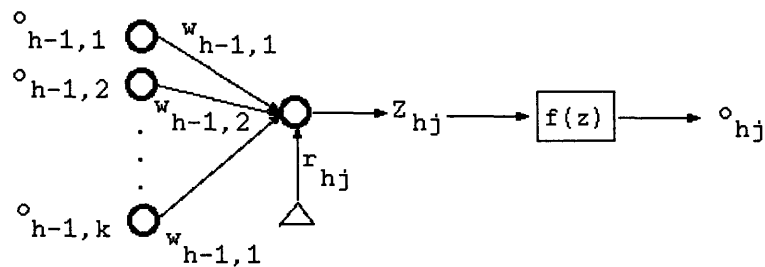


Figure 4.2: Node at layer h and row j

The activation z_{hj} of a hidden or output neuron j at layer h is the sum of the incoming data multiplied by the connection weights like in a matrix product. The individual bias value r_{hj} is then added. I.e.

$$z_{hj} = r_{hj} + o_{h-1,1}w_{h-1,1} + o_{h-1,2}w_{h-1,2} + \dots + o_{h-1,k}w_{h-1,k}$$

There are k neurons in layer h-1. The output u_{hj} is calculated by a nonlinear function f. A popular nonlinearity used in practice is the sigmoid function.

$$u_{hj} = f(z_{hj}) = \frac{1}{1 + e^{-z_{hj}}}$$

A feedforward multilayer perceptron can approximate any function after a suitable

amount of training. Known discrete values of this function, together with expected outputs, are presented to the net in the training phase. It is then expected to learn the function rule [Rumelhart et. al. 1994].

The behavior of the net is changed by modification of the weights and bias values. The network is trained using the Backpropagation algorithm [Cun 1986] [Rumelhart et. al. 1986].

A common performance metric for neural network training is the sum of squared errors. Although theoretically the outputs of a MLP trained using the sum of squared errors can approximate *a posteriori* probabilities [Richard & Lippmann 1991] (under certain assumptions such as a sufficient number of training samples), in practice, due to the finite number of training samples, the probability distributions for likelihoods of character classes given the character image are quite sharp. That is, there are one or two character classes with very high probabilities and the other character classes have close to zero probabilities. Due to high variability in handwriting, the true character class may be given a likelihood that is very close to zero. We want a smoother distribution that will make it more likely for a globally oriented dynamic programming algorithm to pick the true character class in spite of it having a low likelihood for a particular instance. We propose to compare the results between using the Kullback-Leibler divergence and the sum of squared errors. The paper [A2iA 1996] discusses how probability distributions can be smoothed using the combination of the softmax function and the Kullback-Leibler divergence as a metric.

4.3.2 Recurrent Neural Net

A Recurrent Network was also tried. The network takes in two feature vectors: that of the chunk being recognized as a character and that of the segment immediately to the left. The latter provides context which would hopefully help the network identify the chunk correctly. The context helps distinguish between character classes which look similar by themselves but look obviously different when they are seen adjacent to the preceding segment.



Figure 4.3: Right and middle parts of “w” vs. “u” when context is present.

For example, the chunk composed of the middle and right ascenders of a “w” looks very similar to a whole “u” (See Figure 4.3). We would like the chunk to be negatively recognized as a “u” and for a real “u” to be recognized as a “u”. If we include the preceding segment of each, we get a “w” and a “au”, respectively, where “a” is some other character. Now it is very clear that the two images are different.

The basic architecture of the recurrent net that was implemented is a Multilayer Perceptron. The network operates exactly as an MLP except that the feature vector for an extra segment is used.

In order to provide even more context, the segment immediately to the right was also

included. The resulting architecture is a *double-sided* network.

4.3.3 Hierarchical Mixture of Experts

The paradigm of divide-and-conquer is a prominent one in computer science. By dividing a problem into smaller subproblems and then combining the results, we can often come up with efficient and elegant solutions. Divide-and-conquer can be applied to the neural network approach to handwriting recognition by dividing the feature vector space and having a neural network specialize on each division. Other neural networks can be used to arbitrate between these specialists. The combinations of the arbitrators and the specialist neural networks are called modular networks. An interest in modular networks has been stimulated in recent years by the emergence of ‘space partitioning algorithms’ like Classification and Regression Trees (CART) [Breiman, Friedman, Olshen & Stone 1984], Multivariate Adaptive Regression Splines (MARS) [Friedman 1991], and Hierarchical Mixtures of Experts (HME) [Jordan & Jacobs 1994].

The divide-and-conquer approach embodied by modular neural networks is an intriguing one for handwriting recognition. People have various styles in handwriting but there are certain regularities in handwriting styles within certain groups. For example, children tend to draw bigger and more irregular characters than adults. The writer’s nationality and sex may affect the character shape, stroke width, and slant in somewhat predictable ways. In light of these observations, we try to use a modular neural network for off-line cursive handwriting recognition. In particular, we implemented the Hierarchical Mixture of Experts (HME) model.

Among the modular neural networks mentioned above, HME is particularly attractive because it uses “soft splits” of the data. This means the data is allowed to lie simultaneously in multiple regions. This reduces the variance of the recognizer and therefore helps correct one of the problems associated with dividing the input space. The HME has already been applied to regression and classification problems such as robotics control [Jordan & Jacobs 1994] and speech recognition [Waterhouse 1993]. It has not, however, been applied to Handwriting Recognition.

The evolution of HMEs has been enhanced by the use of Generalized Linear Models (GLIMs) in the network and the development of a fast algorithm for training the ‘Hierarchical Mixture of Experts’ using the Expectation Maximization (EM) Algorithm of [Dempster, Laird, & Rubin 1997] [Jordan & Jacobs 1994]. The EM algorithm converges significantly faster than the more familiar gradient descent algorithms and could improve the convergence times for our particular application.

The HME architecture is shown in Figure 4.4. It is a tree with gating networks at the nonterminals and expert networks at the leaves. Gating networks receive each feature vector x as input and produce scalar outputs that are a partition of unity. Each expert network outputs a vector μ_{ij} for each input feature vector x . At the bottom-most level of nonterminals, the μ_{ij} are scaled by the values of the outputs of their corresponding gating networks to form intermediate outputs for the next higher level of nonterminals. This process continues until the root of the tree is reached. The output μ at the root is the output of the entire HME tree.

