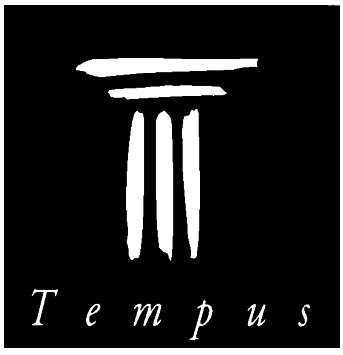
NEURAL NETWORKS IMPLEMENTATION IN PARALLEL DISTRIBUTED PROCESSING SYSTEMS

USER'S MANUAL

João Carlos Negrão Ventura
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
jcv@students.si.fct.unl.pt



Under the supervision of

dr inz. Urszula Markowska-Kaczmar
Wydzial Informatyki i Zarzadzania
Politechnika Wroclawska
Kaczmar@ci-2.ci.pwr.wroc.pl

CONTENTS

1. Introduction....1

2. Compiling the programs....1

3. Description of the programs....2

3.1 Centralized version....2

3.2 Pattern Distribution version....2

3.3 Library Distribution version....3

4. Running the programs....3

4.1 Centralized version....3

4.2 Distributed versions....3

5. Auxiliary files....3

5.1 Neural Network file....3

5.2 Configuration file....5

5.2.1 Method....*5*

5.2.2 Seed....*6*

5.2.3 Stopping Conditions....*6*

5.2.4 Reporting Options....*7*

1. INTRODUCTION

MyDMBP is a set of programs that perform the back-propagation algorithm in fully-connected feed-forward neural networks on distributed systems. It is based on the Matrix Back-Propagation algorithm developed by Dr. Davide Anguita of the University of Genova. Three different versions of the program are covered in this document, a centralized (i.e. non-distributed) version and two different distributed methods. The functionality of all the different versions is exactly the same, only differing in the run time necessary.

A previous knowledge of the theory and practice of neural networks is necessary to understand how to run these programs. Knowledge of the PVM software package is also recommended.

This document starts by explaining how to compile the programs, followed by a brief explanation of the different methods used to distribute the program. Finally, all the details for running the programs are presented. The different command lines necessary and the format and meaning of the configuration files are explained.

These programs were developed by João Ventura under the supervision of dr inz. Urszula Markowska-Kaczmar of the Wroclaw Technical University, in the scope of a TEMPUS sponsored diploma work from March to July 1997.

2. COMPILING THE PROGRAMS

To compile the programs, **bison** and **flex** must be installed on the system. For the distributed versions, **PVM** must also be installed. It is also recommended to use the GNU C compiler (**gcc**) and the GNU **make** utility.

In order to ease the process of compiling the programs, a Makefile was created for each version of the program. The PVM package has its own Makefile type, created to ease the process of compiling in different machines, which uses a file called Makefile.aimk.

In case **gcc**, **flex** or **bison** are not installed on the system, it may be possible to use a replacement program, try to replace (in the Makefile):

`CC=gcc`	for	`CC=cc`	to use the standard system C compiler.
`YACC=bison`	for	`YACC=yacc`	to use the Berkeley yacc parser generator.
`LEX=flex`	for	`LEX=lex`	to use the standard lexical analyzer.

Note however, that no guarantees are provided as to whether the programs will compile with the alternate programs. In the case of the distributed versions, it is also possible to edit the Makefile.aimk to perform these changes.

To compile the centralized version of the program, change to the centralized sub-directory:

```
cd mydmbp/centralized
```

and compile the programs using the make utility:

```
make
```

In order to compile the distributed versions, make sure that you have the following definitions in your environment variables:

```
PVM_ROOT=/usr/local/pvm3          or something similar
PVM_DPATH=$PVM_ROOT/lib/pvmd
```

If none of this definitions is part of your environment, please refer to the PVM documentation in order to correctly configure PVM.

Once you have PVM correctly configured, change to the appropriate directory:

```
cd mydmbp/xxxdist
```

and compile the programs using the PVM make utility:

```
aimk
```

the programs will then be installed on the PVM_ROOT/ARCH/ directory.

3. DESCRIPTION OF THE PROGRAMS

This is a very succinct description of the programs. For a more detailed description, refer to this project's report, where the inner workings of the algorithm, the programs and the run-times obtained are explained.

3.1 CENTRALIZED VERSION

The centralized version is based on the MBP - Matrix Back-Propagation algorithm by Dr. Davide Anguita. This algorithm takes advantage of the low-level properties of modern computers in order to speedup the traditional Back-Propagation algorithm. Transforming all the computations into matrix operations has the benefit that instead of several cyclical operations there is only one operation. This, coupled with a set of very efficient matrix routines, results in a very fast version of BP. The set of routines used take into consideration notions of modern processor architectures such as memory cache and instruction pipelining.

