GRAD-CM2: A Data-Parallel Connectionist Network Simulator*

Thomas Fontaine
Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389

Abstract

A data-parallel simulator capable of training recurrent time-delay connectionist networks is described. The simulator, GRAD-CM2, is written in the C* programming language and runs on a Connection Machine CM-2. GRAD-CM2 is an extension of the serial simulator, GRADSIM [8], and offers similar features. Timing performances of GRAD-CM2 and GRADSIM on a series of spatiotemporal discrimination tasks are presented to emphasize the efficacy of data-parallelism.

1 Introduction

Although the realization that the error gradient of a multi-layer network could be computed (popularized by [4]) led to the application of a variety of powerful gradient minimization methods [7], the training of connectionist models in certain domains remains painstakingly slow. As the scope and complexity of connectionist applications expand, there is an increased demand for faster training methods.

The time required to train a network is dependent upon several factors including the algorithm used, the network topology, and the initial network state. Once these factors are prescribed, the time required to train a network using a deterministic algorithm is a function of the training set. In some applications, a sparse data set may yield the desired result. In many applications, however, particularly those which require the formation of complex decision surfaces, the usage of a massive amount of training data is desirable [3]. Unfortunately, the time complexity of true gradient methods, implemented on a serial machine, is linear with the number of training examples.

It is possible, however, to process all training examples concurrently using a data-parallel approach (see, for example, [5]). A data-parallel simulator, capable of training recurrent

*This work was supported by grant MCS-83-05211 and ARO grants DAA29-84-9-0027 and DAAL03-89-C-0031.

time-delay networks, is described. The simulator, GRAD-CM2, is an extension of the serial simulator GRADSIM [8] and offers similar features, including several classical optimization algorithms: fixed-step descent, conjugate gradient, and pseudo-Newtonian (BFGS). The simulator is written in the C* programming language [6] and runs on the Connection Machine model CM-2.

Section 2 provides an overview of the simulator. Section 3 surveys the C* modules and compilation options which comprise GRAD-CM2. Timing performances are reported in Section 4, followed by final remarks in Section 5.

2 Overview

GRAD-CM2 was developed for rapid network optimization, particularly in applications involving a large number of training examples. The simulator is capable of training networks which utilize arbitrary connectivity (including recurrencies), time-delay links, sigmoidal or linear units, and fixed or variable weights. GRAD-CM2 was written primarily to train networks applied to spatiotemporal recognition problems, but static (*feed-forward*) networks can be trained as well.

2.1 Parallel Algorithms

During network training, the objective value and error gradient of a network is computed by summing the results from individual examples. GRAD-CM2 differs from serial simulators since it computes the objective value and error gradient on all training examples in parallel. Figure 1 outlines the steps taken from the onset of training to the computation of one objective value (several implementation details are omitted for clarity).

A similar scheme is used to compute the gradient of the network, whereby another two dimensional matrix is made available to each active processor to store the output unit error as it is backpropagated through time. The partials of the gradient are accumulated from all processors at each time step of the backpropagation and summed on the front end to form the complete gradient.

Other parallel computations are made by GRAD-CM2, such as the scoring of the current network state at each iteration. These computations are all of the same flavor as the objective value and gradient computation described above (relying on data-parallelism).

2.2 Hardware Limitations

GRAD-CM2 is limited by the hardware on which it runs. If N physical processors are available, and over half the memory of each physical processor is needed to store the activation matrices, then a network can be trained on up to N examples. If only an integer fraction of the memory of each processor is needed, say $1/s$, then a network may be trained on up to $s \times N$ examples (using virtual CM-2 processors).