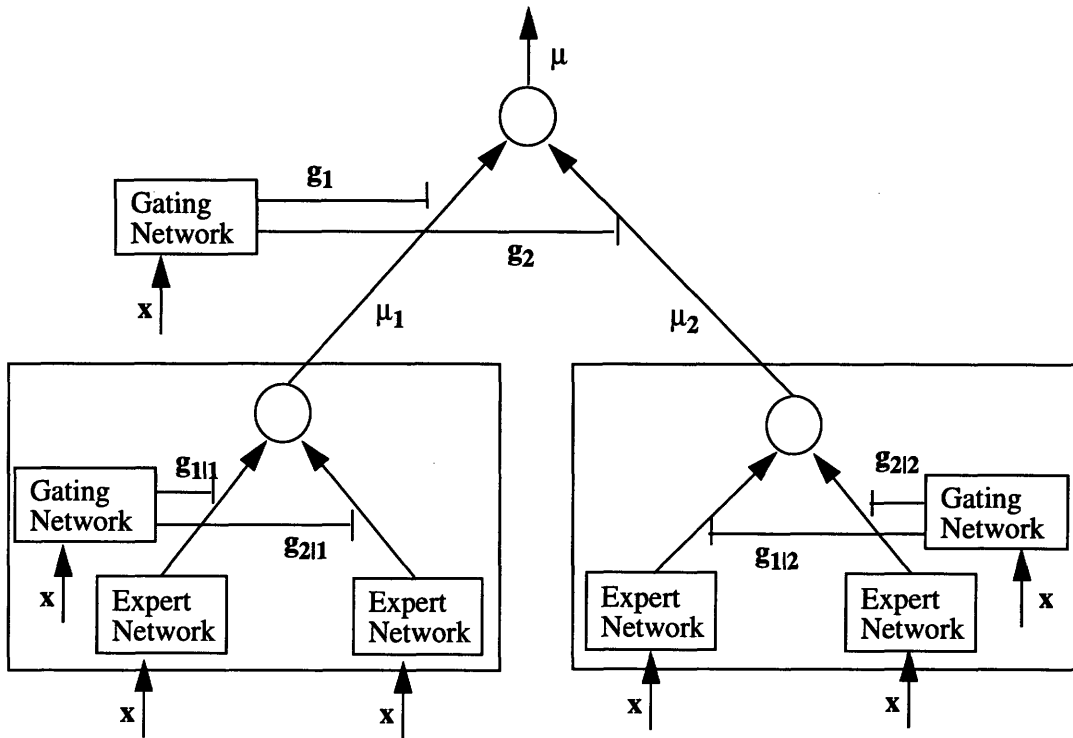


Figure 4.4: A 2-level HME. To form a deeper tree, each expert is expanded recursively into a gating network, a nonterminal node, and a set of subexperts.

4.4 Results

The character-wise performance for the MLP is

Training patterns: 158898 characters

Accuracy: 77.50% (top choice)

Test patterns: 25300 characters

Accuracy: 74.30% (top choice)

The architecture of the MLP is

108 input units corresponding to 108 features.

27 outputs for the Alpha network (26 letters and a reject class).

80 hidden units are used.

The HME never converged when Generalized Linear Models (GLIMs) were used because the covariance matrices became singular very rapidly and the training algorithm

diverged. Attempts to control the divergence by enforcing absolute lower bounds on the eigenvalues did not help.

The recurrent network (along with the one with pre- and post-context) also did not work properly.

Chapter 5

Rejection of bad samples to increase recognition rates

5.1 Problem Definition

The system described in previous chapters does not correctly recognize all inputs. There are inputs that are too noisy, too distorted, or just do not lend themselves to proper preprocessing. In order to increase the recognition rate we can reject these bad inputs.

The situation can be modeled by having two distributions of oversegmented word images. One is for all word images that will be correctly recognized by our system. The other is for word images that will not. The distributions live in a multidimensional feature space¹. Our goal is to increase the recognition rate by identifying a decision boundary and using our recognition system only on those that are on the “good” side of this boundary.

A simple example of this model is when there are two one-variable Gaussian distributions in a random variable x as shown in Figure 5.1.

Distribution A is for the bad inputs; B is for the good inputs, i.e. the ones that will be correctly identified. p_1 is the probability of an input being bad; p_2 is the probability of an input being good. $p_1 + p_2 = 1$. Since our space consists of only one scalar variable x , the decision boundary consists of a single threshold x_0 . In our example, every word image whose feature x is greater or equal to x_0 is used as input into the system for recognition. Each image whose feature x is less than x_0 is rejected.

1. These features are discussed in section 5.3.

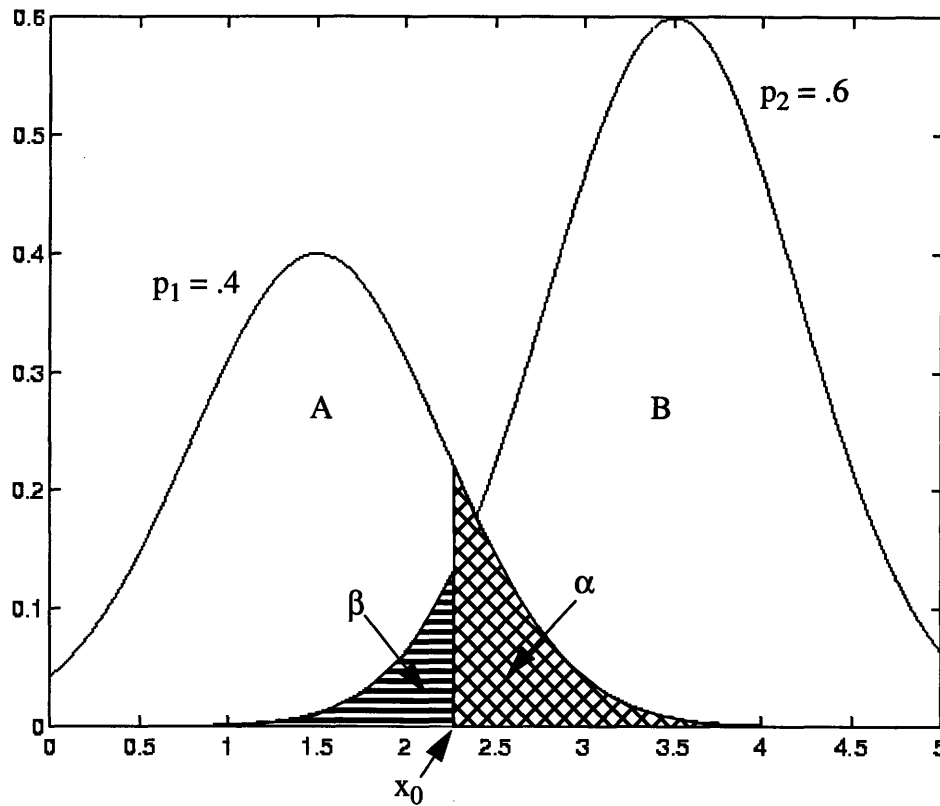


Figure 5.1: Two gaussians representing good and bad word images.

