GRADSIM: A CONNECTIONIST NETWORK SIMULATOR USING GRADIENT OPTIMIZATION TECHNIQUES

Raymond L. Watrous

MS-CIS-88-16
LINC LAB 103

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

March 1988

Acknowledgements: This research was supported in part by DARPA grant NOOO14-85-K-0018, NSF grants MCS-8219196-CER, IRI84-10413-AO2 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

GRADSIM: A Connectionist Network Simulator using Gradient Optimization Techniques

Raymond L. Watrous

March 7, 1988

Abstract

A simulator for connectionist networks which uses gradient methods of nonlinear optimization for network learning is described. The simulator (GRADSIM) was designed for temporal flow model connectionist networks. The complete gradient is computed for networks of general connectivity, including recurrent links. The simulator is written in C, uses simple network and data descriptors for flexibility, and is easily modified for new applications.

A version of the simulator which precompiles the network objective function and gradient computations for greatly increased processing speed is also described. Benchmark results for the simulator running on the DEC VAX 8650, SUN 3/260 and CYBER 205 are presented.

1 Introduction

The GRADSIM package is a collection of software modules in C that together constitute a general purpose connectionist network simulator which offers several methods of iterative gradient optimization techniques for training networks of arbitrary connectivity. It is designed to simulate networks which respond to binary and real-valued data, using simple network and data descriptors and standard unit functions. The package is modular and easily modified to accommodate variations in user problems and modeling requirements.

The GRADSIM package also contains a network compiler which produces C-code to compute the objective function and gradient vector for a specific network. The compiled C-code is optimized for processing speed.

This package was produced over the course of nearly two years during research in connectionist networks for speech recognition. The initial work in this area was undertaken using an early version of the Rochester Simulator [5,21]. This simulator was modified initially to permit use of the back-propagation algorithm for networks with recurrent links. Eventually, however, speed became an overriding consideration, and so a simulator was written that was streamlined for speed. The structures of the Rochester Simulator which provide generality and support interactive examination of network behavior were costly computationally.

In addition, several second-order methods for nonlinear optimization were being investigated[20,17], which would have required further modification to the Rochester Simulator.

This report describes GRADSIM Version 1.4, which has been regularly improved from previous versions. The hope in describing GRADSIM in this report is that it will be found useful as a research tool by other groups, and be enhanced in the course of being applied to different sets of problems.

2 Design Considerations

The GRADSIM simulator was developed with the following design criteria in mind:

1. Speed
2. Flexibility
3. Modularity

The speed criterion arose because of the application of connectionist networks to speech recognition problems, in which large amounts of data are required for network training, and in which network architectures which have recurrent links were employed. Speed was accomplished in one respect by making simplifying assumptions about the need to modify network unit functions. By writing these functions directly into the simulator, the structures which provide increased generality were excluded, with an associated savings in speed.

Flexibility was a design criterion because the network configurations and speech recognition experiments were varying over short time intervals as new insights were gained into network capabilities. It was decided to use ASCII network descriptor files for specifying network connectivity and characteristics. These files could be easily generated and modified using a standard editor (EMACS), as well as generated by UNIX shell scripts. This was preferred to a network generator in which the network was precompiled for a new version of the simulator.

Modularity was also an important consideration because of the need to investigate several learning algorithms, both first and second order, the desire to check these algorithms on standard binary valued test problems such as XOR and various coding problems, and the desire to modify unit functions, and the associated function and gradient computations. Consequently, the design that emerged uses a modular approach in which these types of changes can be made to independent parts of the package.

2.1 Learning

The design of GRADSIM reflects a perspective on machine learning that is decidedly de-psychologized. Following a definition of learning by Tsypkin as [15]

> a process of forcing the system to have a particular response to a specific input signal (action) by repeating the input signals and then correcting the system externally

the perspective of learning as a design problem was adopted. In this case, there is no requirement that the learning algorithm be biologically plausible, and the question then becomes which algorithm is computationally most effective.

This is not to say that learning as a psychological problem is not interesting and important, or that it cannot be addressed using GRADSIM. It is only to point out that GRADSIM includes learning methods such as back-propagation, steepest descent and second order methods, which have not been shown to be biologically plausible learning methods.

2.2 Temporal Flow Model

The design of GRADSIM was also affected by its original application to speech recognition using a model called the *temporal flow model* [21,19,22,20]. This model represents time implicitly and relatively, using delay links instead of time-spatialized units. The model is also characterized by internal recurrent links to provide dynamic temporal properties. This has lead to particular forms of the gradient computation, which are discussed later in this report. The point here is that the simulator is general with respect to connectivity and link delay, up to the constraints in effect when the simulator is compiled.

3 Overview

The GRADSIM connectionist simulator is compiled from a small number of C modules which are tailored to the optimization algorithm of choice, and the application problem. Once compiled, a particular simulator is capable of handling a wide variety of network architectures and experimental designs. The compilation relies on several "include" files to specify various maximum network dimensions.

3.1 Modules

The simulator is divided into several types of module, which are specific to the simulator, the optimization method, and the application.

3.1.1 Simulator Modules

The simulator modules handle reading and writing of the network and experiment descriptors, and perturbation of the link weights at initialization.

weight.c Handles reading and writing of network descriptor.

read.c Handles reading of experiment descriptor file.

map.c Performs internal mapping between link and array data structures, and provides general calls for objective and gradient evaluations.

perturb.c Performs initialization of network by adding a random small number to each link weight.

3.1.2 Optimization Modules

The simulator is compiled separately for each of four possible nonlinear optimization algorithms, which are listed here. In addition, all but the back-propagation algorithm employ the line_search module.

bp.c Implements the standard back-propagation algorithm, with fixed learning and momentum constants.

grad.c Implements the steepest descent gradient optimization algorithm using a linesearch routine.

dfp.c Implements the Davison-Fletcher-Powell quasi-Newton method.

bfgs.c Implements the Broyden-Fletcher-Goldfarb-Shanno quasi-Newton method.

line_search.c Implements a linesearch algorithm using cubic and quadratic interpolation and extrapolation.

3.1.3 Application Modules

The application modules provide code for the unit functions, and computation of the objective function value and gradient vector.

speech.c Provides for speech data input based on experiment descriptor and scaling of speech data.

unit.c Supplies routines to compute unit output function and forward pass through the network, using the potential function. Supports computation of objective function and gradient vector.

gauss_targ.c Supplies Gaussian target function values for objective and gradient computation, as well as target function gradient routines for adapting target function parameters.

compile_delay.c A network compiler which generates a network specific unit function module which is optimized for speed.