- Store the network topology and state on the front end machine.
- Use a number of virtual CM-2 processors equal to the number of training examples.
- For each processor, allocate a two dimensional array of floats (to store a history of all unit activations over time), an array of floats (to store the desired response of the output units), and a single float (to store an error measure).
- Load each training example into the appropriate portion of the activation array in its corresponding processor. Similarly, load the target functions.
- To compute the objective value of the network:

    1. Set t, representing time, to 0.
    2. Consider all processors where t is less than the duration of the stored example to be active. While there are active processors:
    3. For each link in the network stored on the front end machine:
    4. Broadcast the link to the active processors.
    5. Update the activation arrays in the active processors (in parallel), simulating activation being spread across the link.
    6. Increment t.
    7. Reset t to 0 and set the error variable in all processors to 0.
    8. Consider all processors where t is less than the duration of the stored example to be active. While there are active processors:
    9. In active processors, compute the output unit error at time t, and add the error to the error variable.
    10. Increment t.
    11. Sum the error variables over all processors to derive the objective value on the front end machine.

Figure 1: Outline of steps taken by GRAD-CM2 to calculate the objective value of the network over all examples.

In some cases, the memory of an individual processor may not be large enough to accommodate the activation matrices for both the forward and backward passes (individual CM-2 processors do not offer virtual memory). GRAD-CM2 provides a compiler option, **SQUEEZE_MEMORY**, which avoids the allocation of the backpropagation matrix and instead uses a smaller buffer[1] and a data-buffering scheme (see Section 3.5).

If the memory needed to store the forward pass activation matrix and the small buffer, mentioned above, exceeds the physical memory limit of an individual processor, then the current version of GRAD-CM2 cannot be used.

2.2.1 An Example

Say the CM to be used offers 256k bits per physical processor (the CM-2 is bit-addressable), and that the maximum temporal duration of the training examples is 32. Approximately how large of a network can be trained under these constraints, without setting the **SQUEEZE_MEMORY** flag?

An activation matrix is of size MAX_NUM_UNITS by MAX_DURATION, where MAX_NUM_UNITS is greater than or equal to the number of units in the network to be trained (including input and threshold), and MAX_DURATION is taken to be 32 in this case. Since two activation matrices are needed when **SQUEEZE_MEMORY** is not defined, a total of 2 $\times$ MAX_NUM_UNITS $\times$ 32 floats are needed. Assuming four bytes per float, 2048 $\times$ MAX_NUM_UNITS *bits* are needed. Since 256k bits are available, MAX_NUM_UNITS can be around 128 (some memory is needed to store the target function and other variables for each example).

2.3 Using GRAD-CM2

To train a network using GRAD-CM2 on a Connection Machine, the following must be created:

- A GRAD-CM2 executable (built using **cs**, the C* compiler).
- A *network descriptor*, which specifies the network topology and initial network state.
- An *experiment descriptor*, which enumerates the files containing the training examples and prescribes the desired network response to each training example.

Additional descriptors, such as a cross-validation descriptor, may be specified but are not required. The remainder of this section describes the format of the various descriptors used by GRAD-CM2. Details involving building a GRAD-CM2 executable are found in Section 3.

[1]The buffer is a MAX_NUM_UNITS by MAX_DELAY matrix of floats, where MAX_NUM_UNITS is greater than or equal to the number of network units and MAX_DELAY is greater than or equal to the maximum delay along any link in the network.

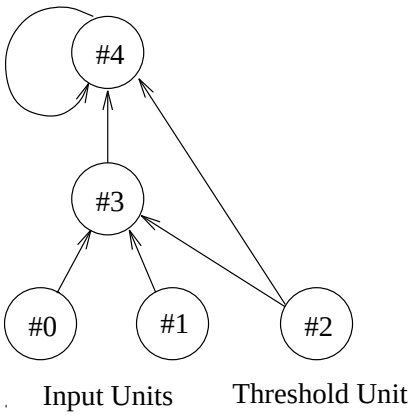

Figure 2: An example network

2.3.1 Network Descriptor

In order to specify a network topology, a unique identifier (in the form of an integer) must be assigned to each unit. GRAD-CM2 expects the I input units of the network to be assigned the identifiers $0 \ldots (I-1)$. Although it is not necessary to sequentially number the remaining units, the practice should be followed in order to minimize the amount of memory used. GRAD-CM2 requires each network to utilize a threshold unit, and it is notationally convenient to assign this unit the identifier I. Figure 2 depicts a network using two input units (labeled 0 and 1), a threshold unit (2), and two other units (3 and 4).

