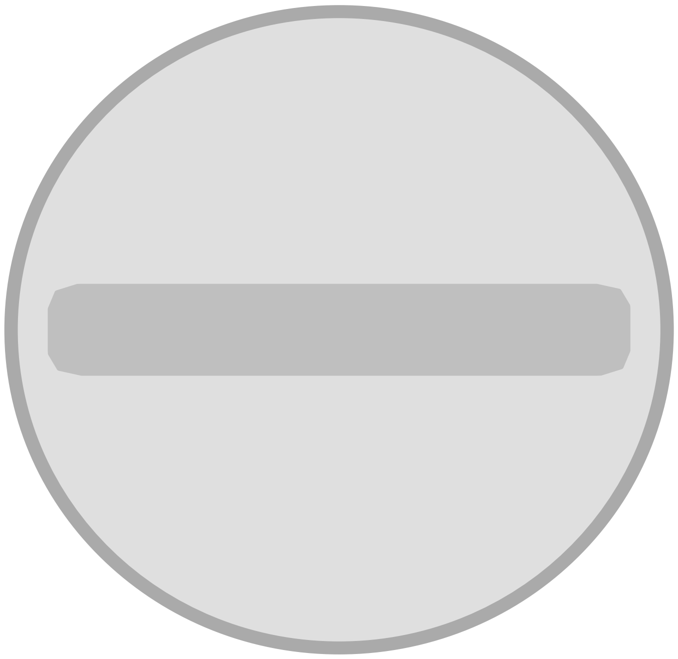


Aspirin/MIGRAINES

Release V7.0

By

Russell R. Leighton

This book was typeset using LaTeX.

If this doesn't work...try Prozac.

The MITRE Corportion has funded the initial development of A/M. Since leaving MITRE, I continue to support A/M with the help of others that value A/M as an important tool.

Many people have contributed to the robustness of the A/M software. It would be difficult to give the appropriate credit all of the people who have made contributions. Some have given suggestions, many have tested the software on different platforms, tracked and fixed bugs and offered me access to their computers. Primarily it has been the work of the INTERNET neural network research community. In particular, the following names come to mind as people that have made significant contributions to A/M: Dave Ault, Chedley Aouriri, Michael Bobak, Val Bykovsky, Bart Conrath, Dan Dickey, Nga Do, Charles Fletcher, David Goblirsch, Mathew Grzech, Duane Gustavus, Glenn Himes, Garry Jacyna, Henrik Klagges, Manette Lazear, Sam Lin, John Michopoulos, William Morgart, Thu Nguyen, Steve Rehfuss, Dave Subar, Alexis Wieland, Michael Yearworth. I look forward to continuing working with these and more interested people to develop A/M to its fullest potential.

Contents

Chapter 1

SUMMARY

A suite of software tools for developing neural network simulations, called Aspirin/MIGRAINES (A/M) has been developed by an internally funded neural network research effort at the MITRE Corporation. These tools include a declarative language for describing neural networks, code generators to take the declarative description and generate optimized simulations, and a user interface for analysis. The A/M V7.0 software is available free from a number of INTERNET ftp sites (see section 2.6).

The software tools are made up of two major components: a neural network description language called "Aspirin" and a user interface called "MIGRAINES". An Aspirin file is created that describes the network architecture. This file is then "compiled" to create a C program for simulating that network. This simulation is then compiled with an ANSI C compiler and linked with application code and the MIGRAINES interface routines.

The software is for creating, and evaluating, feed-forward networks such as those used with the backpropagation [Rumelhart, 1986] learning algorithm. It is aimed both at the expert programmer/neural network researcher who may wish to tailor significant portions of the system to his/her precise needs, as well as at casual users who will wish to use the system with an absolute minimum of effort. The software is written for a C/UNIX environment. A great deal of effort has gone into creating a portable software distribution. As of this writing, the following UNIX systems are supported:

1. Convex
2. Cray
3. DecStation
4. IBM RS/6000
5. Intel 486/386 (Unix System V, Linux)

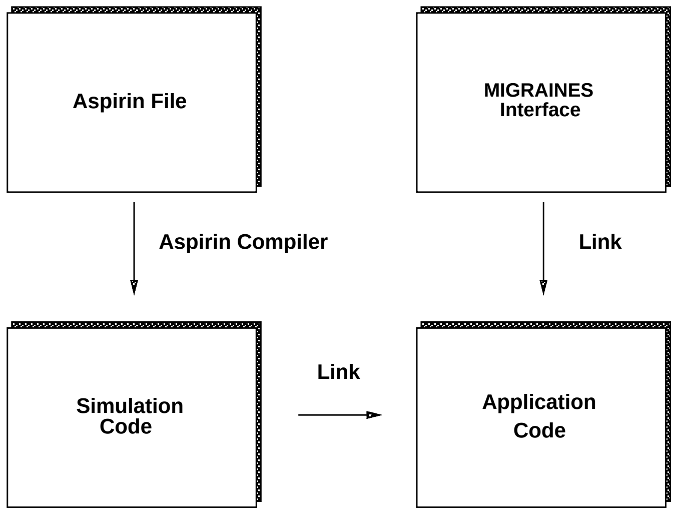


Figure 1.1: Interaction of Aspirin and MIGRAINES

6. HP 9000
7. NeXT
8. News
9. Silicon Graphics
10. Sun

In addition, the software can be run on coprocessor boards added to a UNIX host. As of this writing, the following coprocessors are supported:

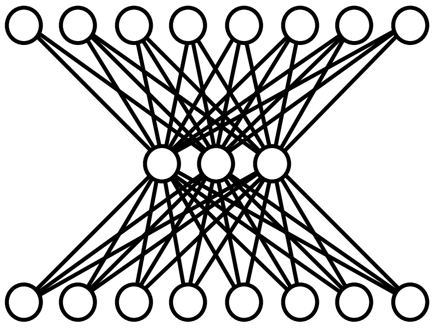
1. Mercury i860 (40MHz)
2. Meiko Computing Surface w/i860 (40MHz) Nodes

1.1 ASPIRIN

Aspirin was originally conceived as a way of dealing with the "MIGRAINES" of coding neural network simulations. Our goal was to create an underlying system that would exist behind the interface and provide the network modeling facilities. The system had to be flexible enough to allow research, that is, make it easy for a user to make frequent, possibly substantial, changes to network

```
DefineBlackBox Encoder
{
        InputSize-> [8 x 1]
        OutputLayer-> Out
        Components->
         {
              PdpNode Out [8 x 1]
              {
                 InputsFrom-> Hidden
              }
              PdpNode Hidden [3 x 1]
              {
                 InputsFrom-> $INPUTS
              }
         }
}
```

Aspirin Neural Network Description



Network Architecture

Figure 1.2: Aspirin Description and Network Architecture

designs and learning algorithms. At the same time it had to be efficient enough to allow large "real-world" neural network systems to be developed.

Aspirin uses a front-end parser and code generators to realize this goal. A high level declarative language has been developed to describe a network. This language was designed to make commonly used network constructs simple to describe, and, potentially, to allow any network to be described. The Aspirin file defines the type of network, the size and topology of the network, and descriptions of the network's input and output. This file may also include information such as initial values of weights, names of user defined functions, and hints for the MIGRAINES interface system.

The Aspirin language is based around the concept of a *black box*. A black box is a module that (optionally) receives input and (necessarily) produces output. Black boxes are autonomous units used to construct neural network systems. Black boxes may be connected arbitrarily to create large, possibly heterogeneous network systems. As a simple example, pre- or post-processing stages of a neural network can be considered black boxes that do not learn.

The output of the Aspirin parser is sent to the appropriate code generator that implements the desired neural network paradigm. The goal of Aspirin is to provide a common, extendible front-end language and parser for different network paradigms. The publicly available software includes a backpropagation code generator that supports several variations of the backpropagation learning algorithm with wide variety of capabilities.

A file describing a network using the Aspirin language is processed by the Aspirin parser. Files containing C functions to implement that network are generated. This code can then be linked with an application using these routines to control the network. Optionally, a complete simulation can be automatically generated that is integrated with an interactive interface and can read data in a variety of file formats. If a different neural network paradigm is desired a new code generator is used. Our hope is that after the software is released to the public other researchers, using this code as an example, will add more code generators for other paradigms, machines, and computer languages. For example, if a researcher wished to modify the backpropagation learning rule, he/she could slightly change the existing code generators to produce a new code generator integrated with the other Aspirin/MIGRAINES tools.

1.2 MIGRAINES

MIGRAINES is an interactive interface that allows users to export data from the neural network simulation to analysis and graphics packages via Unix pipes. The MIGRAINES interface has been designed to be flexible and portable. Since this interface uses a very simple data format, the data can be easily used by many different plotting packages. (see chapter 8). MIGRAINES is not necessary for the execution of the Aspirin system. Networks may be designed, executed, tested, and saved entirely apart from any interface.

Chapter 2

INTRODUCTION

Aspirin is a declarative language used to describe complex neural networks. Our goal in designing Aspirin was to make, not only, commonly used network constructs simple to describe, but also to allow almost any network to be specified. An Aspirin neural network description is parsed and a number of simulation routines are generated in a high level language (currently in ANSI C [Kernighan, 1988]). These computer routines are then compiled using standard compilers and linked either to the MIGRAINES interface (a process that is usually done automatically) or used with other more application-specific systems. Aspirin currently supports backpropagation learning techniques with a number of algorithmic and topological variations. These include skip-level and tesselated connections (i.e., limited receptive fields), several node transfer functions, delays, "static" feedback, and a number of learning heuristics.

MIGRAINES is an interactive interface that allows users to export data from the neural network simulation to analysis and graphics packages via Unix pipes. The MIGRAINES interface has been designed to be flexible and portable. Since this interface uses a very simple data format, the data can be easily used by many different plotting packages.

The Aspirin/MIGRAINES system is being written for a UNIX environment with current development being carried out on a Silicon Graphics Indigo. The Aspirin compiler makes use of the UNIX utility `yacc`. The Aspirin/MIGRAINES utility `bpmake` is built on the UNIX utility `make`.

2.1 CONTENTS

This book describes the command syntax and options for creating, compiling, and running Aspirin simulations. Most of the compilation commands are executed automatically for the user by the `bpmake` utility. One of the strengths of the Aspirin/MIGRAINES system is its ability to *automatically* generate optimized simulation code. A number of tutorial examples and code fragments are provided to help explain the material and how these capabilities can be used to best advantage. It is recommended that the user read these sections before large neural network system simulations are developed.

2.1.1 Notation and Conventions

Throughout this manual, text that is to be viewed as input to or output from the computer is printed in a typewriter-like font. For example, the file name `network.aspirin` and the command:

```
aspirin network.aspirin network -c backprop
```

are printed in this font to denote their special role.

Because the Aspirin/MIGRAINES system is case sensitive, the strings `Network`, `network`, `NETWORK`, and `NeTwOrK` are treated as distinct by the system.