3.2 Include Files

The include files listed below specify the data types shared between modules and the maximum sizes of several important data structures. These constraints are checked during most of the simulation, although it is still possible to dump core.

net.h Contains the maximum network size, number of input units, the link data structure, and several other parameters.

read.h Contains the definition of the experiment descriptor data structure and related constants.

vector.h Contains the vector data structure definition and maximum link count parameter. Controls the basic simulator size, especially for the second order methods.

3.3 Compilation

Generation of the simulator is by means of a standard "makefile" using the UNIX "make" utility. An example "makefile" is included here for illustration.

```
CFLAGS = -O

bfgs : bfgs.o perturb.o unit.o weight.o line_search.o speech.o
map.o gauss_targ.o read.o
cc ${CFLAGS} -o bfgs bfgs.o perturb.o unit.o weight.o
line_search.o speech.o map.o gauss_targ.o read.o -lm

read.o speech.o : read.h
bfgs.o line_search.o map.o speech.o unit.o : vector.h
map.o perturb.o speech.o unit.o weight.o : net.h
```

3.4 Calling sequence

The simulator is called by the name of the optimization algorithm selected for compilation. Thus, for example, a simulator based on the Broyden method would be called by

```
bfgs -c original_net.1 experiment.4x
```

The "-c" option instructs the simulator not to perturb the link weights as read from the network configuration file, "original_net.1". This enables optimization to continue properly from a point of interruption, such as a power failure or system crash, which can easily occur in the three weeks it takes to optimize some networks!

Intermediate results are written to a file called "net.tmp.tmp" so that work is not lost. If the simulator terminates normally, a final configuration file, "net.x.fin" is also written. Care must be exercised in saving these files between runs of the simulator in the same directory.

3.5 Network Meta-Compiler

A network meta-compiler was developed [1] which generates C-code for the "unit.c" module to provide objective function and gradient evaluation subroutines for a particular network.

The meta-compiler was designed to generate code that is optimized for speed. Speed enhancements were accomplished by a number of means. The gradient and unit output array indices are precomputed using the network link delay values. The inner loops are unwound, and replaced by the correct number of explicit operations. This saves counter increment and test operations. Register variables, data pointers and autoincrement arithmetic operators are also used for speed.

In the process of optimizing the code, it was observed from an examination of the assembly code of the inner loops that multiple float/double conversions were being generated in the calls to the exponential function. Changing the data types consistently to "double" solved this problem.

Minimizing array initialization was done by sorting the links by delay and controlling the sequence of value initialization and increment.

The network meta-compiler speed enhancements have been measured in benchmark tests as described in Section 5. The compiler has been used on MASSCOMP [2], DEC VAX and SUN 3/200 series computers.

The meta-compiler is itself compiled, and then called for each network separately, to generate a module which replaces "unit.c" in the simulator compilation makefile described above.

3.6 Network Descriptor

The network descriptor is an ASCII file that consists of a threshold unit identifier and link descriptors. The descriptor is defined in terms of a unit index, which is based at zero. A single threshold unit is identified explicitly. Each link is defined in terms of its from unit, to unit, delay and initial weight values, as shown in the example below.

```
Threshold Unit 26
0 16 1 0.05
1 16 1 0.05
2 16 1 0.05
3 16 1 0.05
```

[1] Bruce Ladendorf, Siemens Research and Technology.

[2] It includes a switch option in order to correctly handle the difference between MASSCOMP and UNIX in treating exponential function overflow.

```
4 16 1 0.05
5 16 1 0.05
6 16 1 0.05
```

The input units are assumed to begin at unit 0, and extend for the number of units defined in "net.h" as 'INPUT_UNITS'. There are no other restrictions on the order of link descriptors, or unit indexing. The number of links must not exceed "MAX_VECTOR", which is defined in "vector.h". The highest unit index should be less than "MAX_NUM_UNITS" in "net.h".

The number of output units is controlled by the NUM_OUTPUT_UNITS constant in "net.h". The indices of the output units is specified in the experiment descriptor.

3.7 Experiment Descriptor

An experiment descriptor is an ASCII file which describes the network output units, the input data files, and the target value parameters for each output unit and input file.

The output units descriptor consists of an output unit count, and a corresponding number of output unit indices. The output units must be fewer in number than "MAX_OUTPUT_UNITS" in "read.h". This is followed by an arbitrary number of data file names, up to the maximum given by "MAX_TRAIN" in "read.h".

For each data file, the target function for each output unit is specified by a + or -, and two parameters. These parameters can be used to characterize the target function, by giving the mean and standard deviation for a Gaussian, for example.

```
3                   24  25  26
data/cv01rw01.eng   -   -   +   0.500 3.00
data/cv11rw01.eng   -   +   -   0.500 3.00
data/cv21rw01.eng   +   -   -   0.500 3.00
```

The experiment descriptor is easy to generate and modify. The only caution in using it is that the output unit indices must match the network being optimized. If the experiment descriptor is used in an optimization experiment with a network descriptor which it doesn't match, unpredictable results will occur. This problem can only partially be solved by checks in the descriptor I/O routines, since output units cannot be implicitly defined.

4 Simulator Theory

The problem of learning using connectionist networks, in which the network connection strengths are modified systematically so that the response of the network increasingly approximates the desired response, can be structured as a nonlinear optimization problem. This section presents very briefly some of the optimization methods which are included in the simulator. More detailed information can be obtained about the theory of nonlinear optimization from other sources [11]. The application of nonlinear optimization to connectionist network learning is discussed in a separate technical report[16].

The methods described here have in view a learning process in which the network performance is improved by minimizing a mean squared error measure. The gradient methods have in common the computation of the partial derivative of the error with respect to the weight values, and proceed iteratively by modifying the weights in the direction of the negative gradient.

4.1 Gradient Computation

The computation of the gradient of the error surface for feedforward connectionist networks is straightforward. A method for approximating the gradient for networks with recurrent links was presented [21] which relied on truncating the recursion in the gradient computation after a fixed number of steps.

A method for computing the complete gradient for networks of general connectivity was derived by Kuhn[10] which proceeds from left-to-right in time, accumulating partial results. A similar method was developed by Ladendorf[18], in which the computation of network activation proceeds from left-to-right, with a single right-to-left pass to compute the gradient. Thus, two methods exist for computing the complete gradient in networks of arbitrary connectivity.