Some bad images might still be included in the input set for the system because their feature values are on the “good side” of the boundary. The probability of including a bad image in the input set is α as shown in Figure 5.1. It is usually called the *false positive* rate. Similarly, some good images will be rejected, causing the recognition rate of the system to go down. The probability of rejecting a good image is β as shown. It is called the *miss rate*. The rejection rate of the system is defined as $p_1(1-\alpha) + p_2\beta$. The recognition rate is $p_2(1-\beta)/[p_1\alpha+p_2(1-\beta)]$. The overall success rate is $p_2(1-\beta)$.

It is desirable to have a high recognition rate and a low rejection rate. x_0 can be moved around to manipulate these values but there is a trade-off. In most cases a lower α increases the recognition rate of the system. As we move x_0 to the right, α decreases but β

increases, thus increasing the rejection rate. The opposite happens when we move x_0 to the left. α increases causing the recognition rate to decrease. However the rejection rate decreases at the same time due to the decreased β . To maximize the overall success rate, the optimal value of x_0 has to be found. The next section discusses our method of finding this value.

5.2 Multilayer Perceptron Approach

In practice the distributions are not Gaussian and x is a multi-dimensional vector, with x_0 representing a surface in that space. Finding the optimal x_0 is equivalent to finding the best partitioning of the space into two parts: the good and the bad side. We do this using a Multilayer Perceptron (MLP).

An input to the rejection system is composed of an oversegmented word image and its corresponding candidate word list. The verification algorithm discussed in Chapter 6 is performed to rank the candidates. Features are then extracted from all this information and fed into the rejection MLP. The output is a boolean value which is normally represented by real number with a threshold. If the number is greater than or equal to zero, the word image is considered to be on the good side and is kept as input. If it is less than zero, it is pronounced bad and is rejected.

5.3 Features used

The nine features that were used as input to the MLP are listed below. When it is not specified, the feature refers to the top candidate.

1. Probability of top candidate divided by the number of characters of the top candidate.
2. Difference in probability of the top two candidates.
3. Number of deleted segments divided by the number of characters.
4. Number of characters that were ranked no. 1 by the neural character classifier divided by the number of characters.
5. Number of characters that were ranked no. 2 by the neural character classifier

divided by the number of characters.

6. Number of characters that were ranked no. 3 by the neural character classifier divided by the number of characters.

7. Number of characters that were ranked no. 4 by the neural character classifier divided by the number of characters.

8. Number of characters that were ranked no. 5 by the neural character classifier divided by the number of characters.

9. Number of characters that were ranked no. 6 to 10 by the neural character classifier divided by the number of characters.

Feature 1 distinguishes hypotheses that have high probability purely because there are fewer characters, and hence fewer neural net likelihoods to multiply, from those that have high probabilities because the average neural net likelihood is high. This feature is intended to salvage very long hypothesis words.

From empirical results, the difference in probability between the true hypothesis and any other hypothesis is large compared to the difference between two wrong hypotheses. Feature 2 arises from this observation.

Feature 3 separates hypotheses that have many *delete characters* from those with few. A delete character is a one-segment character that is unreadable and does not belong to a chunk that is recognized as a character, i.e. it does not represent any of the states of the HMM. A good match between a hypothesis and the word image should not create too many delete characters relative to the number of characters in the hypothesis.

Features 4 to 9 collect information on the relative rankings of the characters in the hypothesis based on neural network likelihoods. The higher the rankings of the character states in the hypothesis, the more confident we are that the hypothesis is a good match for the word image.

5.4 Results

Figure 5.2 shows the trade-off between the error rate and the rejection rate when a rejec-

tion network is used to filter word images that are fed into the verification algorithm. The data is taken from the CEDAR (Center for Excellence for Document Analysis and Recognition) database. All the usable city name images were used.

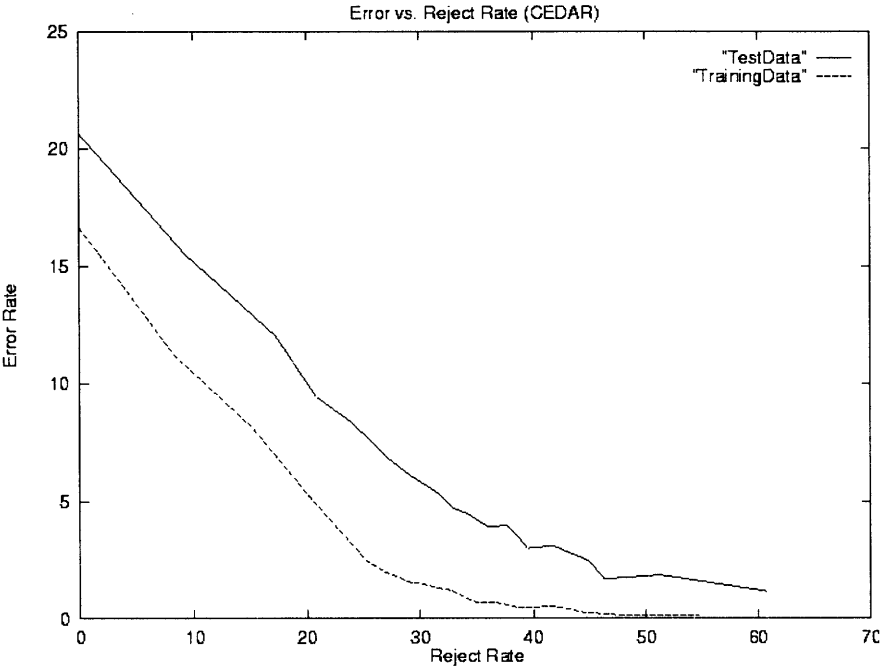


Figure 5.2: Error rate v.s Rejection rate for the verification algorithm

Chapter 6

Improving recognition by verification and correction

6.1 Introduction

The system outlined in Chapter 3 produces sequences of characters which might not even correspond to actual words. This presents two opportunities to improve the recognition rate: (a) the lexicon can be used to identify candidate sequences that are also actual words; and (b) the context given by surrounding word images could help better identify the best state sequence for each word image.