Once identifiers are assigned, the network descriptor can be created. The descriptor for a network with L links is a text file consisting of $L+1$ lines. The first line specifies the identifier associated with the threshold unit. Each of the following lines describes a separate network link, using four values: the unit from which the link emanates (integer), the receiving unit (integer), the associated delay (integer), and an initial weight (floating point). For example, assuming that all the links in the network depicted in Figure 2 have a unit delay, except for the link from Unit 1 to Unit 3 which has, say, a delay of two, the network descriptor (with random initial weights between -1.0 to 1.0) might look as follows:

```
Threshold Unit 2
0       3       1       0.101
1       3       2      -0.438
3       4       1       0.292
4       4       1       0.272
2       3       1       0.796
2       4       1      -0.911
```

After completing each training iteration, GRAD-CM2 writes a new network descriptor to the file **net.tmp.tmp**. The topology specified by **net.tmp.tmp** will remain consistent with the initial network descriptor, but the weights will be updated (therefore, **net.tmp.tmp** will always contain the most recent state of the network). If a training run is interrupted

for some reason, training may be restarted where interrupted by using **net.tmp.tmp** as the new initial network descriptor.

2.3.2 Experiment Descriptor

The *experiment descriptor* specifies the desired response of a set of units to a set of training examples. If N training examples are to be used, the experiment descriptor is a text file of $N + 1$ lines. The first line specifies the number of output units, O, followed by the integer identifier assigned to each of the output units. Each subsequent line of the experiment descriptor contains:

- the name of one training example file, followed by
- a sequence of $+$ or $-$ signs (totaling O in number), specifying how each output unit should respond to the training example (either positively or negatively), and
- two parameters, XI (real, between 0 and 1) and MAG (real, positive) which are used to describe the desired response, over time, of the output units for problems involving a temporal dimension. XI and MAG are not used in static problems, although dummy values must be specified.

For example, if the network depicted in Figure 2 is to be trained to discriminate between objects of type A and B (such that Unit 4 responds to examples of type A but not type B), a typical training descriptor might look like:

```
1               4
A.example1      +       0.5 10.0
A.example2      +       0.5 10.0
B.example1      -       0.5 10.0
B.example2      -       0.5 10.0
```

If it were desired that a similar network with an additional output unit, Unit 5, be trained such that Unit 4 responds to A examples but not B examples, and that Unit 5 respond to B examples but not A examples, the descriptor might look like:

```
2               4       5
A.example1      +       -       0.5 10.0
A.example2      +       -       0.5 10.0
B.example1      -       +       0.5 10.0
B.example2      -       +       0.5 10.0
```

where [AB].example[12] are names of training example files, each of which contains data representing one training example.

By default, GRAD-CM2 expects a training example file to be a text file consisting of a number of lines (one per time sample), where each line is an uninterrupted string of ASCII

0's and 1's (one ASCII character per input unit). Although this format is incapable of handling multi-valued inputs, GRAD-CM2 can be compiled to expect the training files to be in a different format, representing multi-valued inputs (see Section 3.5).

2.3.3 Unit Type Descriptor

If the use of linear units is desired, an additional descriptor must be provided to distinguish sigmoidal units from linear units. A *unit type descriptor* is a text file containing a number of lines equal to the number of units in the network which *receive* input. Each line contains two integers: a unit identifier (consistent with the identifiers in the network descriptor), and either a 0 (sigmoid unit), or a 1 (linear unit). For example, if it were desired that Unit 3 from Figure 2 be sigmoidal and Unit 4 be linear, the unit type descriptor would look as follows:

```
3 0
4 1
```

Note that the identifiers for the input units and the threshold unit do not appear in the descriptor, since they do not receive input.

2.3.4 Weight Mask Descriptor

In some applications, it may be desirable to freeze a number of weights while allowing other weights in the network to be variable. A *weight mask descriptor* is a text file which is of the same form as a network descriptor, except that the field specifying the link's weight contains either a 0 (frozen weight), or a 1 (variable weight).

2.3.5 Cross-Validation Descriptor

A *cross-validation descriptor* has the same form as a training descriptor. The example files specified in a cross-validation descriptor will have no effect on the optimization, but the objective value on the examples will be reported.