Within many of the code samples there will be text within angle brackets (e.g. `<text>`). These are descriptive names for variables that are to be replaced in user code by actual variable names. As an example, the descriptive name `<aspirin-file>` could be replaced by the file name `network.aspirin`.

This report is written assuming that the Aspirin/MIGRAINES software is running in the C shell environment on a Unix workstation. See Appendix A for information regarding particular UNIX systems.

2.2 Prerequisites

Before Aspirin and MIGRAINES can be used the proper UNIX environment variables must be set (Appendix B on page 94).

In the remainder of this report it is assumed that the environment variable `NNTOOLS` has been set to the Aspirin/MIGRAINES "home" directory. If, for example, Aspirin/MIGRAINES has been installed in the directory `/usr/nntools`, this can be accomplished by including the statement:

```
setenv  NNTOOLS  /usr/nntools
```

in the user's `.cshrc` file. The environment variable `NNTOOLS` is referenced in a command by adding the prefix "$" to `NNTOOLS`, e.g., `$NNTOOLS`. For example, to move into the directory defined by `NNTOOLS` you would type

```
cd $NNTOOLS
```

References will be made throughout this manual to Aspirin/MIGRAINES files and directories using `$NNTOOLS`, such as referring to the Aspirin directory `$NNTOOLS/src/aspirin`.

In addition, the user's search path must include the appropriate Aspirin/MIGRAINES `bin` directory. For a Sun3 this directory is `$NNTOOLS/bin/sun_68k` and for a

Sun4 this directory is `$NNTOOLS/bin/sun_sparc`. Generally this is done by including a statement of the form

```
            set path = ($path $NNTOOLS/bin/$MACHTYPE)
```

in your `.cshrc` file. The `$MACHTYPE` variable should be set to the kind of machine you are working on.

The following commands enable you to check that the appropriate environment variables have been set. First check to see that the environment variable `NNTOOLS` is defined by typing:

```
printenv NNTOOLS
```

The computer should respond with `/usr/nntools`, or whatever the correct directory path is on your system.

If that works correctly, check to see if you can find the executable utilities (that is, if your path has been set correctly) by typing:

```
aspirin
```

Since you did not give this program the proper arguments it should reply with a message telling you what arguments to use, for example:

```
Usage:  aspirin <aspirin file> <C file> -c <network compiler> [<compiler flags>]
```

If this message does not appear, see Appendix A(page 92) and Appendix B(page 94).

2.3 THE HISTORY OF ASPIRIN/MIGRAINES

The Aspirin/MIGRAINES system has existed in some form since 1986 and has principally been the work of Russell Leighton and Alexis Wieland. It was preceded and motivated by a series of hand-coded systems for implementing and testing various neural network paradigms under a variety of machines and graphic environments. The name MIGRAINES was conceived by David Subar[1]. The declarative language Aspirin, which actually predates the name MIGRAINES, was envisioned as the behind-the-scenes work horse which would tie the system together. Its use in this role of dealing with the tedium of neural network simulation("MIGRAINES") that inspired its name.

2.4 Previous Versions

During its life, Aspirin/MIGRAINES has existed in a variety of forms on a number of machines and has been developed and used by a number of people.

[1]The actual meaning of the acronym MIGRAINES has changed repeatedly since its introduction.

The original system was designed for the Symbolics and was written in a mixture of Zeta Lisp and Common Lisp.

The UNIX-based versions were first written and used as MITRE began its neural network research under a MITRE Sponsored Research (MSR) program in October of 1987. The people originally involved with writing and using this software made the initial assumption that any fixed neural network simulation system would not be computationally powerful enough for many applications. There was also the assumption made that, since people would want to write their own applications, Aspirin/MIGRAINES would only be a tool to remove the hard or tedious parts of coding networks. However, since 1988, people outside of MITRE's neural network research team have begun to use this software. As a result, the team members have realized that most people do not want to write *any* software. Enhancements made in response to this conclusion have made Aspirin/MIGRAINES a much more powerful system. The basic system architecture has been described in [Wieland, 1988].

Version 4.0 was released on the InterNet in March of 1991. It was very much like V5.0 but used a window system called NeWS1.1 for a graphical interface.

Version 5.0 introduced:

- Support for autoregressive nodes [Leighton, 1991].
- Line search and conjugate gradient optimization.
- Interface allowing the user to open Unix pipes from neural network data structures.

Version 6.0 introduced:

- ANSI C implementation and code generation
- The `analyze` utility

2.5 Current Version - V7.0

This release introduces:

- Tk/Tcl graphical interface to MIGRAINES called **NNinspect**.
- Bugs fixes for problems with AM6.0.

2.6 ACQUIRING ASPIRIN/MIGRAINES

The original am6 software is available from two FTP sites, CMU's simulator collection. The updated am7 software is available at http://www.elegant-software.com. The compressed tar file is a little less than 2 megabytes. Most of this space is taken up by the documentation and examples.

Computer mail concerning Aspirin/MIGRAINES should be sent to:

Russell Leighton russ@elegant-software.com

Chapter 3

A TUTORIAL INTRODUCTION

Since most of the Aspirin/MIGRAINES neural network tools are straightforward to use, this manual begins with a short tutorial example. This example demonstrates the major features of both Aspirin/MIGRAINES, and the use of these tools to carry out neural network research and development.

The danger in this approach is that in the attempt to elucidate the simple features of the Aspirin/MIGRAINES system, its true power and utility may be obscured. The Aspirin/MIGRAINES package was explicitly developed to facilitate neural network research. The bulk of the user's manual describes the full power of the system and gives hints on how to extend it.

3.1 Before Starting

This section assumes that Aspirin/MIGRAINES has been installed correctly on your computer and that the appropriate environment variables have been set (see page 9).