3.2 PATTERN DISTRIBUTION VERSION

This version of the program works by dividing the input patterns into several parts and performing in parallel the forward and backward phases of BP. At the end of each step, the slave tasks (also known as workers) report their results to the master task which then accumulates all the partial results and resends it to the slaves.

3.3 LIBRARY DISTRIBUTION VERSION

This version works by replacing all the calls of the more computationally intensive routines, the matrix multiplications, with a set of distributed routines that perform the multiplication in parallel. The set of routines used is based on the SUMMA algorithm, that stands for Scalable Universal Matrix Multiplication Algorithm.

4. RUNNING THE PROGRAMS

4.1 CENTRALIZED VERSION

The syntax of the command line for this version is:

```
mydmbp <net file> <config file>
```

Both parameters need to be supplied. The first file details the neural network architecture and the set of input and output patterns. The second file deals with other aspects of the program, such as learning methods, report options, stopping criteria, etc. The syntax of these two files will be explained in the next section.

4.2 DISTRIBUTED VERSIONS

In order to run the distributed versions, you have to run the PVM daemon as well. To do this, type `pvm` in the command prompt. After, add the desired number of machines in which you want to be able to run worker tasks by using the `add <hostname>` command of the PVM command interface. Quit leaving the daemon running. For further information, refer to the PVM documentation.

All the distributed versions share the same command line format, which is:

```
mydmbp <net file> <config file> <number of slaves>
```

The syntax is almost the same as the centralized version, except for the last parameter that specifies the number of slave (worker) tasks. This value is usually smaller or equal to the number of machines available in the PVM processor pool. The binaries for these versions are usually found in the PVM_ROOT/ARCH/ directory.

5. AUXILIARY FILES

As was mentioned previously, the program uses two different types of auxiliary files. The first type contains all the information of the problem being analyzed, while the second contains the parameters that specify and alter the learning algorithm.

5.1 NEURAL NETWORK FILE

The neural network description file details the layout of the network (i.e. how many input neurons, hidden layers and numbers of neurons in each and the number of output

neurons). It also contains the mapping of input to output neurons which the network should perform. The syntax is then:

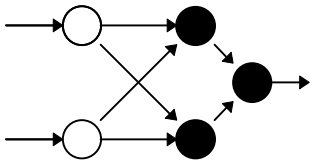
```
<nº of input neurons> <nº of hidden layer 1 neurons> … [nº of hidden
layer n neurons] <nº of output neurons>
<input pattern 1> <output pattern 1>
<input pattern 2> <output pattern 2>
…
<input pattern N> <output pattern N>
```

As can be seen, the syntax is extremely simple, allowing easy conversion of raw data into the format required. The first line contains the number of neurons in each layer of the network. It starts with the input layer, followed by an arbitrary number of hidden layers and finishes with the output layer. The following lines detail each input pattern and the output pattern into which it should be mapped.

As an example, this is the file for learning the classical XOR problem:

```
2 2 1
0 0 0
0 1 1
1 0 1
1 1 0
```

This specifies the following neural network,



with two units in the input layer, two units in the only hidden layer and 1 unit in the output layer. This can be seen by the first line containing the sequence 2 2 1. The remaining 4 lines contain the XOR function.

A more elaborate problem, the 4-bit parity problem, which is an extension of the XOR problem and a possible network for learning it is as follows:

```
4 4 1
0 0 0 0 0
0 0 0 1 1
0 0 1 0 1
0 0 1 1 0
0 1 0 0 1
0 1 0 1 0
0 1 1 0 0
0 1 1 1 1
1 0 0 0 1
1 0 0 1 0
1 0 1 0 0
1 0 1 1 1
1 1 0 0 0
1 1 0 1 1
1 1 1 0 1
1 1 1 1 0
```

Note that the number of hidden layers and the number of neurons in each is completely arbitrary. Usually, it only influences the result in terms of learning time. Probably a 4x3x1 neural network would be perfectly able to learn the 4-bit parity problem, in slightly more time than the 4x4x1 network. On the other hand, a 4x1x1 network would probably be unable to learn it in a reasonable time.

Another different type of problem family is the encoder. This type of problem maps the input patterns into itself, passing through a very narrow hidden layer. An 10-2-10 encoder, which maps a 10-bit pattern into itself, passing through 2 neurons in the hidden layer is detailed in the following file:

```
10 2 10
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
```

5.2 CONFIGURATION FILE

The configuration file specifies how the learning will proceed, namely the learning method used, the seed for the random number generator that will be used to initialize the weights, and finally the stopping conditions. All the keywords are optional, except for the method which must be specified. The parser is very flexible, so there are no rules of placement of the keyords.