3 The GRAD-CM2 Package

GRAD-CM2 is comprised of a number of modules written in the C* programming language. The simulator can be obtained via anonymous ftp from **linc.cis.upenn.edu** (as can GRAD-SIM). The following subsections present descriptions of the modules comprising GRAD-CM2, and give an example of building and running the simulator.

3.1 Header Files

Three **.h** files control various facets of the optimization. The constants defined in the files should be set in accordance with the network descriptor and the experiment descriptor. *Anomalous behavior may occur if the header files are not consistent with the descriptor files.*

csnet.h Contains parameters and supporting structures specific to the network to be trained.

csexamples.h Contains parameters and structures which control aspects of input and propagation of the training examples.

cscharacter.h GRAD-CM2 was written primarily to serve the needs of research involving character recognition using recurrent time-delay networks (see, for example, [2]). This file controls the type of "scans" which will be used during optimization. If the **CHARACTER** compilation flag (see Section 3.5) is not set, **cscharacter.h** has no effect on the optimization. Most applications will not require this file.

3.2 Optimization Modules

GRAD-CM2 offers three optimization algorithms: fixed-step descent (or "backprop"), conjugate gradient, and BFGS (see, for example, [1][8]). Each optimization method has a corresponding C* module (suffixed with a **.cs**):

bp.cs Contains the fixed-step descent algorithm. The companion makefile is **bp.mkf**.

conjugate.cs The **conj.mkf** makefile is used to compile this module to perform optimization using the conjugate gradient method.

bfgs.cs The BFGS method can be used by compiling GRAD-CM2 with the **bfgs.mkf** makefile.

A vital component of the conjugate gradient and BFGS methods of optimization is the one dimensional line search contained in the module **line_search.cs**. The line search, by itself, does not constitute a general optimization algorithm and hence does not have a corresponding makefile.

3.3 Target Function Modules

In order to train a network, the algorithm must be provided with the desired network response. Although the experiment descriptor provides the information regarding whether an output unit should be active or inactive (+ or −), it does not specify *how* active the units should be and *when*. The modules which provide this information are of the form ***_targ.cs**. For temporal problems, the desired response may vary over time and hence a *target function* is necessary. GRAD-CM2 offers three temporal target function modules:

gauss_targ.cs The target is Gaussian over time, centered at XI (assuming the example is processed over a unit time interval), with a deviation governed by MAG.

sig_targ.cs The target is sigmoidal over time, with the inflection point at XI and a rate of increase determined by MAG.

linear_targ.cs A linear target is used and XI and MAG have no effect.

The target values at the onset and the termination of assimilation of an example can be manipulated by altering the COEFFICIENT and OFFSET constants found in the ***_targ.cs** modules.

In the static case, the target module **static_targ.cs** must be used. The POSITIVE_TARGET constant defined in **static_targ.cs** may be set to the desired positive target value, and the desired negative response is taken to be 1.0-POSITIVE_TARGET.

3.4 Support Modules

Several modules, listed here in alphabetical order, support the optimization:

apply.cs Supports initialization, objective function calculation, and gradient computation.

map.cs Maps network linked list structures to vector format and vice-versa.

perturb.cs Perturbs the initial network weights.

read.cs Reads training and cross-validation descriptors.

score.cs Scores the network on the training and cross validation sets at each iteration.

unit.cs Performs forward and backward activation passes.

util.cs Contains miscellaneous utility routines, such as vector manipulation.

weight.cs Reads and stores the network descriptor.

3.5 Compiling GRAD-CM2

GRAD-CM2 must be compiled on a machine offering **cs**, the C* programming language compiler. GRAD-CM2 offers flexibility through several compilation flags:

ASYMMETRIC The default format for a positive target function is to increase from a target of 0.5 at the onset of the example to 0.95 by the end. The default negative target is the positive subtracted from 1.0. If this flag is set, the positive target function will increase from 0.05 at the onset to 0.95 by the end, while the negative target function will remain 0.05 over the duration of the example (recall that these values may be altered by changing the COEFFICIENT and OFFSET constants in the ***_targ.cs** modules).