4.2 Back Propagation

The back propagation (BP), or error propagation algorithm for learning in connectionist networks was reported by Rumelhart, Hinton and Williams[3][14,13].

The algorithm is an iterative method in which a step in the negative gradient direction is taken at each iteration. The step size is a fixed constant. The name error- or back propagation derives from the definition of an error value for each unit, which is recursively computed from the error values of units to which it is connected. The error at the output units is defined as the difference between the actual and desired values.

An accelerated gradient technique for back propagation has been published called 'momentum' [13]. The momentum term is defined by:

$$\Delta w_{ij}(n+1) = \mu \Delta w_{ij}(n) - \eta \frac{\partial E}{\partial w_{ij}(n)}$$

where w_{ij} is the connection strength from unit i to unit j, E is the error function value, η is the learning constant and μ is the momentum constant. The effect of the momentum term is to magnify the learning constant for regions of weight space where the gradient is essentially constant, and to focus the movement in a downhill direction by averaging out the components of the gradient which alternate in sign [16].

In the present GRADSIM simulator, the learning constant and the momentum term are compiled constants, ETA and MU.

4.3 Steepest Descent

Steepest descent is a gradient optimization method in which the error function is fully minimized along the direction of the negative gradient, at which point the gradient is recomputed and the process is iterated.

For connectionist learning theory, it should be noted that this method is not a local method; that is, the modification of the weights must be controlled by a central process which monitors the minimizing step size for each gradient direction. This is true also of the quasi-Newton methods. If the focus of research is on solutions to problems rather than biologically feasible learning algorithms, steepest descent provides important advantages over back propagation.

[3]As the authors indicate, it appears that similar learning algorithms were proposed by Parker [12] and Le Cun [13]. An earlier version due to Werbos is more recently credited [23].

4.3.1 Line Search

Line search is an optimization method which is the basis for steepest descent, and is generally a standard component in more complex optimization algorithms. The problem is to find the minimum value of the objective function along a particular direction of search.

$$\min_{\alpha}(f(x + \alpha\sigma))$$

where α is a scalar, and σ is the search direction vector.

There are three aspects of the line search, bracketing, sectioning, and interpolation [7]. A bracket maintains the permissible range for α, which is reduced in a controlled way as the search progresses. The bracket is used to protect against undesirable results of an interpolation algorithm for which assumptions about the objective function are not met; for example, extrapolation from a point in a function which is not convex in the interval could lead to values of α outside the interval, and result in a failure to converge.

Sectioning is a method for subdividing the range of α in order to reduce the interval in which the minimum must lie. It is generally assumed that the objective function is unimodal in the region of interest, and that the process of sectioning can be carried out to isolate the neighborhood of that minimum.

The methods of interpolation (and extrapolation) are based on low-order polynomial approximations to the objective function. The coefficients of the polynomial can be computed from a few points, and the corresponding minimum estimated directly.

The line search is in practice the most critical part of an optimization algorithm, and involves many design decisions. These include criteria for terminating the line search, controlling the sectioning limits, the initial value of α, and the details of the interpolation algorithm (see Appendix A). Modification of the simulator in this regard may be indicated for some applications, and requires a thorough knowledge of the theory [11].

4.4 Quasi-Newton Methods

The second derivative of the error with respect to the network weights gives information about the curvature of the error surface, and is the basis for a set of powerful optimization algorithms. The inverse of the second derivative matrix can be used in conjunction with the gradient to more accurately estimate the point of function minimum value.

Since the computation of the second derivative may be very complex, and computationally expensive, and since the matrix inversion is $O(N^3)$, the so-called quasi-Newton methods were developed in which the inverse matrix is approximated iteratively. This avoids the direct computation of the second derivative, and the computational complexity is reduced by a factor of $O(N)$ [7].

The basic quasi-Newton algorithm consists of the following steps:

1. set a search direction $\sigma = -Hg$
2. minimize f along σ
3. update H

where H is the approximate inverse second derivative matrix.

The heart of the algorithm is the update strategy for the approximate inverse matrix. One method is called the Davidon-Fletcher-Powell (DFP) method [4,8], and is given as:

$$H_{DFP} = H + \frac{\delta\delta^T}{\delta^T\gamma} - \frac{H\gamma\gamma^T H}{\gamma^T H\gamma} \tag{1}$$

where $\gamma_i = g_{i+1} - g_i$ and $\delta_i = x_{i+1} - x_i$.

Another method is due to Broyden, Fletcher, Goldfarb and Shanno [3,6], the BFGS algorithm:

$$H_{BFGS} = H + \left(1 + \frac{\gamma^T H\gamma}{\delta^T\gamma}\right)\frac{\delta\delta^T}{\delta^T\gamma} - \frac{\delta\gamma^T H + H\gamma\delta^T}{\delta^T\gamma} \tag{2}$$

The initial inverse matrix H is usually selected to be I.

These algorithms are implemented very directly in GRADSIM. They are currently based on a square matrix of dimension MAX_VECTOR; this data structure could be reduced by approximately one-half because of symmetry to reduce memory usage, so larger networks could be handled.

5 Speed Enhancements

This section investigates the effect of several speed enhancement techniques on simulator performance. These include the algorithmic gradient computation improvement and network meta-compilation.

A benchmark program was used for performance evaluation which consisted of 11 iterations of the back-propagation algorithm on a simple two-layer temporal flow model network with 27 units and 164 links, including self-recurrent links at the 8 hidden and 2 output units. The training data consisted of two speech files, and a linear target function was used for training. The benchmark corresponds to the no/go problem described in [21].

The baseline version of the algorithm included a form of gradient computation in which the gradient is computed from left-to-right in time, and approximates the gradient by limiting the backward recurrent series to a fixed number of iterations[21]. This number was set to 3 for this case.

The second version shows the improvement due to the new gradient algorithm, which computes the complete gradient in a single right-to-left pass.

The third version of the simulator involves using multiple output arrays to store network responses to each training token separately. This trades memory for time, by reading and scaling the training data once. It also permits a single call to the forward pass function in the gradient computation, since the forward pass is computed in the objective function, which is always called prior to the gradient.

The meta-compiled version of the simulator uses C-code for the objective and gradient computation generated automatically for an individual network, as described above. This simulator performance is given for the multiple forward pass version, followed by the single forward pass version.