This chapter discusses a new dynamic programming algorithm which exploits these opportunities. It ranks word candidates taken from the lexicon for a particular segmented word image. This constrains the list of candidates to be members of the lexicon. The candidates are chosen using an existing verification and correction system [Lorie 1994] that finds words (state sequences) that are actually in the lexicon and “fit” the neighboring words.

The input to the verification and correction system is a list of candidate state sequences for each word image in an entire line of text. This is normally generated by the system described in Chapter 3. The output is a list of candidate state sequences for each word image. These candidates are from the lexicon and were chosen for their closeness to the input state sequences and for their relevance in the context provided by the neighboring word images.

Once the candidate list is retrieved, the candidates are ranked by the dynamic programming algorithm. The algorithm, using the oversegmented word image, maps the candidates to probabilities and sorts them according to the probabilities.

6.2 A Dynamic Programming Algorithm

In this section we describe a new dynamic programming algorithm that, given an oversegmented word image, maps candidate state sequences to probabilities.

Let us review some notation. A *word image* is of the form $S = \langle s_1, s_2, \dots, s_R \rangle$. The s_i are segments and there are R of them. A *candidate* is of the form $Q = \langle \pi, q_1, q_2, \dots, q_T \rangle$. There are T character states. Because the image is oversegmented, each character could be represented by at least one and at most three contiguous segments. Therefore $T \leq R$.

We introduce a new construct, *combo*, which is a partition of S into *chunks*. Recall that a chunk is a sequence of one, two, or three segments contiguous segments. A combo is of the form $O = \langle o_1, o_2, \dots, o_T \rangle$ and it forms a one-to-one mapping between chunks and non- π states in the candidate. The chunk o_1 can only be one of $\langle s_1 \rangle$, $\langle s_1, s_2 \rangle$, and $\langle s_1, s_2, s_3 \rangle$. Similarly, o_T could be $\langle s_{T-2}, s_{T-1}, s_T \rangle$, $\langle s_{T-1}, s_T \rangle$, or $\langle s_T \rangle$. The combo is only *valid* if, (a) each segment occurs in at most one chunk; (b) the combo contains all the segments in the word image; and (c) if a segment appears before another segment in the word image, the chunk of the former appears before, or in the same chunk as, the latter.

If $T < R$, multiple combos correspond to the word image. Let us define the function f over the set of all oversegmented words S to be $f(S) = \{ O \mid O \text{ is a valid combo for } S \}$. If S corresponds to the word Q , all but one of the elements O of $f(S)$ are mis-segmentations of Q . Otherwise, all the elements of $f(S)$ will be mis-segmentations. At least one chunk o_i of a mis-segmentation combo will not look like a valid character. A few elements of $f(S)$ will prove to be a close match for the candidate. The degree to which a combo is close to the perfect segmentation is captured in a probability estimate. If the state sequence Q is a bad candidate for S , all the probabilities for the O 's will be low. If Q is a good candidate, at least one candidate will have a high probability.

The algorithm discussed here finds the combo with the highest probability. This probability, multiplied by the probability of Q itself, is then regarded as the probability mapped to the candidate in the system described in section 6.1.

The *a posteriori* probability estimate for a combo O given a candidate Q is

$$P(O|Q) = P(o_1|q_1)P(o_2|q_2)\dots P(o_T|q_T)$$

The best combo O^* and its probability are given by

$$O^* = \underset{O \in f(S)}{\operatorname{argmax}} P(O|Q)$$

$$P(O, Q) = P(O^*|Q)P(Q)$$

We have taken out all the conditioning with respect to S in the expressions above since the equations are all conditioned on some word image S. Recall that P(Q) is just the Markov chain probability

$$P(Q) = a_{\pi, q_1} a_{q_1, q_2} \dots a_{q_{T-1}, q_T}$$

So now the problem is how to find O^* efficiently. We go through the algorithm with an example. Suppose we had the word image in Figure 6.1 and we wanted to match it with the state sequence $Q = \langle \pi, 'w', 'o', 'r', 'd' \rangle$.

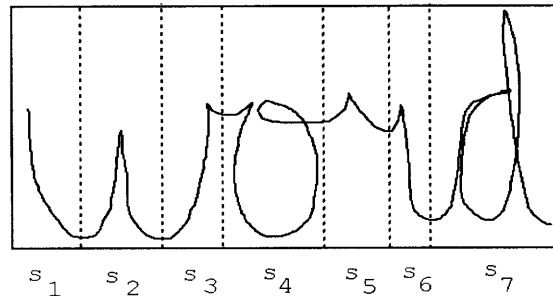


Figure 6.1: An oversegmented image of “word”.

The dynamic programming table for this case is shown in Figure 6.2.

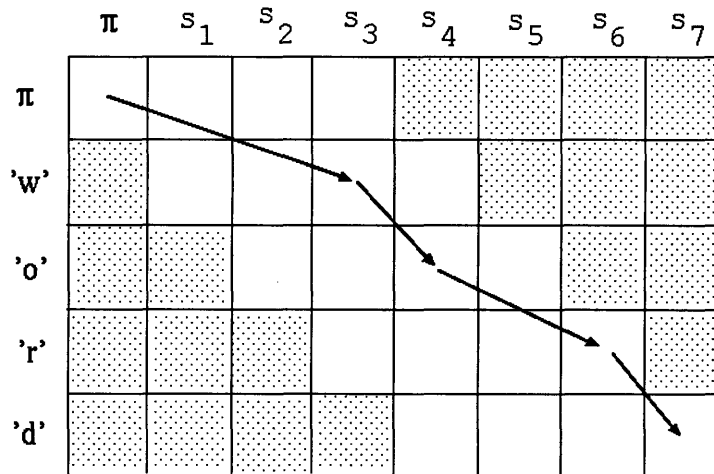


Figure 6.2: Dynamic Programming Table ψ for “word”.

Every combo must start at the (π, π) position and end at the lower right square on the table. Each chunk corresponds to an arrow. An arrow goes from one row to the next, and down and to the right. It can only go one, two, or three spaces to the right, and one space down. The number of spaces it goes to the right denotes the number of segments the chunk eats up.

For example, in the combo shown in Figure 6.2, the chunk $o_1 = \langle s_1, s_2, s_3 \rangle$ corresponds to ‘w’, $o_2 = \langle s_4 \rangle$ corresponds to ‘o’, $o_3 = \langle s_5, s_6 \rangle$ corresponds to ‘r’, and $o_4 = \langle s_7 \rangle$ correspond to ‘d’. The sequence $\langle \pi, o_1, o_2, o_3, o_4 \rangle$ happens to be the optimal combo. The table is used to explore different possible combos and find the optimal (highest probability) one.