BYTE_INPUT_FORMAT GRAD-CM2 expects binary data by default. A training example consists of a number of lines (one per time sample), where each line consists of consecutive ASCII 0's and 1's (one character per input unit). Although this format is simple to use, it is inefficient and incapable of handling multi-valued inputs. If the **BYTE_INPUT_FORMAT** option is specified, an example file is expected to consist of an uninterrupted string of bytes, with each byte representing a real input value in the range [0,1] (thus, 255 real values may be specified). Assuming that the network has I input units and an example spans T time steps, the example should contain $I \times T$ bytes.

CHARACTER Specifies that the examples should be "scanned" over time in directions specified in **cscharacter.h** (see Section 3.1).

CROSS_VALIDATION An extra training descriptor representing a cross-validation set is expected if this flag is specified and GRAD-CM2 will report progress on this set as well as the training set.

EVALUATE If it is desired that a network be tested on a set of examples instead of trained, this flag should be defined. Execution will halt after the first objective function evaluation and GRAD-CM2 will produce a file, **score.errors**, containing the names of the examples in error.

LMS If set, the objective value will be normalized by the number of training examples and output units.

MASKED If it is desired that weights on certain links not be altered, this flag must be set. A network mask descriptor to distinguish frozen weights from variable weights is expected on the command line.

MULTI_UNIT_TYPE The default unit output function is sigmoidal. If this flag is defined, linear units may also be employed. A unit type descriptor distinguishing sigmoidal units from linear units is expected on the command line.

STATIC This flag must be set for static recognition problems (involving links with 0 delays). **STATIC** is incompatible with several other flags such as **ASYMMETRIC** and **CHARACTER**.

SCORE Instructs GRAD-CM2 to score the network on the training set (and cross-validation, if **CROSS-VALIDATION** is defined) at each iteration. Scoring is performed by making one classification hypothesis for each output unit and checking if the hypothesis which generates the least error matches the actual classification (defined in the training descriptor). The hypothesis error for an output unit is computed by assuming that the unit should respond positively and all other output units should respond negatively (thus, the **SCORE** flag is only appropriate for problems in which exactly one output unit should respond positively to a given example).

SQUEEZE_MEMORY Since individual CM-2 processors do not offer virtual memory, GRAD-CM2 is constrained to handle problems of a limited size. This flag makes more space available by buffering data, but slows the optimization considerably.

TERMIN Causes the state of the network to be saved at certain objective value milestones. The milestones can be set by altering the NUM_GOALS and BASE_GOAL constants defined in **csnet.h**.

After selecting the optimization algorithm to be used, the corresponding makefile for the algorithm should be edited to include the desired target function module. For example, if it is desired that the BFGS algorithm be used with a sigmoidal target function, **bfgs.mkf** should be edited to include **sig_targ.o**. A typical makefile specifying the usage of a mean squared error, an asymmetric sigmoidal target function, a cross-validation set, network saves at milestones, and scoring at each iteration might look like:

```
CFLAGS = -O -DLMS -DASYMMETRIC -DCROSS_VALIDATION -DTERMIN -DSCORE
CS = cs
CSFLAGS = $(CFLAGS)
.SUFFIXES: .cs
.cs.o:
        $(CS) -c $(CSFLAGS) $<

bfgs: bfgs.o weight.o apply.o map.o read.o perturb.o sig_targ.o \
unit.o line_search.o score.o util.o
        cs ${CFLAGS} -o bfgs bfgs.o weight.o apply.o map.o read.o \
perturb.o sig_targ.o unit.o line_search.o score.o util.o \
-lm -L/usr/include/cs/6.0.1
```

3.6 Running GRAD-CM2

Assuming GRAD-CM2 has been built with the makefile above, training could begin by issuing the following command line:

```
cmattach bfgs [-c [-r seed]] net_desc train_desc cross_desc
```

where net_desc is a network descriptor, train_desc is a training descriptor, and cross_desc is a cross-validation descriptor.

The first argument is optional. If not specified, each weight will be perturbed by .1% of its original value. The -c flag will prevent this perturbation, while the [-r seed] option will assign each weight an initial value between -1.0 and 1.0.