Table 1 lists the benchmark resource utilization statistics for the various simulator versions. The simulator versions were compared using the VMS C-compiler "vcc" on a VAX 8650 at the University of Pennsylvania. The values listed in the table are the average of three runs of the benchmark program, and show timing and program parameters obtained from the "time" utility under UNIX. The user time is given in seconds, the shared text size and unshared stack size are given together, the number of blocks on input and output are shown together, followed by the number of major page faults. The number of swaps was also recorded for each case, and was uniformly zero.

Version	Time	Size	I/O	Page Faults
Baseline	55.2	29+331	141+60	0
New Gradient	38.1	29+448	112+40	0
Single Forward	21.2	31+1096	15+25	0
Compiled	17.9	36+574	93+26	36
Cmpld Sgl Fwd	10.2	38+841	8+14	35

Table 1: Speed Enhancements

The user benchmark time is seen to decrease for the improvements listed from 55.2 seconds for the baseline to 10.2 seconds for the best version tested.

The new gradient computation is the source of an improvement of 30% in the user time, from 55.2 to 38.1 seconds. This improvement is a function of MAXTAU, and would be increasingly significant with larger values of MAXTAU.

It can be seen that the compilation process results in a speed-up of a factor of 2 over the uncompiled form, both for the single output array version (38.1 versus 17.9) and the multiple output array version (21.2 versus 10.2).

The use of multiple output arrays, and the consequent single forward pass, contributes nearly another factor of 2 in performance: 1.8 for the uncompiled version (38.1 versus 21.2) and 1.75 for the compiled version (17.9 versus 10.2).

The program size is also seen to vary with simulator version. The use of multiple output arrays is seen to increase the unshared stack size (1096 and 841) relative to the single output array versions (331, 448 and 574).

The use of multiple output arrays has reduced the number of blocks on input by a factor of 7.5 for the standard version (15 versus 112), and 11.6 for the meta-compiled version (8 versus 93). This corresponds approximately to the number of iterations in the experiments. The limit to this improvement is reached when the program exceeds the available memory. Future versions of the simulator may allocate memory dynamically to take maximum advantage of the space/time tradeoff.

It is noted that the number of page faults is zero for noncompiled versions and nonzero for the compiled versions. The reason for this difference is unknown at present.

The conclusion of this comparison is that the improved gradient computation, which is also correct and complete, has significantly reduced the computation time for optimization. The use of multiple output arrays and generation of optimized C-code using a network meta-compiler has resulted in an additional factor of nearly 4 in performance.

6 Extensions

Several extensions to GRADSIM would improve its utility. First, it would be desirable to implement dynamic memory allocation for network data structures, and especially for experiment data structures. This would guarantee optimal space/time tradoffs in performance.

Now that the simulator has been written, debugged and used for some time, it could be cleaned up and rewritten. This would be especially important in making it more general purpose. A module should also be written which handles feedforward networks and binary data, for standard problems, such as encoding. This was done at an earlier stage and needs to be updated.

Finally, a graphical interface for the simulator is under development (GRADVIEW) and should be documented in the near future. It is currently running on the SUN 3 workstations.

Acknowledgements

I am thankful for the opportunity to have performed this work with the support of Siemens Research and Technology and with the grant support from the University of Pennsylvania. I have appreciated the encouragement from Wolfgang Feix of Siemens and Dr. Lokendra Shastri at the University of Pennsylvania over the course the development of the simulator. I would like to acknowledge the contributions of Bruce Ladendorf, of Siemens Research and Technology, who wrote the network meta-compiler and in the process discovered a method for complete gradient computation. Dr. Gary Kuhn, of the Institute for Defense Analyses, also contributed a complete gradient method which he developed. I was also helped by the design of the Rochester Simulator and by discussions with Dr. Max Mintz about nonlinear optimization. I would also like to thank Ed Hook, Lola Anacker and Vercell Vance of the staff of the John von Neumann National Computer Center for their assistance, as well as Ira Winston, the system administration wizard at the University of Pennsylvania.

A Line Search

Line search is a method for locating the point of minimum function value along a particular direction of search. This can be stated as a minimization problem as follows:

$$\min_{\alpha>0} f(x_0 + \alpha\sigma) \tag{3}$$

where f is the objective function, x_0 is the initial value, σ is a search direction, and α is a scalar. Line search is a standard component of many minimization algorithms, including steepest descent and higher order algorithms, such as the quasi-Newton methods included in the GRADSIM simulator.

The method for solving a one-dimensional minimization problem plays an important role in the overall minimization strategy. The control structure for a good line search algorithm is moderately complex, due to the need to use limiting values of α and other methods to avoid infinite loops in the search. In addition, the need to prevent numerical problems such as floating point overflow makes careful analysis of the algebra of hte interpolation methods necessary for proper implementation of the algorithm. Consequently, the basis for the line search algorithm is derived and discussed in some detail.

A.1 Cubic Interpolation

Interpolation is used to compute values of α for which the objective function is at a minimum, based on a polynomial approximation to the objective function. The polynomial approximation is based on several sample points and their associated gradients, depending on the order of the polynomial. This section derives the formulae for computing the minimum of a third order polynomial. Second order polynomials are discussed in Section A.2

The derivation proceeds by solving for the points where the first derivative is zero, and then applying the constraint for a minimum that the second derivative be positive.

A.1.1 Problem Definition

Suppose, the objective function, f, is approximated by a cubic as follows:

$$f(\alpha) = b_3\alpha^3 + b_2\alpha^2 + b_1\alpha + b_0 \tag{4}$$

Then, the first derivative is given by:

$$f'(\alpha) = 3b_3\alpha^2 + 2b_2\alpha + b_1 \tag{5}$$

and the second derivative as:

$$f''(\alpha) = 6b_3\alpha + 2b_2 \tag{6}$$

Also suppose that the objective function value and first derivative is known for two values of α, that is α_1 and α_2.

Now, solve for α_{min} such that:

$$f'(\alpha_{min}) = 3b_3\alpha_{min}^2 + 2b_2\alpha_{min} + b_1 = 0$$

under the condition that $f'' > 0$:

$$f''(\alpha_{min}) = 6b_3\alpha_{min} + 2b_2 > 0$$

The quadratic formula is used to solve the first equation, which becomes:

$$\alpha_{min} = \frac{-b_2 \pm \sqrt{b_2^2 - 3b_1b_3}}{3b_3} \tag{7}$$

This solution is invalid for $b_3 = 0$, in which case the function is quadratic, and the solution for this case is developed in Section A.2 below.