3.2 Encoder/Decoder Example

Throughout this report we use the encoder/decoder problem presented in [Rumelhart, 1986] as a simple neural network example. A complete example of an encoder/decoder system can be found in `$NNTOOLS/examples/encode`.

The encoder/decoder problem is to take each of eight input vectors, force it through a bottleneck of three computing nodes, and then reproduce the same input vector as the output. This problem requires that the network find some way of compressing, or encoding, the inputs via the small number of hidden nodes to create the desired output.

3.2.1 Creating An Executable Program

You are advised to create a new directory for each new network description and its associated files. In this case, we are going to use the encoder network description and the data files that come with the Aspirin/MIGRAINES distribution. To create a directory and copy these files into it, type the following commands:

1. `mkdir encode`[1]
2. `cd encode`[2]
3. `cp $NNTOOLS/examples/encode/* .`[3]

You have now created a new directory and copied the following files into it: `encode.aspirin` which is an Aspirin language description of the neural network, a data format file named `encode.df`, and an ASCII data file named `encode.data`. The purpose of each of these files will be explained later in this example.

The next step is to generate an executable program from the Aspirin language file. This is done by typing the single-word command:

```
bpmake
```

A dozen or so lines of text will appear on the screen as the program `aspirin` is called to translate the `encode.aspirin` file into an `encode.c` file. This new C file is then compiled by the C compiler into a machine executable program named `encode`. The name of the executable program is the same as the name of the Aspirin file minus the ".aspirin" suffix.

The file `encode.c`, created by `bpmake`, is a C program for implementing the network-specific portion of the encoder task. Interested readers can browse this code. (The network-independent portion of the code is in the file `$NNTOOLS/migraines/bp/Backprop.c`. It includes all of the hooks for running the simulator, both with and without MIGRAINES) You have now completed the process of creating an optimized simulation.

3.2.2 Running `encode`

This example is called `encode` because the neural network will learn to encode a set of eight patterns with a hidden layer of three nodes . The network architecture has eight inputs connected to all three hidden nodes. The nodes in

[1]Creates a new subdirectory, named "`encode`" in the current directory.
[2]Move into the newly-created directory.
[3]Copies all of the files for the encoder into this new directory.

the hidden layer are fully connected to an output layer of eight nodes. To see this, type: `more encode.aspirin` [4]. The aspirin file describes the network to be simulated and the `bpmake` program creates the executable simulation.

The simulation created by `bpmake` can read a number of data file formats (see page 75). One of these formats is an ASCII format. The data for this example is located in a file called `encode.data`. Type: `more encode.data` to see the eight patterns that the neural network will learn to encode. The executable simulation, called `encode`, can take a large number of optional arguments. Type: `encode -help`. Only a few of these options will be discussed here.

One of the options is `-d <datafile>` . The `<datafile>` is *not* the file `encode.data`. The input of data into simulations generated by `bpmake` is indirectly referenced through a file called a data format file (or .df file). This file tells the simulation which files to load and specifies the format of the data in the respective files. Type: `more encode.df`. This .df file contains one `ReadFile` command that tells the simulation to load the data in `encode.data` and that this file has data of type ASCII. There are also some optional keywords that are included within the `ReadFile` command.

Another option to the `encode` simulation is `-l`, which stands for learn. Type: `encode -l` . You should see the program exit with a message like: "No generators defined!" [5] This means that the program has no data to use for learning. Type: `encode -l -d encode.df`. You should see something like:

```
(silicon.14) <- encode -l -d encode.df


Backpropagation Learning

Black Boxes:

encode (Saved at Iteration 0)

Loading Data Files...Done.

Learning Rate: 0.200000
Inertia: 0.950000
Your learning rate is a bit high.
If this does not converge try lowering it.

Learning...
Testing every 5000 iterations.
Saving every 5000 iterations.
```

[4]The `more` program will display a page at a time of the Aspirin file, hit the space bar to see more of the file.

[5]A generator is a function that supplies data to the simulation (see page 82).

```
Dumping Network at 5000.
Success!! Dumping Network.Finished.


Total Iterations 5000

Elapsed compute time: 2.440000 seconds
Elapsed system time: 0.170000 seconds
(silicon.15) <-
```

The simulation has successfully learned to do the encoding and has saved the state of the network parameters (weights, etc.) to a file called `Network.Finished`.

The simulation will periodically save its state to a file called `Network.save`. The period is controlled by the `-s` option. Type: `encode -l -d encode.df -s 1000`. You should see it dump the network every 1000 iterations. The default is to save the state every 5000 iterations.

The simulation periodically pauses and checks to see if the network has converged. How often it checks is controlled by the `-t` option. The `-t` option has three arguments. The first is the number iterations between tests, where an iteration is one forward and one backward pass. If this is set too low, the simulation will spend the most of the time testing, not learning. If it is set very high, you will have to wait longer before your simulation will stop. The second argument is the number of iterations the simulation must pass the convergence test before being allowed to stop. Usually this number equals the number of patterns in the data file. However, if noise is being added dynamically to the input patterns, you want this number much higher than simply the number of patterns. The third argument is a real number denoting the error bound. If the convergence test is passed for all of these iterations then the network is said to have converged. The convergence test uses the maximum absolute difference as the error calculation. If the absolute value of the difference between each output node and its associated target during all of the specified iterations is less than some number (the error bound) then the network is said to have converged. Type: `encode -l -t 10 8 0.2 -d encode.df`. This tells the simulation to learn and pause every ten iterations to check for convergence. The network will be said to have converged if, for eight successive iterations, the maximum absolute difference between *all* output node values and their associated target values is less than 0.2. You should notice that this finished executing with fewer iterations than the previous run.