3.7 GRAD-CM2 Output

GRAD-CM2 will write useful information to the standard output at each iteration of the optimization: the current iteration number, the total number of objective function calculations, the total number of gradient calculations, the objective value, and the norm of the network. If the **SCORE** flag is set during compilation, the accuracy of the network on the training set is reported at each iteration. If the **CROSS_VALIDATION** flag is set, information regarding the cross-validation set is also reported.

The network state at the onset of optimization is written to the file **net.testio.fin**. Upon completion of each iteration, the current state of the network is written to the file **net.tmp.tmp**. Upon termination of the optimization (due to either success or failure), the network state is written to **net.x.fin**.

4 Timing Performances

4.1 Problem Set

Timing performances were derived by using GRAD-CM2 and GRADSIM to train three separate recurrent time-delay networks on four different data sets. The common task was to discriminate between handprinted samples of the digits "0" and "1" (for details on applying recurrent time-delay networks to character recognition, see [2]).

The three networks, Net1, Net2, and Net3, shared a similar topology. Each consisted of 20 input units and a single output unit and differed in the number of hidden units used (4 for Net1, 20 for Net2, and 40 for Net3). The networks were completely connected between layers, with delay links of 1 between the input and hidden layer, and delay links of 1, 3, and 5 between the hidden and output layer. Self-recurrent links were placed on all hidden units and the output units. Since each link in a network corresponds to a variable in an optimization process, the networks were constructed to provide a simple gradation in the number of links: Net1 consisted of 102 links, Net2 used 502, and Net3 employed 1002.

The four training sets contained preprocessed samples of handprinted "0" and "1" images, drawn from the "United States Postal Service Office of Advanced Technology Handwritten ZIP Code Database (1987)." The first set consisted of 10 examples, the second 100, the third 500, and the fourth 1000.

Each combination of network and training set was timed using GRAD-CM2 (running on a Connection Machine CM-2 with a Sun-4 front end) and GRADSIM (running on an IBM RS/6000 Model 320, a very fast serial machine).

4.2 Results

Figure 3 depicts the timing results. The horizontal axis represents the number of training examples used, with tick marks at 10, 100, 500, and 1000, corresponding to the four training sets. The vertical axis depicts the actual time, in minutes, of the fixed-step descent (back-