It is also assumed that $\alpha_2 > \alpha_1$, which can normally be assured by the order of the search points. If $\alpha_2 = \alpha_1$, the search has collapsed; this condition is discussed in Section A.2.2.

The cubic approximation equation is solved by solving for b_3, b_2 and b_1, and then substituting in the quadratic formula of Equation 7.

A.1.2 Coefficient Solutions

Using the two known function values, b_0 can be eliminated from Equation 4 to obtain b_1 in terms of b_2 and b_3. Note that b_0 can be solved given the other coefficients and the values of f and α at either point.

$$f_1 = b_3\alpha_1^3 + b_2\alpha_1^2 + b_1\alpha_1 + b_0$$
$$f_2 = b_3\alpha_2^3 + b_2\alpha_2^2 + b_1\alpha_2 + b_0$$

Subtracting,

$$(f_2 - f_1) = b_3(\alpha_2^3 - \alpha_1^3) + b_2(\alpha_2^2 - \alpha_1^2) + b_1(\alpha_2 - \alpha_1)$$

Factoring $(\alpha_2 - \alpha_1)$, and dividing, since $\alpha_2 \neq \alpha_1$:

$$b_1 = \left(\frac{f_2 - f_1}{\alpha_2 - \alpha_1}\right) - b_3(\alpha_2^2 + \alpha_2\alpha_1 + \alpha_1^2) - b_2(\alpha_2 + \alpha_1) \tag{8}$$

Similarly, the known first derivatives can be used to eliminate b_1 from Equation 5, solving for b_2 in terms of b_3.

$$f_1' = 3b_3\alpha_1^2 + 2b_2\alpha_1 + b_1$$
$$f_2' = 3b_3\alpha_2^2 + 2b_2\alpha_2 + b_1$$

Subtracting yields:

$$(f_2' - f_1') = 3b_3(\alpha_2^2 - \alpha_1^2) + 2b_2(\alpha_2 - \alpha_1)$$

Again, factoring $(\alpha_2 - \alpha_1)$, and rearranging,

$$2b_2 = \left(\frac{f_2' - f_1'}{\alpha_2 - \alpha_1}\right) - 3b_3(\alpha_2 + \alpha_1) \tag{9}$$

Now, introduce the following definitions for compact notation:

$$\delta = \frac{f_2 - f_1}{\alpha_2 - \alpha_1}$$

$$\xi = \frac{f_2' - f_1'}{\alpha_2 - \alpha_1}$$

Now, Equation 5 is used for known values of α and f; choosing α_1 and f_1':

$$f_1' = 3b_3\alpha_1^2 + 2b_2\alpha_1 + b_1$$

Now, replace b_2 in this equation using Equation 9, to obtain b_1 in terms of b_3:

$$f_1' = 3b_3\alpha_1^2 + (\xi - 3b_3(\alpha_2 + \alpha_1))\alpha_1 + b_1$$

which becomes:

$$b_1 = f_1' + \alpha_1(3b_3\alpha_2 - \xi) \tag{10}$$

Now solve for b_3 substituting Equations 9 and 10 in Equation 8:

$$f_1' + \alpha_1(3b_3\alpha_2 - \xi) = \delta - b_3(\alpha_2^2 + \alpha_2\alpha_1 + \alpha_1^2) - \left(\frac{\xi - 3b_3(\alpha_2 + \alpha_1)}{2}\right)(\alpha_2 + \alpha_1)$$

After some algebra, this yields:

$$b_3 = \frac{f_1' + f_2' - 2\delta}{(\alpha_2 - \alpha_1)^2} \tag{11}$$

Note that the value of this term will determine whether the approximating polynomial is quadratic ($b_3 = 0$) or cubic.

A.1.3 Cubic Radical

Now, solve for the radical of the quadratic form of Equation 7 in terms of b_3:

$$b_2^2 - 3b_1b_3 = \left(\frac{\xi - 3b_3(\alpha_2 + \alpha_1)}{2}\right)^2 - 3b_3(f_1' + \alpha_1(3b_3\alpha_2 - \alpha_1))$$

After some manipulation, this becomes:

$$\frac{1}{4}\xi^2 - \frac{3}{2}b_3(f_1' + f_2') + \frac{9}{4}b_3^2(\alpha_2 - \alpha_1)^2$$

Substituting Equation 11 gives the following result, after some algebra:

$$b_2^2 - 3b_1b_3 = \left(\frac{1}{\alpha_2 - \alpha_1}\right)^2 \left((f_1' + f_2' - 3\delta)^2 - f_1'f_2'\right)$$

Again, for compactness, define:

$$z = f_1' + f_2' - 3\delta$$

and

$$w = \sqrt{z^2 - f_1'f_2'}$$

Thus, the solution to this point becomes:

$$\alpha_{min} = \frac{-b_2 \pm \frac{w}{\alpha_2 - \alpha_1}}{3b_3} \tag{12}$$

In order for the equation to have a real-valued solution, the radical must be positive:

$$(f_1' + f_2' - 3\delta)^2 - f_1'f_2' > 0$$

When this condition is not met, alternative action must be taken to establish a new value for α, based on the search interval limits.

A.1.4 Cubic Substitution

Now, substitute for b_2 in Equation 7 using Equation 9 to obtain:

$$\alpha_{min} = \frac{-\frac{\xi - 3b_3(\alpha_2 + \alpha_1)}{2} \pm \frac{w}{\alpha_2 - \alpha_1}}{3b_3}$$

Finally, substitute for b_3 using Equation 11 to obtain the solution:

$$\alpha_{min} = \alpha_1 + (\alpha_2 - \alpha_1)\left[1 - \frac{f_2' + z \mp w}{3(f_1' + f_2' - 2\delta)}\right] \tag{13}$$

A.1.5 Cubic Curvature

The curvature condition now must be applied to this solution, in order to ensure that

$$f''(\alpha_{min}) = 6b_3\alpha_{min} + 2b_2 > 0$$

which reduces to the condition:

$$3b_3\alpha_{min} + b_2 > 0$$

First, substitute for α_{min} using Equation 12 to obtain:

$$-b_2 \pm \frac{w}{\alpha_2 - \alpha_1} + b_2 > 0$$

Since $(\alpha_2 - \alpha_1) > 0$, this can be reduced to:

$$\pm w > 0$$

Since w is taken to be positive, this condition holds only for $+w$. Thus, Equation 13 becomes:

$$\alpha_{min} = \alpha_1 + (\alpha_2 - \alpha_1)\left[1 - \frac{f_2' + z - w}{3(f_1' + f_2' - 2\delta)}\right] \tag{14}$$