Note that the algorithm does not use the entire table. There are entries that are useless because they violate the constraints on the relationship between chunks and segments. Therefore, only squares within the parallelogram are used.

Let us now go through the algorithm in detail. It requires two dynamic programming tables: $LP(i,j)$ and $\psi(i,j)$, where $i = 0, 1, \dots, T$ and $j = 0, 1, \dots, R$. Each element in the table

corresponds to a state i and a segment j . As a special case $LP(0,0)$ and $\psi(0,0)$, the upper left square in the tables, correspond to the π vs. π element. An example of a ψ table is shown on Figure 6.2.

For each valid i and j value, $LP(i,j)$ is to the maximum log probability¹ of ending the i^{th} character with the j^{th} segment, and hence the acronym. Each element of the table $\psi(i,j)$ is a pair $\langle k,l \rangle$ which indicates the position of the box which maximizes the probability $LP(i,j)$.

Initialization:

1. For $i = 0, 1, \dots, T$
2. For $j = i, i+1, \dots, \min(i+T-1, T)$
3. $LP(i,j) = -\infty$
4. $\psi(i,j) = \text{nil}$

The initialization phase sets all the interesting elements of LP to negative infinity, the lowest log probability possible, and all the interesting elements of ψ to nil.

Fill in the tables:

5. $LP(0,0) = 0$
6. For $i = 1, \dots, T$
7. For $j = i, i+1, \dots, \min(i+T-1, T)$
8. For $k = \max(i,j-3), \max(i,j-3)+1, \dots, j-1$
9. $\text{tmp} = LP(i-1,k) + \log(NN(k+1, \dots, j))$
10. If $(\text{tmp} > LP(i,j))$
11. $LP(i,j) = \text{tmp}$
12. $\psi(i,j) = \langle i-1, k \rangle$

1. Note that maximizing log probabilities is the same as maximizing probabilities since logarithm is a monotonically increasing function.

Line 5 sets the probability of $LP(0,0)$ to zero in order to give it the highest log probability among other squares on the 0'th row, making it the starting point of all paths through the table. The next row will only point to either $\langle 0,0 \rangle$ or to nil. Line 6 starts the loop that goes through the rows of each table going from the first to the last (T^{th}) character in the hypothesis word. Line 7 starts the loop through the columns in each row. In order to have only valid combos as discussed earlier, not all the columns are explored. The loop on Line 8 goes through the possible ways a link could point to the box being considered. There are at most three, each corresponding to the number of segments that are joined together to form a chunk. For each of these possibilities, the log probability of the source box is added to the neural net output for the corresponding chunk (see line 9). The neural net takes in the feature vector extracted from the chunk composed of the $k+1^{\text{st}}$ up to the j^{th} segment. The source box that gives the highest log probability in line 9 is chosen and the log probability value is updated in line 11. Line 12 stores the location of this box in the ψ table.

The optimal log probability for the hypothesis is $LP(T,R)$. We often want to find the optimal combo that gave rise to this probability to visually check the algorithm's behavior. This is done by tracing back the pointers in the ψ table as described in the pseudocode below. The notation $\psi(x,y).j$ gives the j -component of the tuple stored in $\psi(x,y)$.

13. $C_T = \psi(T,R).j$
14. For $i = T-1, T-2, \dots, 1$
15. $C_i = \psi(i, C_{i+1}).j$

Each C_i is one less than the index of the first segment of the i^{th} chunk. It could also be interpreted as the index of the last segment of the $i-1^{\text{th}}$ chunk. In line 13, C_T is set to be the j -component of the $\psi(T,R)$, i.e. the segment or column number of the square that maximizes the step to the last chunk. It is therefore the index of the last segment of the chunk that precedes the last chunk in the sequence. In lines 14 and 15, the process recurses by

finding the j -component of the square that maximizes the step to the last chunk on the reverse path. C_1 must be equal to zero because all optimal paths start at the upper left hand square.

The optimal combo is therefore

$$\langle \langle s_{C_1+1}, \dots, s_{C_2} \rangle, \langle s_{C_2+1}, \dots, s_{C_3} \rangle, \dots, \langle s_{C_T+1}, \dots, s_T \rangle \rangle$$

The algorithm does not allow for the existence of *delete characters*, segments that are specks of noise in the bitmap and do not really correspond to a valid character. Our implementation extends the algorithm above by allows for one-segment delete characters by eating up a segment without assigning it to a character/state. This implies a horizontal movement in the dynamic programming tables. In the pseudocode, line 8 is extended to also go through the box directly to the left of the box being considered, i.e. $\langle i, j-1 \rangle$. The neural network probability assigned to this single segment is its likelihood of being a delete character divided by an empirically derived parameter. This parameter is introduced because most single segments would have a high probability of being a delete character and delete characters should only be used sparingly to avoid mistaking parts of actual characters for delete characters. Note that the extra processing introduced in lines 8 to 12 adds a greedy component in the algorithm because local decisions are made about the identity of a segment, i.e. being a delete character.

Another exception to the basic algorithm is the case where the oversegmentation fails and two characters are actually put together in a single segment. This happens rarely but if not handled properly, it causes the algorithm to perform suboptimally. Fixing it involves considering box $\langle i-2, j-1 \rangle$ in addition to the other boxes in line 8. This modification does not introduce any greediness to the algorithm.

Note that these two modifications expand the parallel work area into the entire rectangular table for both LP and ψ . This implies an increase in the running time.

6.3 Correctness

For the purpose of proving the correctness of the dynamic programming algorithm consider only the basic algorithm which does not have any greedy components.

For partial correctness, it suffices to show that the algorithm has an optimal substructure property. At each box, the values that are compared must be optimal themselves. Otherwise, there is a contradiction.

The algorithm terminates because there are a finite number of entries in the tables to fill. Hence, we have total correctness.

6.4 Asymptotic Performance

We do an $O(1)$ computation for each element of the parallelogram in the table. The width of the parallelogram is $R-T+1$. The length is $T+1$. So the parallelogram has $(R-T+1)(T+1)$ elements and filling up the table takes $O(R-T+1)(T+1)O(1)$ or just $O(RT-T^2)$. Backtracking takes $O(R-T+1+T+1) = O(R)$ time so filling up the table so the whole algorithm takes $O(RT-T^2)+O(R)$. Informally, since the backtracking path contains unique blocks and these blocks form a proper subset of the parallelogram, the total running time must be $O(RT-T^2+R) = O(RT - T^2)$. Recall that $T \leq R$ because there are at most as many states as there are segments. Therefore, an upper bound on the running time is $O(R^2)$.