The `-alpha` option allows you to override the learning rate parameter. Type: `encode -l -d encode.df -t 10 8 0.2 -alpha 0.35`. This should converge even faster than last time. Try this with many different learning rates. You will notice that convergence time does not vary linearly with the learning rate parameter.

Sometimes you might want to see the actual output values. The `-p` option used with the `-f` option will print the values of the outputs and the associated targets. The `-f` option takes one argument which is the number of times to propagate forward. Type: `encode -d encode.df -p -f 8 Network.Finished`.

This rest of this section illustrates using the MIGRAINES interface with the Aspirin generated simulation. Before continuing you should bring up a window system that will allow you to use **gnuplot** (see appendix G).

3.2.3 Analyzing `encode` with MIGRAINES and gnuplot

You can run the executable program `encode` with the MIGRAINES interface by typing the following command to the UNIX shell:

```
encode -d encode.df
```

You should see the following printed out:

```
Aspirin/MIGRAINES 6.0

Backpropagation Learning

Black Boxes:

encoder (Saved at Backprop Iteration 0)

Welcome to the MIGRAINES user interface.
Copyright (C) 1988-1992 The MITRE Corporation.
Type ? for help.
Type quit to end this session.

MIGRAINES%
```

This is a prompt for the MIGRAINES interface. This interface allows you to navigate through the neural network structures declared in the .aspirin file and connect Unix pipes to extract weight and node data. Each black box, layer and connection matrix are *contexts* that you `push` into and `pop` out of. Within a context you can apply commands consistent with that context. For example, if you are in a layer context, then you can open a Unix pipe to get the data in the nodes in that layer.

In this tutorial we will explore a few of the commands and then run a demonstration file that will extract weights and node values from the neural network. These will then be plotted by **gnuplot**[6]. At the prompt:

```
MIGRAINES%
```

[6]If you do not have **gnuplot** this will not work. However, the files produced by this demo are ASCII and could be plotted by other plotting packages.

type `?`. You will see two groupings of text. The first is entitled `Global Commands`. These are commands that can be executed from any context that allow you to navigate through the neural network structures as well as export data to Unix pipes. The second grouping of text is entitled `SubContexts`. This text lists the available contexts.

Since you have already taught the network to perform the encoding problem, there should be a file in the current directory called `Network.Finished` that contains the weights of the converged network. Type `load Network.Finished` to the MIGRAINES interface. This will result in the converged weights to be loaded into the network.

Type `push TestingContext`. This will put you into the testing context so that you can evaluate how well the network has converged. Type `?`. Notice there are now commands available that are local to this context. Type `Info`. You should see:

```
    Bound: 0.100000 Iterations: 100
```

This means that the testing criteria are a maximum absolute error of 0.1 and 100 iterations. Since the encoding problem only has 8 patterns, the number of iterations needs only to be 8. Type `SetIterations 8`. Type `Info`. Type `SetBound 0.5` then type `StartTesting`. The simulator will evaluate each pattern and stop when either a pattern exceeds the error bound or the number of testing iterations is reached. To evaluate the network performance, set the error bound to be large, test to see which patterns fail and then repeat the process making the error bound gradually smaller. For example:

```
TestingContext% StartTesting

        Testing 8 patterns...
        Success, 8 patterns passed with 0.500000 error bound!
TestingContext% SetBound 0.4

TestingContext% StartTesting

        Testing 8 patterns...
        Success, 8 patterns passed with 0.400000 error bound!
TestingContext% SetBound 0.3

TestingContext% StartTesting

        Testing 8 patterns...
        Success, 8 patterns passed with 0.300000 error bound!

TestingContext% SetBound 0.2
```

```
TestingContext% StartTesting

        Testing 8 patterns...
        Success, 8 patterns passed with 0.200000 error bound!
TestingContext%

TestingContext% SetBound 0.1

TestingContext% StartTesting

        Testing 8 patterns...
        Pattern 0 (3 in encode.data) exceeded a 0.100000 error bound.
TestingContext%
```

Type pop. Now you have moved out of the `TestingContext` back to the root context. Type `push encoder`. For information on the `encoder` black box type `Info`. You should see:

```
MIGRAINES% push encoder

encoder% Info

        Black Box: encoder
                Dynamic (can learn)
                Efferent (has target outputs)
                Input Size: 8 x 1
                2 Layers
```

Type ? to see the subcontexts:

```
encoder% ?

Global commands:

pbinary (Output pipe data in binary)
pascii (Output pipe data in ascii)
pnoheader (Do not use header on pipe data)
pheader (Use header on pipe data)
pinfo (Info on pipes)
pclose <pipe name> (close a Unix pipe)
popenWeights <x node index> <y node index> <pipe name> <commands> (open a Unix ...)
popenInputs <pipe name> <commands> (open a Unix pipe to accept input vector)
popenTargets <pipe name> <commands> (open a Unix pipe to accept output target vector)
popenBiases <pipe name> <commands> (open a Unix pipe to accept node bias vector)
popenNodes <pipe name> <commands> (open a Unix pipe to accept node value vector)
cycle (<iterations> present inputs/targets to network)
load (<filename> load neural network weight file)
```

```
push (<context> down context)
pop (up context)
poproot (go to root context)
source  (<filename> evaluates commands in file)
!  (<string> executes system call)
echo  (<string> prints string)
quit (quit MIGRAINES)
? (this message)

Current context commands:

Info

Subcontexts:

Targets
$INPUTS
encoder:Output_Layer
encoder:Hidden_Layer
```