A.2 Quadratic Case

This section discuss the case when b_3 is found to be zero, and the approximating cubic reduces to a quadratic. In this case, Equation 5 becomes:

$$f'(\alpha_{min}) = 2b_2\alpha_{min} + b_1 = 0$$

whence

$$\alpha_{min} = -\frac{b_1}{2b_2} \tag{15}$$

Now, Equation 10 becomes, with $b_3 = 0$:

$$b_1 = f_1' - \alpha_1\xi \tag{16}$$

Also, Equation 9 becomes:

$$2b_2 = \xi \tag{17}$$

By substituting Equations 16 and 17 in Equation 15, the following solution is obtained:

$$\alpha_{min} = \frac{\alpha_1 f_2' - \alpha_2 f_1'}{f_2' - f_1'} \tag{18}$$

Note that this solution is valid only for $f_2' \neq f_1'$; the solution for the exceptional case is discussed in the next section.

A.2.1 Quadratic Curvature Conditions

Now, the conditions on the curvature which make the point of inflection a minimum are that $f''(\alpha_{min}) > 0$. Thus, for $b_3 = 0$, Equation 6 becomes:

$$f''(\alpha_{min}) = 2b_2 > 0 \tag{19}$$

From Equation 17 above, this implies that

$$\frac{f_2' - f_1'}{\alpha_2 - \alpha_1} > 0 \tag{20}$$

Since $\alpha_2 > \alpha_1$, this implies that $f_2' - f_1' > 0$, or

$$f_2' > f_1' \tag{21}$$

It is assumed that $f_1' < 0$, since the search direction is chosen to be downhill. Note that if $f_1' > 0$ this implies that f_2' must be more *positive*, and thus the minimum point is to the *left* of α_1: $\alpha_{min} < \alpha_1$. If $f_1' < 0$, f_2' must not be more negative, and $\alpha_{min} > \alpha_1$.

If $f_2' = f_1'$, the approximation is a straight line; if $f_2' < f_1'$, the quadratic has a *maximum* at the solution point. In both cases, the lack of quadratic convex behavior indicates that the search is not in the neighborhood of a minimum, so some compensatory action is required. A reasonable solution in this case is to choose:

$$\alpha_{min} = \alpha 1 + 2(\alpha 2 - \alpha 1)$$

A.2.2 Interval Collapse

If α_2 approaches α_1, the search interval is diminishing. If this interval decreases below a selected value, the search is considered to have failed, and appropriate action at a higher level needs to be taken. This could involve terminating the optimization procedure, or reorienting the search direction by reinitializing the Hessian.

B GRADSIM on the CYBER 205

This section discusses the implementation of GRADSIM on the CYBER 205, a vector supercomputer, at the John Von Neumann Center in Princeton, NJ. This section includes benchmark performance figures for the CYBER 205, SUN 3/260 and DEC VAX 8650.

The Control Data Corporation CYBER 200 Series Computer is an SIMD vector processing computer, a "large-scale, high-speed, arithmetic-computing system" [1]. The central processing unit includes a scalar processor and a vector processor which may consist of 1,2 or 4 vector pipes. The scalar processor has a 20 nanosecond minor cycle time, and a 256 64-bit word register file. The main memory access time is 80 nanoseconds, and uses either 4K bipolar RAMS (400 series) or 16K MOS RAM (600 series).

Since the simulator is written in C, the decision was made to use the VECTORC compiler [2] for the CYBER 205, rather than recode everything in FORTRAN. It was understood that in the process, vectorization would have to be done explicitly, since VECTORC is not a vectorizing compiler. It was also stated that the resulting performance would still be about 80% of what was obtainable using FORTRAN because of the mismatch between the machine architecture and the programming language C. However, the changeover was expected to be easy using the same source code, and the 20% performance difference appeared to be a small price to pay for avoiding a recoding exercise.

B.1 Scalar Implementation

This section reports the transfer and implementation of a *scalar* version of the simulator. The C source code for GRADSIM was transferred to the DEC VAX 8650 front-end processor using "ftp", and the shell scripts generated for transfering the appropriate files to the CYBER 205 and running compilations using the necessary passwords and job-control directives.

What follows is a list of the problems encountered in getting the simulator running on several simple test problems.

B.1.1 Command Line Arguments

The simulator was configured to solve a standard nonlinear optimization test problem, in order to test the overall structure of the simulator on the simplest case available. The Rosenbrock function was used for this test. The simulator is written in this case to require floating point arguments on the command line, which give the coordinates of initial search point.

The period, however, is a VSOS line terminator. Therefore, the decimal point in the floating point argument must be enclosed in quotes:

```
BFGS_ROSEN "-1.2" "1.00"
```

However, the quotes cause problems for the C-shell which generates the job control file for the CYBER. This requires the following syntax:

```
BFGS_ROSEN \"-1.2\" \"1.00\"
```

B.1.2 External Array Length

The length must be included with the declaration of external arrays in order to use the [*] vector reference format:

```
extern weight[MAX_VECTOR];

x = weight[*] @. q[*];
```

Otherwise, the length is taken as zero. Note that this syntax is different from what would be expected in the C environment, and as far as I know is not documented.

B.1.3 Missing Include Files

Certain 'include' files referenced in the VECTORC manual are used in the connectionist simulator. As it happened, the include files were missing because of a problem with the archiving process which had wiped out part of the pool of 'include' files. This problem was fixed by JVNC staff.

B.1.4 Variable and Function Names

The CYBER uses fixed limits on the length of variable and function names. This generated numerous 'non-unique entry point' errors, which were fixed by renaming functions and variables.

B.1.5 Reading Files

Since the file structure on the CYBER 205 is radically different from that in UNIX environment, file I/O is also affected. As a result, a data file had to be first opened with 'open' with the appropriate flags for the corresponding file types. For the binary speech data files, the flags 'O_RTU' and O_RDONLY' were used. The 'O_RTU' flag specifies a VSOS U record type, which is treated as an unformatted byte stream. The 'O_RDONLY' flag was used to prevent unexpected writes to the data files from having any effect.

```
if ((fd=open(filename,O_RDONLY | O_RTU,0)) != -1)
fp = fdopen(fd,"r");
```

The binary data is then read by requesting the maximum number of samples, and using the actual number read as the size of the file:

```
count = fread(input,ByteCount,MAX_DURATION,fp);
if (count == MAX_DURATION)
        {
        fprintf(stderr,"Read Max Count - %d\n",count);
        exit(0);
        }
fclose(fp);
```

However, if the data file is not an exact multiple of 'ByteCount', fread returns MAX_DURATION, causing a fatal error and exit. This appears to be an artifact of the CYBER file I/O, and does not conform to standard C usage.