If we use the entire table, as is the case when we allow for delete characters and segments which contain more than one character, the running time is $O(RT)$. This is bounded above by $O(R^2)$.

6.5 Results

The rejection network was evaluated using the candidate ranking verification algorithm

Therefore, the results for the latter is simply the degenerate case for the rejection network where the rejection rate is zero percent (see section 5.4). That is, for the test data there is a 79% success rate and for the training data there is an 83% success rate. This is an improvement over [Kornai 1997] which yielded 63.3% success rate on the same database (CEDAR).

Chapter 7

Extension: Parallel Implementation

7.1 Motivation and Problem Definition

Handwritten word recognition systems should lend themselves to quick training and perform online operation rapidly.

Quick training is desirable because there are typically large amounts of data [Petrowski 1991]. A system should also provide high throughput when it processes fresh inputs online. This is usually more important because users usually have stringent requirements for online performance at the plant site. A mail sorting system may need to process thousands of pieces of mail in a matter of a few hours.

In this chapter, we consider ways to speed up the runtime of the online handwriting recognition system. A simplified version of the system described in Chapter 3 is built using coarse-grain and fine-grain parallelism. It handles the case where an HMM is used to recognize hand printed words. Instead of having oversegmented bitmaps of words, the inputs are sequences of bitmaps of individual characters. By making this simplification we can use the standard Viterbi algorithm instead of the modified Viterbi algorithm discussed in section 3.2. This implementation serves only to demonstrate how the essential parts of a handwritten word recognition engine can be parallelized. In reality, the full power of a Hidden Markov Model (HMM) is not necessary for this problem.

As in the original model, character bitmaps are mapped to the HMM states¹ using probability distributions arising from a Neural Network.

1. The HMM states correspond to the letters of the alphabet 'a' through 'z'.

The system is implemented using C and MPI (Message Passing Interface) on a Sun HPC (High Performance Computer) cluster and run in both shared memory and distributed memory modes.

Coarse grain parallelism is used to compute the individual character probability distributions¹. Two processors alternately compute the Neural Network outputs of odd and even numbered character bitmaps.

Fine grain parallelism is used to compute vector “dot products” in parallel for the forward phase of the Viterbi algorithm. The states of the HMM are distributed among processors in almost² uniform blocks. The search for the maximizing state at the end of the forward phase is implemented using a pointer jumping algorithm.

The basic HMM for handwritten word recognition has twenty seven states. In the general case where we could be dealing with cursively handwritten words, the number of states could represent parts of characters and could grow very fast. This implies that for these cases, parallelism would have a greater impact. Data is gathered for timing purposes but since the implementation does not use the extra states, we do not test for correctness for these cases.

7.2 The Model

The Hidden Markov Model for hand printed word recognition has twenty seven states. There is a state corresponding to each letter of the alphabet: state 1 = ‘a’, state 2 = ‘b’, ..., and state N = ‘z’ where N = 26. The 27th state is a special start state called the π state. All paths through the model must start from this state. The figure in the introduction showing the generic HMM is reprinted in Figure 7.1.

-
1. Recall that these are outputs from the Neural Network classifier.
 2. The last processor gets the remainder after the states are equally split among the processors.

There is a link from the π state to every other state. The graph with all the states excluding π is fully connected. Each link corresponds to the transition probability of going from the source state to the destination state. As usual, the sum of the probabilities of all the outgoing links from a certain node must be equal to one. In keeping with the Markov property, the transition probability of going to some state only depends on which state you are currently at. These probabilities are derived from a lexicon containing the domain of words that our system will be used on.

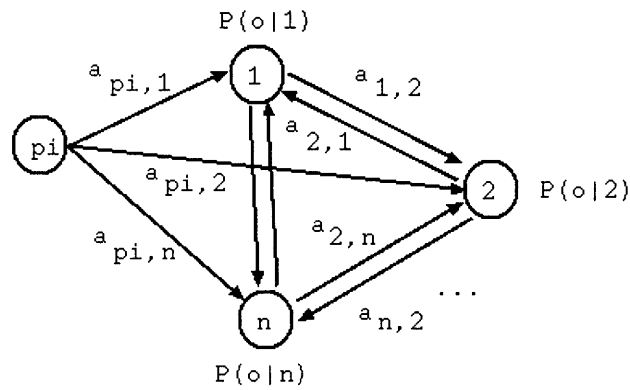


Figure 7.1: HMM with a π state and 26 character states

At each non- π state, there is a probability distribution $P(o_i|q_i)$ which maps the observations¹ to the states. We make the assumption that the handprinted characters are separate. This allows us to have identical indices for the observations and their corresponding states.

Given a sequence of character bitmaps, we would like to identify the single best word that would account for the sequence. We therefore need to maximize the probability of a sequence of character bitmaps O given a state sequence Q , $P(Q|O)$, which is equivalent to maximizing $P(Q,O)=P(Q|O)P(O)$ because the sequence of bitmaps stays fixed.

1. These are the feature vectors that are derived from the character bitmaps.

The probability of a certain bitmap sequence with feature vectors o_1, o_2, \dots, o_T given the state sequence q_1, q_2, \dots, q_T is

$$P(Q, O) = a_{\pi, q_1} P(o_1|q_1) a_{q_1, q_2} P(o_2|q_2) \dots a_{q_{T-1}, q_T} P(o_T|q_T)$$

The $P(o_i|q_i)$ come from a Multilayer Perceptron (MLP) that takes in feature vectors that have been extracted from the character bitmaps.

7.3 Coarse grain parallelism

Coarse grain parallelism is achieved by evaluating the outputs of the MLP for different bitmaps in parallel. In the viterbi algorithm, the probability distribution $P(o_i|i)$ is needed only at each step t .

Therefore, these computations can be computed in parallel and the results can be distributed to the processes representing the HMM states when they are needed.