Each subcontext is a component of the black box. The `Targets` and `$INPUTS` subcontexts allow you to open Unix pipes to the target and input data, respectively. The other subcontexts represent the layers in the neural network. From within these subcontexts you can open Unix pipes to the node values and the bias weights. Type `push encoder:Output_Layer`. Type `Info` for information about the layer:

```
encoder% push encoder:Output_Layer

encoder:Output_Layer% Info

        Layer: encoder:Output_Layer
                Size: 8 x 1
```

Type `?` to see the subcontexts:

```
encoder:Output_Layer% ?

Global commands:

pbinary (Output pipe data in binary)
pascii (Output pipe data in ascii)
pnoheader (Do not use header on pipe data)
pheader (Use header on pipe data)
pinfo (Info on pipes)
pclose <pipe name> (close a Unix pipe)
popenWeights <x node index> <y node index> <pipe name> <commands> (open a Unix ...)
popenInputs <pipe name> <commands> (open a Unix pipe to accept input vector)
```

```
popenTargets <pipe name> <commands> (open a Unix pipe to accept output target vector)
popenBiases <pipe name> <commands> (open a Unix pipe to accept node bias vector)
popenNodes <pipe name> <commands> (open a Unix pipe to accept node value vector)
cycle (<iterations> present inputs/targets to network)
load (<filename> load neural network weight file)
push (<context> down context)
pop (up context)
poproot (go to root context)
source  (<filename> evaluates commands in file)
!  (<string> executes system call)
echo  (<string> prints string)
quit (quit MIGRAINES)
? (this message)

Current context commands:

Info

Subcontexts:

encoder:Hidden_Layer->encoder:Output_Layer
```

The connection between the hidden layer and this layer is represented as a subcontext called `encoder:Hidden_Layer->encoder:Output_Layer`. Type:

```
push encoder:Hidden_Layer->encoder:Output_Layer
```

then type `Info`:

```
encoder:Output_Layer% push encoder:Hidden_Layer->encoder:Output_Layer

encoder:Hidden_Layer->encoder:Output_Layer% Info

        Connection: encoder:Hidden_Layer->encoder:Output_Layer
                Full connection
                Size: 3 x 1
```

From within this subcontext you can open Unix pipes to the weights into each node that are represented by this connection.

Finally, the `source` command allows you to evaluate a file as if you had typed it into the MIGRAINES interface. In this directory there is a file called `encode.cmd`. Study this file. It extracts all of the weights and extracts the hidden layers values after cycling through the input data. Type `source encode.cmd`. There now exists ASCII data files that can be used for plotting to see the weights in the neural network. In addition, there is a file that contains the values of the hidden nodes. Since there are three hidden nodes, we can plot these in a 3-dimensional plot to see how well the neural network separates the input data in "hidden node space". Similarly, the weight vectors from the hidden layer to the output layer are also three element vectors. We can plot these in a 3-dimensional plot to see how the neural network "encodes" the data. If you have **gnuplot**, type `gnuplot encode.gnu` to see these plots. Although the neural network may converge to different solutions from different initial conditions, in most cases these 3 dimensional plots reveal that the hidden layer acts as a binary encoding. This can be seen by a box-like shape to the points in the weight vector space.

Read the file called `Learn` in the other examples directories. Executing `Learn` in the example directory typically causes the network to learn, followed by analysis, then display using **gnuplot**.

Chapter 4

DETECTOR TUTORIAL

Many neural network applications can be described as either detector or classifier problems. This tutorial outlines how to approach a detection problem with a backpropagation network. Although this tutorial uses an Aspirin neural network, the concepts and issues are the same for *any* detection problem. The reader is referred to an introductory text on detection and estimation for further reading.

This detector tutorial illustrates the sensitivity of the neural network to choices of output threshold and prior probabilities in the training data. The implications of these sensitivities have a large impact on the application of neural networks as detectors and classifiers. It will be shown that:

- Choice of an output threshold must be made based on receiver operator characteristic (**ROC**) curves.
- The probability of using an element from a particular class during training can significantly bias the performance statistics.

Typically, an analysis such as this would be parameterized in terms of signal to noise ratio (**SNR**), however, for convenience, in this tutorial we will parameterize only in terms of noise variance. After a network has been trained it will be tested with different amounts of additive Gaussian noise on the same data. The effect of this noise on the testing will be illustrated by histograms of the output distributions and **ROC** curves.

Two networks of the same architecture have been trained. Each network was trained with a different "mix" (probability of signal and the probability of noise/clutter):

1. Bias towards signal ($P(signal) = 10.0P(noise/clutter)$)
2. Bias towards noise/clutter ($10.0P(signal) = P(noise/clutter)$)

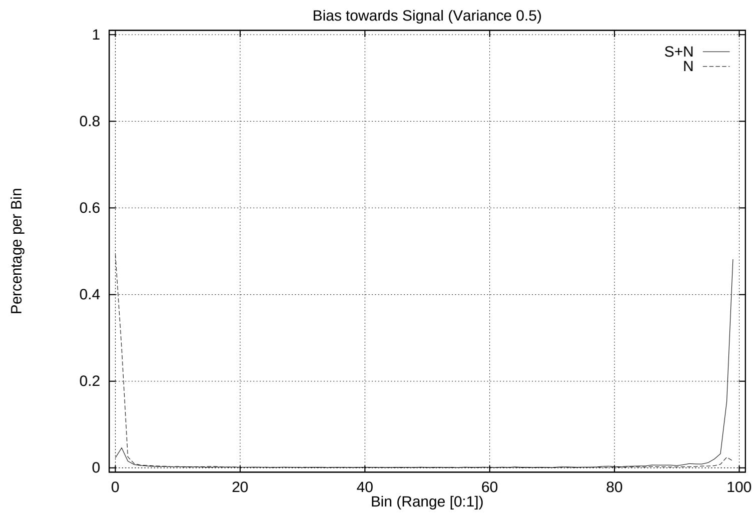


Figure 4.1:

The network architecture and data are taken from the Characters example (page 31), with the following small changes:

- The output layer has only 1 node, trained to output 1.0 for signal and 0.0 for noise/clutter.
- The "signal" class consists of the data for the letters C and D.
- The "noise/clutter" class consists of the data for the letters A and B, as well as a zero'd vector.