Standard ASCII files, such as the network and experiment descriptors can be read with simple 'fopen' calls, as in the original C code.

B.1.6 Writing Files

File names on the CYBER are expected to be upper case, and without extensions. Using 'fopen' to create a file called 'net.tmp' resulted in an error, whereas 'nettmp' is acceptable. This is somewhat unexpected, since the VECTORC manual [2] states that files are *read* ignoring the dot:

> For example, <sys/file.h> really reads as FILEH, "my_header_file" really reads as MY-HEADER, and so forth.

B.1.7 Data Files

Binary speech data files were transferred to the CYBER using the 'MFLINK' "GET" utility, and given VSOS file type 'U', which is unformatted binary data. The files were then made permanent on the CYBER for future reference in experiments using the 'DEFINE' utility. The following fragment illustrates this method:

```
PURGE,CVOORW11.
MFLINK(CVOORW11,ST=VA6,DD=UU,JCS=\"USER watrous $unix_passwd\",
   \"GET /usr/user4/watrous/cvx/cv00rw11.eng\")
SWITCH,CVOORW11,RT=U.
DEFINE,CVOORW11.
```

B.1.8 Memory Allocation

The declaration of global two-dimensional arrays was problematic. The original module contained such a data structure for the unit outputs, and would compile successfully by itself. When compiled in conjunction with the rest of the simulator, the META assembler would generate the following error:

```
-THE MAXIMUM LENGTH OF A MSEC IS REACHED
-ASSEMBLY ABORTED IN PASS 2
 LAST STATEMENT NUMBER   1/0035


  11.48.24 CC,IN=SPEECHC,BPC,READC,WEIGHTC,UNITC,MAPC,GAUSSC,
  11.48.24 LIB=LIBM,CN=BPS,LINK=M.
  11.48.26  CPP SPEECHC Q6CP2995
  11.48.38  CCOM -N SPEECHC Q6CP2995 Q6CC2959
  11.49.00  META,INPUT=Q6CC2959,BINARY=BINARY
  11.49.03  META 2.3 670B
  11.49.06  ** ASSEMBLY ABORTED **
  11.49.06         1 MODULES ASSEMBLED
  11.49.06      .410 SECONDS ASSEMBLY TIME
  11.49.06 Fatal META error(s)
```

Another technique was suggested in which the array was allocated dynamically, using 'malloc'. This got through the compilation phase, but resulted in a crash at execution time. This created an impasse, at which point the effort was abandoned for several months.

B.1.9 New C Compiler

When the effort to use the CYBER 205 was resumed, a new experimental VECTORC compiler was made available, which had the effect that the memory allocation problem, and a problem with the scalar exponential function were solved, and the simulator ran for the first time on speech data.

B.2 Conclusions on Simulator Transfer

Several conclusions may be drawn from the experience of porting the GRADSIM connectionist network simulator to the CYBER 205. These conclusions are based on experiencing the consequences of the differences in hardware and software between VECTORC running under VSOS on the CYBER 205 and C running under UNIX on DEC VAXes and SUN workstations.

C Compiler The problems with the earlier version of the VECTORC compiler were serious enough to preclude the possibility of generating operable software. The conclusion from this fact is that the original version of VECTORC was not complete or correct, and *should not have been offered for use as a working compiler.* Discovering the inadequacy of the VECTORC compiler consumed several weeks of effort, which was largely wasted.

Software Development Effort Because of the differences between C and VECTORC, such as the VECTORC restrictions on name lengths, VSOS constraints on file names and differences in file I/O, the versions of the simulator which will run under VECTORC are significantly different from the original C code. There is also presently no tools to automate conversion from C to VECTORC. Thus, the effort to make the transition from C to VECTORC is considerable, and requires programmer expertise and direct code modification. Thus, it is difficult to maintain version compatibility between simulator code for UNIX machines and the CYBER 205. This has the effect of doubling the software maintenance effort.

The first conclusion from this observation is the conversion from UNIX C to VSOS VECTORC should be done as few times as possible. Consequently, it is best that only stable code should be transfered to a vector machine; code that is under development should be completed in one environment before being transferred to another. However, this advice may be difficult to observe in practice; the motivation for using the supercomputer to begin with may be to get results more rapidly in an exploratory stage of a scientific research project where the simulation tool may be modified in response to experimental results.

Secondly, the software development effort needs to be amortized over sufficiently long usage time in order for the speed-ups achievable at supercomputer performance rates to make it worth the effort.

Software Development Costs VECTORC does not run on the VAX 8650 under UNIX; all software development had to be done on the CYBER 205 itself. Thus, the cost for the discovery and correction of all of the problems encountered was charged at CYBER 205 rates, which was expensive (approximately $9,000).

Code Optimization The VECTORC compiler is not a vectorizing compiler. In order to obtain supercomputer performance from the CYBER 205, the software has to be explicitly vectorized. In addition, the scalar code should also be optimized for performance. These optimization efforts require knowledge of the VECTORC compiler and the CYBER 205. This requires time and effort, which is impacted by the software development effort in porting an application to the VSOS environment. The conclusion of this observation is that additional code vectorization and optimization time must be allocated in estimating the performance tradeoffs of using the supercomputer. In the present case, the software development effort consumed so much time relative to the available time for the project that insufficient time was left for vectorization and scalar optimization

B.3 Vectorization

Vectorization of the GRADSIM code was carried out in reference to the vector operations and algorithmic structures [9]. The vectorization of the GRADSIM simulator was carried out with respect to the vector operations resulting from an analysis of the optimization algorithms considering one token at a time. This was done for two reasons; first, it was easier since this required less extensive code modifications, and second, because the benchmark which was being used to evaluate performance on the CYBER 205 consisted of only two tokens. It was discovered that the problem does not vectorize well under this approach, since the resulting vectors are small, and close to the point where the overhead of vector setup time outweighs the efficiency of the vector processor.

Restructuring the algorithm for multiple tokens would result in long vectors where the vector efficiency would be fully achievable. However, this would require care in dealing with arbitrary numbers of tokens, in which case the maximum vector length could be exceeded; this would result in code necessary to pack tokens appropriately for maximum length vectors.