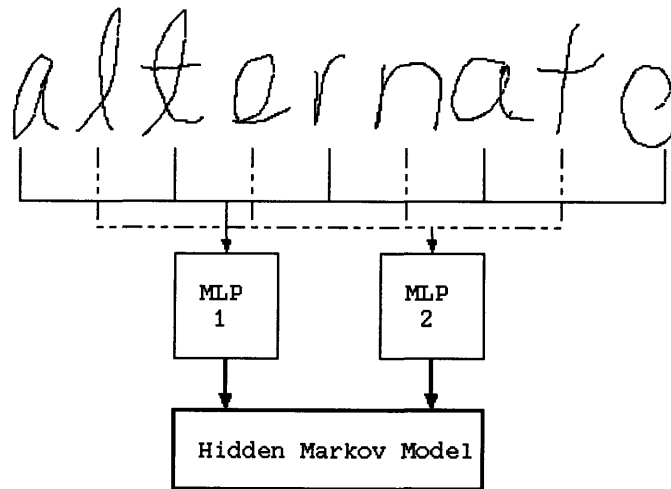


Figure 7.2: Coarse grain parallelism by alternating MLP evaluations

Two processors are assigned to evaluate the MLP. While one is distributing the results for time t , the other is computing the values for time $t+1$.

7.4 Fine grain parallelism

7.4.1 Forward-phase Viterbi

We use fine grain parallelism in computing the matrix-vector “product”¹ for each step t of the algorithm. I repeat the function evaluated at each state at time t at node i :

$$d_{t+1}(j) = \max_{1 \leq i \leq N} \{d_t a_{ij} P(o_{t+1}|j)\}$$

This can be done independently for each state i . Therefore, we can introduce parallelism by distributing the $n=26$ states over p processors.

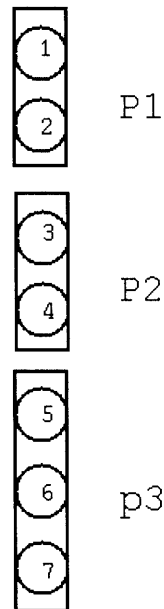


Figure 7.3: $n = 6$ nodes or states, $p = 3$ processors or processes

1. This operation actually replaces the addition operation in a Euclidean matrix-vector product with a maximizing operation. The analogy is used here because the runtime of both operations are the same.

The Viterbi algorithm now takes up the following amount of time:

$$c_1(p, n) + c_2\left(\frac{n}{p}\right)$$

c_1 is the cost of broadcasting $O(n/p)$ values to P processors; c_2 is the cost of doing n/p “dot products”¹. The problem is to empirically determine the value of p that minimize $c_1(p, n) + c_2(n/p)$.

This optimization problem is naively stated because it does not consider the time for backtracking phase of the Viterbi algorithm. If the states are spread across many processors, more communication time is used up querying the right processors for the pointers to the next state in the sequence. Hence, the optimization problem is to minimize the total time taken up by the forward phase and the backtracking phase.

Note that because the amount of time taken by the backward and forward phases are both proportional in the same way to the number of character bitmaps, the minimization problem is independent of word lengths.

7.4.2 Pointer jumping algorithm for finding maximum

At the termination of the forward phase of the Viterbi algorithm, we need to find the state that has the maximum probability. To do this, a pointer jumping algorithm was used to convert the $O(n)$ problem into an $O(\log n)$ problem [Cormen, Lieserson, Rivest 1990]. However, this code proved to be slow and was later taken out.

1. Again, it is not really a dot product because we compute the maximum instead of the sum.

7.4.3 MLP Evaluations

We could also use fine-grain parallelism for evaluating MLPs. The general approach is similar to the one for the Viterbi algorithm: we realize that we can split up matrix-vector multiplications into independent dot products evaluated at each node. This allows us to distribute the nodes among P processors. This approach is not implemented in the current system.

7.5 Results

The parallel implementation shows a factor of 2 speedup for HMMs with at least 1500 states peaking at 2 to 4 processors. The coarse grain parallelism for the MLPs generally improved performance and the pointer jumping algorithm did not. We only really gain speedups when the number of HMM states n is greater than 500.

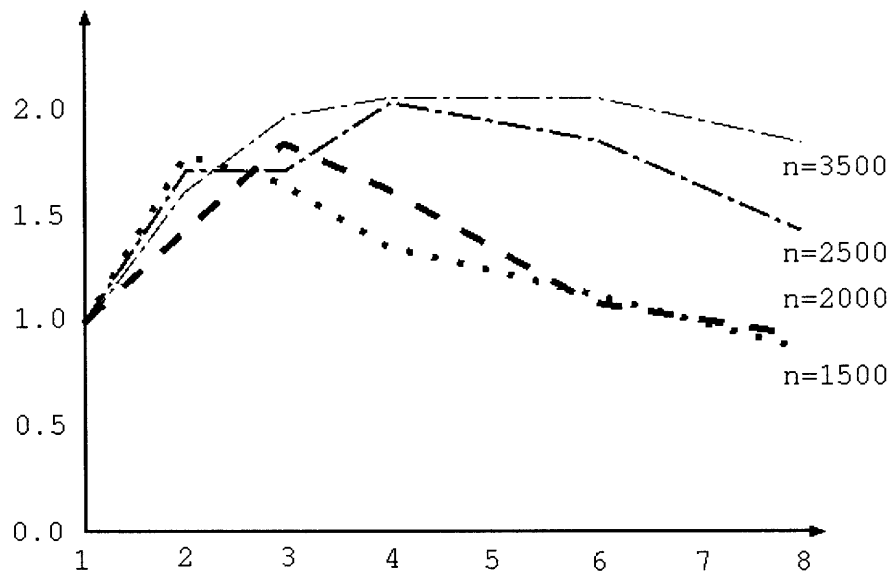


Figure 7.4: Speedup (y-axis) vs. number of HMM state processors (x-axis)

Chapter 8

Conclusions and Future Work

8.1 Conclusions

A system for offline cursive handwritten word recognition was built. As the framework for the recognition task, a variant of VSLHMM was introduced: HMM with Multiple Observation Sequences (HMMMOS). The results show a close visual correlation between the hypotheses and the actual word bitmaps. When global context and a lexicon are used to verify and rank hypotheses, the system recognizes words better than other systems when applied to the Center for Excellence in Document Analysis and Recognition (CEDAR) database.

Among the three neural classifiers tried, only the Multilayer Perceptron (MLP) performed satisfactorily. This might be explained by the fact that MLPs have been successfully used in the past for this kind of application, while Hierarchical Mixture of Experts (HME) and Recurrent Networks have been relatively unexplored.

Another MLP was successfully trained and used to reject bad inputs in order to increase the recognition rate.

Finally, a parallel implementation for a simpler system shows that adding one to three processors doubles the online recognition speed.

8.2 Future Work

8.2.1 Compare the three neural classifiers.

It is interesting to make the HME with Generalized Linear Model (GLIM) experts converge or find out why it does not converge. In [Waterhouse 1993] an HME with GLIM

(Multinomial) experts is applied to voice recognition. Although this did not converge either, a setup that did converge in that thesis is one that had a tree of HMEs, each with only two logistic functions. This kind of architecture is usually undesirable because the output probabilities are not normalized and there is no reliable way to compare them to even get a ranking. However, it would be interesting to see if this alternative architecture converges for our system.