5.2.1 METHOD

The method must be specified, by including a line with the following syntax:

```
method <gd | vogl | yprop | rprop | qprop | scg> <par1> <par2>
```

The last two parameters are method dependent and are given as real numbers. No explanation of the algorithms is given here, since some theoretical explanation would have to be made in order to introduce the notation and even then, some algorithms would take a large explanation. Refer to this project's report where a detailed explanation of the algorithm is given.

5.2.1.1 Gradient-descent method

For the basic back-propagation algorithm also known as gradient descent, the syntax is:

```
method gd <η> <α>
```

5.2.1.2 Vogl's acceleration

```
method vogl <φ> <β>
```

5.2.1.3 YPROP

```
method yprop <K_a> <K_d>
```

5.2.1.4 RPROP

```
method rprop <Δ_0> <Δ_max>
```

5.2.1.5 QuickProp

```
method qprop <ε> <μ>
```

5.2.1.6 Scaled Conjugate Gradient

```
method scg <σ> <λ_1>
```

5.2.2 SEED

At the beginning of the learning algorithm, the network must be initialized with random weights. In order to recreate a run, it is useful to be able to use the same random numbers again. This is done by using a user supplied seed for the random number generator, by providing a line with the following syntax:

```
seed <number>
```

The number must be an integer. If this parameter is not supplied, the current time obtained using the time(2) system call is used (i.e. the number of seconds since 00:00:00 GMT, January 1, 1970). Since the random number generator used is supplied by the standard C library, and the implementation of the latter is vendor-dependent, the behavior will only remain the same when using the same value for the seed in computers with the same operating system.

5.2.3 STOPPING CONDITIONS

Except for Scaled Conjugate Gradient, no algorithm has any stopping criteria embedded, making it necessary for the user to provide one or more conditions that when achieved make the learning stop. If no stopping conditions are supplied, the algorithm will continue indefinitely (this is true also for SCG, since it's stopping condition is nearly impossible to achieve).

5.2.3.1 Stop

By including a line with the following syntax:

```
stop <number>
```

the algorithm stops after doing the specified number of epochs. That is, after presenting all the input patterns several times. This option should always be included, since it is the only condition that will always be fulfilled.

5.2.3.2 MaxErr

The most usual stopping condition used in neural network research is the sum of the squares of the errors (difference between the desired and the obtained output). To use this condition, supply a line with the syntax:

```
naxerr <number>
```

The value of this parameter should not be too high, which would lead to great errors in the output and, nor too low, which may be nearly impossible to achieve. A good value is usually between 0.2 and 0.5.

5.2.3.3 MaxWrong

The MaxWrong condition is defined as the maximum percentage of wrong inputs allowed. An input is considered right when the obtained output is within the 40% range nearest to the desired output. This criteria should only be used with binary outputs. The syntax to this condition is:

```
maxwrong <number>
```

5.2.4 REPORTING OPTIONS

After learning, it is useful to obtain some output from the program. In order to obtain some type of information, supply a line containing the type of reports desired:

```
report [count] [time] [stats] [mips] [weights] [output]
```

All, some or none of the above options can be supplied. The order in which it is supplied is not important, and it does not affect the order in which the report is emitted, which always comes in the count, time, mips, stats, weights and output order.

5.2.4.1 Count

To obtain a count of the number of runs performed until stopping, include the `count` option. The report emitted will be similar to:

```
In 8309 iterations
```

5.2.4.2 Time

To obtain the time elapsed between the beginning of the run and the end, include the `time` option. The time elapsed is the real time elapsed, and does not take into account

any time lost because of allocation of the processor to another process. The report obtained by including this option will be similar to:

```
In 42.5 seconds
```

5.2.4.3 Statistics

By including the `stats` option, a report containing the error (sum of the squares of the errors) and the percentage of wrong outputs will be printed. An example is:

```
Error 0.2467 Wrong 12.5
```

5.2.4.4 Millions of connection updates per second

To obtain a report about the number of MCUPS (millions of connection updates per second) obtained, include the `mips` option in the `report` line. Don't ask why it isn't mcups. The line obtained will be something like:

```
With 1.70563 MIPS
```

5.2.4.5 Weights

It may be useful to know the value of the weights obtained at the end of the learning algorithm. In order to obtain a dump of this information, include the `weights` option in the `report` line.

5.2.4.6 Output

By including the `output` option, a dump of the output at the output layer is obtained. This information may be useful for the user to evaluate the quality of the learning performed.