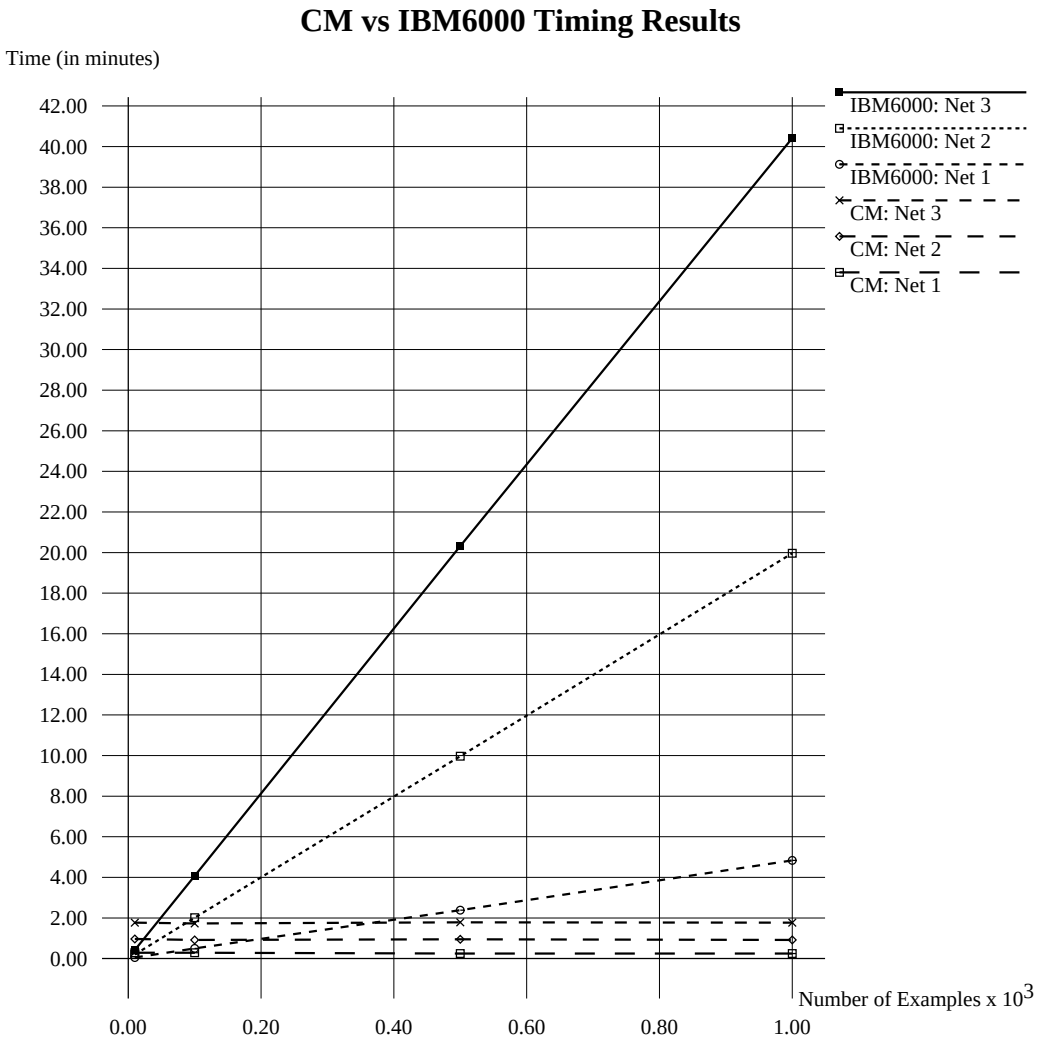

Figure 3: Timing results of the backpropagation algorithm on the CM and an IBM RS/6000. The horizontal axis represents the number of training examples, while the vertical axis demarcates the time taken, in minutes, to process ten backpropagation iterations. Each line on the graph represents the timing results of Net 1, Net 2, or Net 3, on either the CM or IBM RS/6000.

propagation) algorithm to complete *ten* iterations. Ten iterations were used to provide an average measure of performance. As expected, the performance times of GRAD-CM2 are independent of the number of training examples.[2]

The *speed-up* of GRAD-CM2 over serial implementations increases in proportion to the number of training examples used. In addition, the time necessary to train a network is proportional to the complexity of the network, whereas the time needed to train a network using a serial implementation increases as a *multiple* of the complexity of the network and the number of training examples. This savings is significant, particularly when a large number of training examples are used.

5 Final Remarks

A data-parallel connectionist simulator capable of training recurrent time-delay networks on the Connection Machine CM-2 was described. The simulator was benchmarked with a serial implementation, and a time complexity independent of the number of training examples was corroborated.

As research in connectionism advances, and the demand for more powerful training methods grows, particularly in largely experimental and data-intensive areas, the advantage of a parallel approach becomes clear. In a field where emphasis is on parallel and distributed computation, it is natural to utilize parallel and distributed training algorithms to their best advantage.

[2]Classification results are not reported, since focus was on timing performance.

References

[1] R. Fletcher. *Practical Methods of Optimization.* John Wiley and Sons, 1980.

[2] T. Fontaine and L. Shastri. Handprinted digit recognition using spatiotemporal connectionist models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1992.

[3] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 396–404. Morgan Kaufmann, 1990.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representation by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. Cambridge, MA: MIT Press, 1986.

[5] A. Singer. Implementations of artificial neural networks on the Connection Machine. *Parallel Computing*, 14:305–315, 1990.

[6] Thinking Machine Corporation. *C* Programming Guide*, November 1990.

[7] R. Watrous. Learning algorithms for connectionist networks: applied gradient methods for nonlinear optimization. Technical Report MS-CIS-87-51, University of Pennsylvania, June 1987.

[8] R. Watrous. GRADSIM: a connectionist network simulator using gradient optimization techniques. Technical Report MS-CIS-88-16, University of Pennsylvania, March 1988.