Because the neural classifier is the basis and bottleneck for the HMM, it is important that we get good performance from it. It is therefore of interest to compare the three neural classifiers tried here and possibly others.

8.2.2 Stochastic Context Free Grammars

It would be interesting to implement and compare an SCFG system with the HMMOS. As mentioned in section 1.6, the two are similar and a further study to try to reconcile them might lead to better performance.

References

- [1] "The A2iA Recognition System for Handwritten Checks -- Intercheque V1.0 and Intercheque-Coupon V1.0", A2iA Technical Report No. 11, 22.03.1996
- [2] Y. Bengio, Y. Le Cun, D. Henderson, "Globally Trained Handwritten Word Recognizer using Spatial Representation, Convolutional Neural Networks and Hidden Markov Models," *NISP*, 1993
- [3] R. M. Bozinovic and S. N. Srihari, "Off-line cursive word recognition," *IEEE Trans. Pattern Analysis, Machine Intelligence*, vol. 11, pp. 68-83, Jan. 1989
- [4] L. Breiman, J. Friedman, R. Olshen, and C. J. Stone, Classification and Regression Trees, Belmont, CA: Wadsworth International Group
- [5] R. G. Casey and G. Nagy, "Recursive segmentation and classification of composite characters," in *Proc. Int. Conference on Pattern Recognition*, pp. 1023-1025, 1982
- [6] M. Y. Chen, A. Kundu, and S. N. Srihari, "Variable Duration Hidden Markov Model and Morphological Segmentation for Handwritten Word Recognition," *IEEE Transactions on Image Processing*, Vol. 4, No. 12, 1995
- [7] M. Y. Chen, A. Kundu, and Jian Zhou, "Off-line Handwritten Word Recognition Using a Hidden Markov Model Type Stochastic Network", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 16, No. 5, 1994
- [8] N. Chomsky, "Three models for the description of languages", *IRE Transactions on Information Theory*, Vol. 2 pp. 113-124.
- [9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, Cambridge, MA: New York : MIT Press ; McGraw-Hill, 1990
- [10] Y. L. Cun, "Learning Processes in an Asymmetric Threshold Network", Disordered systems and biological organization, pp. 233-240, Les Houches, France, Springer-Verlag, 1986
- [11] A. P. Dempster, N. M. Laird, D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm", *Journal of the Royal Statistical Society, B*, Vol. 39, pp. 1-38
- [12] R. Farag, "Word level recognition of cursive script," *IEEE Trans. Computer*, vol. C-28, no. 2, pp. 172-175, 1979
- [13] K. S. Fu, "Learning with stochastic automata and stochastic languages", Computer Oriented Learning Processes, J. C. Simon, ed., Noordhoff-Leyden, 1976.
- [14] J. H. Friedman, "Multivariate adaptive regression splines," *The Annals of Statistics*, 19, 1-141
- [15] Y. He, M. Y. Chen, and A. Kundu, "Alternatives to Variable Duration Hidden Markov Models in Handwritten Word Recognition," 1996
- [16] J. J. Hull and S. N. Srihari, "A computational approach to word shape recognition: Hypothesis generation and testing," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 156-161, 1986

- [17] R. A. Jacobs, M. I. Jordan, and A. G. Barto, "Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks," *Cognitive Science*, 1991
- [18] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Computation*, Vol. 3, pp. 79-87, 1991
- [19] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the EM algorithm," *Neural Computation*, vol. 6, pp. 181-214, 1994
- [20] A. Kornai, "An Experimental HMJM-Based Postal COR System", *Proceedings Intl. Conf. on Acoustics, Speech and Signal Processing*, 1997
- [21] K. Lari, S. J. Young, "Applications of stochastic context-free grammars using the Inside-Outside algorithm", *Computer Speech and Language*, Vol. 5 pp. 237-257, 1991
- [22] R. Lorie, "A System for Exploiting Syntactic and Semantic Knowledge," *Proceedings of DAS'94*, Kaiserslautern, Germany, 1994
- [23] G. Martin, M. Rashid, D. Chapman, and J. Pittman, "Learning to See Where and What: Training a Net to Make Saccades and Recognize Handwritten Characters," *NISP*, 1991
- [24] K. M. Mohiuddin and J. Mao, "A comparative study of different classifiers for hand-printed character recognition", *Pattern Recognition in Practice IV* by E. S. Gelsema and L. N. Kanal, 1994
- [25] M. Mohamed and P. Gader, "Handwritten Word Recognition Using Segmentation-Free Hidden Markov Modeling and Segmentation-Based Dynamic Programming Techniques," Accepted for Publication in *IEEE Trans Pattern Analysis and Machine Intelligence*, 1996
- [26] A. Petrowski, "A pipelined implementation of the back-propagation algorithm on a parallel machine". *Artificial Neural Networks*, pp. 539-544, Elsevier Science Publishers B.V. (North-Holland), 1991.
- [27] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition", *Proceedings of the IEEE*, vol. 77, No. 2, 1989
- [28] M. D. Richard and R. P. Lippmann, "Neural network classifiers estimate Bayesian *a posteriori* probabilities," *Neural Computation*, vol. 3, pp. 461-483, 1991
- [29] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning internal representations by error propagation", *Parallel Distributed Processing: Exploration in the microstructure of cognition*, vol. 1, pp. 318-362, MIT Press, Cambridge, MA, 1986
- [30] D. E. Rumelhart, B. Widrow, M. A. Lehr, "The Basic Ideas of Neural Networks", *Communications of the ACM* Vol. 37, No. 3, pp. 87-92, March 1994
- [31] R. Seiler, M. Schenkel, and F. Eggimann, "Off-line Cursive Handwriting Recognition compared with On-line Recognition," Swiss Federal Institute of Technology, Zurich, Signal and Information Processing Laboratory Technical Report No. 9601, 1996
- [32] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimal decoding algorithm," *IEEE Trans. Informat. Theory*, vol. IT-13, pp. 260-269, 1997
- [33] S. R. Waterhouse, "Speech Recognition using Hierarchical Mixtures of Experts," MPhil Thesis in Computer Speech and Language Processing at University of Cambridge, 1993

- [34] L. Xu and M. I. Jordan, "On Convergence Properties of the EM Algorithm for Gaussian Mixtures", *Neural Computation*, Vol. 8, pp. 129-151, 1996