Restructuring of the algorithm required to exploit the vector processing of the CYBER 205 requires expertise, time and software modifications. This aspect of the implementation of the GRADSIM package on the CYBER 205 generates and strengthens many of the same conclusions listed above under Software Development Effort.

B.4 Performance

This section describes a benchmark comparison between the CYBER 205 , DEC VAX 8650 and Sun 3/260. The problem is described, and the performance figures given and analyzed. This leads to a decision regarding use of the CYBER 205.

B.4.1 Benchmark

The benchmark problem was taken as a network optimization problem described in Section 5 above, in which a temporal flow model was trained to distinguish the word pair "no/go" [21]. It should be noted that the CYBER version of the simulator does not include the random perturbation of the network at initialization. This is done by the front-end VAX, and the simulation begins with a pre-randomized network descriptor.

B.4.2 Results

The optimization algorithm was run for 10 iterations on each of three machines, and timed on the UNIX machines using the "time" utility.

Sun 3/260	VAX 8650	CYBER Scalar	CYBER Vector
124.5	72.3	76.7	49.4

Table 2: Benchmark Execution Times in Seconds

The CYBER 205 scalar processor is described as a 50 MIPS machine (20 ns cycle) and had been promoted as giving a minimum of 8 times the performance of the DEC VAX 8650. This benchmark shows that the CYBER 205 scalar processor, which was described subsequent to this experiment as a 20 MIPS machine, is no faster than the DEC VAX 8650. The reason for this discrepancy must lie in the VECTORC compiler. The Sun 3/260 with FPA, a 4 MIPS machine, runs in approximately 70% more time than the 8650. The vectorized version requires 68% of the time of the 8650, for a speed-up of nearly 50%.

A second benchmark was run on the CYBER for 50 instead of 10 iterations, in order to estimate the amount of initialization cost. In this case, the 10 iteration version ran in 83.1 seconds, and required 368.9 seconds for 50 iterations. This corresponds to 7.145 seconds per iteration, and implies an initialization time of 11.65 seconds.

It is noted that vectorization must be complete in order to achieve full vector processor rates. Thus, if an application has a non-vectorizable residue which consumes 10% of the processing time, then the maximum speed-up using the vector processor is a factor of 10.

B.5 Conclusions on Performance

The conclusions from this benchmark are that

1. The scalar processor is effectively no faster than the VAX 8650, and only 40% faster than the SUN 3/260. Since the use of the scalar processor is charged at full vector processor rates, it should only be used if it is less expensive than the VAX 8650, or less than 65% more expensive than the SUN 3/260.

2. The GRADSIM package does not vectorize well as structured for one token at a time.

3. The performance improvements due to optimization of the GRADSIM C code and the network compiler are such that use of the CYBER 205 should be discontinued for the present time.

References

[1] *CDC CYBER 200 MODEL 205 COMPUTER SYSTEM.* Control Data Corporation, Minneapolis, MI, 1984.

[2] *CYBER 200 VECTORC.* Control Data Corporation, Minneapolis, MI, July 1986.

[3] C. G. Broyden. The convergence of a class of double-rank minimization algorithms. *J. Inst. Maths Applics*, 6:76–90,222–231, 1970.

[4] William C. Davidon. *Variable Metric Methods for Minimization.* AEC Research and Development Report ANL-5990 Rev, Argonne National Laboratories, November 1959.

[5] Mark Fanty. *Connectionist Network Simulation Tools.* University of Rochester, 1985.

[6] Roger Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, August 1970.

[7] Roger Fletcher. *Practical Methods of Optimization.* Volume 1 Unconstrained Optimization, John Wiley & Sons, 1980.

[8] Roger Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 6:163–168, 1963.

[9] M. J. Kascic Jr. *VECTOR PROCESSING ON THE CYBER 200.* Technical Report, Control Data Corporation, Minneapolis, MI, 1979.

[10] Gary Kuhn. *A First Look at Phonetic Discrimination using a Connectionist Network with Recurrent Links.* Technical Report SCIMP Working Paper No. 4/87, Institute for Defense Analyses, Communications Research Division, September 1987. IDA-CRD Log No. 82018.

[11] David G. Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley, second edition, 1984.

[12] David B. Parker. *Learning-Logic.* Center for Computational Research in Economics and Management Science TR-47, Massachusetts Institute of Technology, 1985.

[13] David E. Rumelhart, Goeffrey Hinton, and Ronald Williams. Learning internal representations by error propagation. In J.L.McClelland D.E.Rumelhart and the PDP research group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Volume I Foundations*, chapter 8, MIT Press, Cambridge, MA, 1986.

[14] David E. Rumelhart, Goeffrey Hinton, and Ronald Williams. *Learning Internal Representations by Error Propagation.* Institute for Cognitive Science ICS Report 8506, University of California, San Diego, September 1985.

[15] Ya. Z. Tsypkin. *Adaptation and Learning in Automatic Systems.* Volume 73 of *Mathematics in Science and Engineering*, Academic Press, 1971. Translated by Z. J. Nikolic.

[16] Raymond L. Watrous. *Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization.* Technical Report MS-CIS-87-51, University of Pennsylvania, June 1987.

[17] Raymond L. Watrous. Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization. In *Proceedings of the First International Conference on Neural Networks*, pages 619–627, June 1987.

[18] Raymond L. Watrous, Gary Kuhn, and Bruce Ladendorf. Complete gradient optimization of a recurrent network applied to /b/, /d/, /g/ discrimination. In *115th Meeting of the Acoustical Society of America*, May 1988. Abstract only - submitted.

[19] Raymond L. Watrous and Lokendra Shastri. Learning acoustic features from speech data using connectionist networks. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, pages 518–530, July 1987.

[20] Raymond L. Watrous and Lokendra Shastri. Learning phonetic features using connectionist networks: an experiment in speech recognition. In *Proceedings of the First International Conference on Neural Networks*, pages 381–388, June 1987.

[21] Raymond L. Watrous and Lokendra Shastri. *Learning Phonetic Features Using Connectionist Networks: An Experiment in Speech Recognition.* Technical Report MS-CIS-86-78, University of Pennsylvania, October 1986.

[22] Raymond L. Watrous, Lokendra Shastri, and Alex Waibel. Learned phonetic discrimination using connectionist networks. In *European Conference on Speech Technology*, pages 377–380, September 1987.

[23] Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.