The networks are trained with a small amount of additive Gaussian noise, hence we have a two class detector problem with one class the signal plus noise and the other class just noise (noise and clutter, where the clutter consists of the noise corrupted letters A and B).

The effect of the different "mixes" can be seen in the histogram plots [1] of the output node (Figures 4.1, 4.2). The histograms for the network trained to be biased towards signal has a shift in the output distributions to the right (towards 1.0), while the histograms for the network trained to be biased towards noise/clutter are shifted to the left (towards 0.0). Therefore, depending on the "mix" during training, a particular threshold choice may have a very different

[1]The output node values under the condition of signal and noise/clutter were collected in 100 bin histograms.

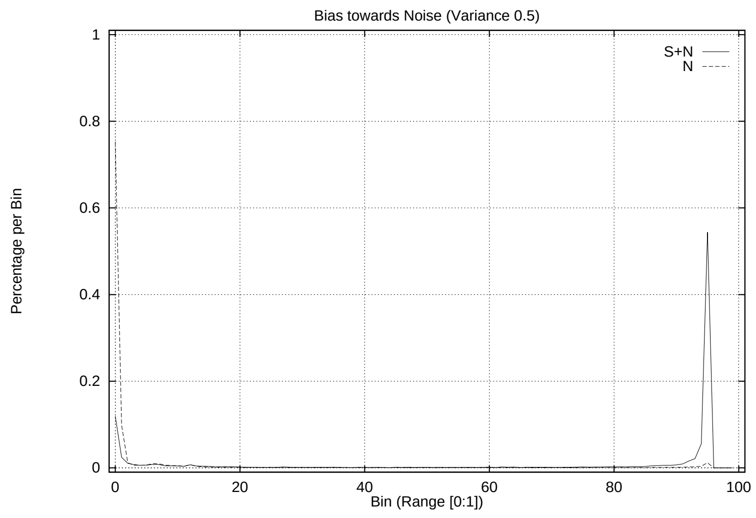


Figure 4.2:

performance in terms of probability of detection (**Pd**) and versus probability of false alarm (**Pfa**). Clearly one *cannot* arbitrarily choose an output threshold. Further, the sensitivity to the "mix" during training implies that consideration be given to the quality and proportions of training data.

In order to evaluate the networks trained with different "mixes" we need to access the performance in terms of **Pd** versus **Pfa**. This is done by creating an **ROC** curve. An **ROC** curve is created by varying the output threshold and plotting **Pd** on one axis and **Pfa** on the other. A set of curves are typically constructed, where each curve represents a different noise variance or **SNR**. Figures 4.3 and 4.4 each show a family of curves generated by calculating histograms under different noise conditions. The further a curve is to the upper left, the lower the noise variance. The squared off curve at the upper left reflects the performance under the condition of no noise at all and the square shape implies perfect separation. The lower right curve reflects the performance at extremely high noise variance and the fact that it runs along the diagonal implies no separation. The shape of the curves between the upper left and lower right describe the performance of the network under the conditions of increasing noise variance.

Note that the important criteria for training a neural network detector concern **Pd** and **Pfa** not the convergence to within some RMS bounds. It is quite likely that a network not "converge" during training to producing some small RMS error, yet be an excellent detector. This can happen when the output

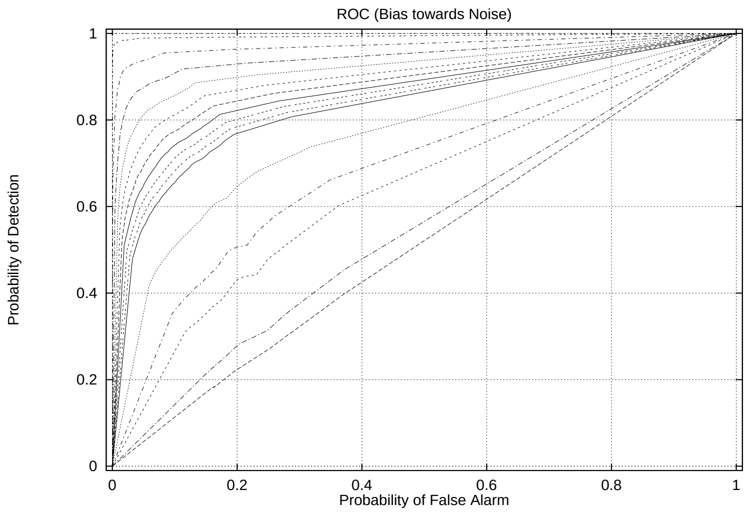


Figure 4.3:

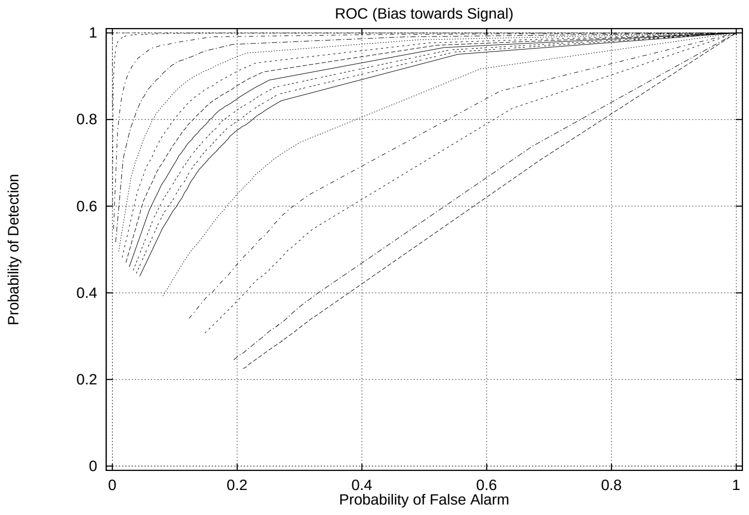


Figure 4.4:

distribution never comes close the target levels in the training data *but* there is still separation of the two classes.

The important points to be taken from this tutorial are:

- The threshold for the detector should be determined from an **ROC** curve. The decision of a particular threshold implies a decision about operating performance in terms of **Pd** verus **Pfa**.
- The prior probabilities of the training class must be considered during training. In most cases, the priors should be waited equally, however, it may be desirable to reflect the *true* prior probabilities of the classes in the operational environment within the training set.
- The notion of *convergence* of a neural network detector (or classifier) is not a useful concept, rather, the degree of *separation* is what is important.

Chapter 5

FREQUENTLY ASKED QUESTIONS

`Why is the number of iterations reported from loading a saved network different from the number of iterations that the network was actually trained?`
This a *feature* not a bug. There is a heuristic that is enabled by default which will only update the weights (and the iteration counter) if the error is less than a specified threshold (0.00005 by default). See section 7.4.3. This generally makes learning much faster by not updating the weights when there is very little error.

`Will A/M run on a PC under DOS or Windows?`
What an ugly thought ... but yes, it has been done.

`Can I use A/M generated code in a commercial application?`
Yes.

`Is it easy to add new code generators for new algorithms?`
Not really, but it can be done. Writing code generators is an ugly business. I think that the code *generated* by the current aspirin backprop compiler is reasonably clean as are the support libraries. Unfortunately, the code generator itself is quite complex.

`How do I add a neural network that I have trained into another program as a module?`
The easiest way to do this is to link the compiled object file into your program and manipulate the network as a single object. This is done in the **bayes** and **ntalk** examples (pages 30,32), refer to the file `Makefile.am`. In addition, most applications will not require learning, therefore when including Aspirin code in an application you can declare all black boxes

`Static->` and recompile the network. This will result in a much smaller executable, both in code and static storage.

`Can I move the weight files from one machine to another?`
Be careful about this! There are two issues here. First, since the weight files are binary, the word format and byte order must match between the two machines. If this is not the case then you need to produce an ascii version of the file with the `-AsciiDumpNoFmt` option, move the ascii file and reload it. The second issue is that even if you have compatible word formats or you moved the weights as ascii, the difference between the hardware and math libraries of the two machines may make the network behave differently. Validate all networks if you do this!

Chapter 6

EXAMPLES

The software distribution comes with a directory of examples. These examples are meant to illustrate different features of the Aspirin/MIGRAINES system. The examples *sonar* and *nettalk* are well known neural network applications. The data has been made available by the authors of these applications **for research purposes only** [1].

Each example has a group of README files that describe the usage of Aspirin/MIGRAINES. Read these carefully to understand how to run the programs generated by `bpmake`.

In each directory is a file called `Learn` which is an executable shell script. The `Learn` files are meant as examples of how to set up the networks to learn. Read the `Learn` files and then execute them. Upon successful completion of a learning run a dump file called `Network.Finished` will be written to the directory. The weights in the final network can be viewed through a plotting package. All the `Learn` files use Gnuplot3 as the plotting package, although almost any plotting package may be used. In particular, the patterns that the weights have formed are of interest.

Each example directory also contains a `Runs` directory which contains files of runs of the `Learn` script on different machines.

BAYES

This example shows that a backprop neural network can learn to approximate the optimal bayesian decision surface. Four normal random variables that represent different classes are used to train the network. Plots are generated showing the decision surface and the input distributions. The optimal bayesian decision surface follows the basin of the valleys between the different input random vari-

[1] This data was acquired via the **nnbench** mailing list, Scott Fahlman moderator.

ables. The plots show that the neural network has learned to place the decision surface along these valleys.

This example illustrates the use of a `user_init.c` file to create your own data generators.

CHARACTERS

This example contains data used in "An Analysis of Noise Tolerance for a Neural Network Recognition System" [Wieland, 1988]. The network in this example learns to recognize the letters **A,B,C,D** independent of rotation and in the presence of noise. This is a very simplified version of the experiment described in [Wieland, 1988].

This example illustrates the use of tessellation and multiple black boxes. This example does not *require* multiple black boxes. This is a situation where one might want multiple black boxes so, once trained, they can be used as modules in other networks.

The data used is in Type1 files generated on a Sun workstation (IEEE 32 bit big-endian floats).

ENCODE

This example is also taken from [Rumelhart, 1986]. The goal of this network is to map a binary set of inputs to the same binary set of outputs using a small number (e.g. $\log_2(n)$) of hidden nodes.

After training the network examine the weights connecting the hidden layer to the output. Many times these weights converge to a binary number encoding. If the magnitudes of the weights are ignored and just the signs considered, then you can read these as the binary numbers 0 through 7.

DETECT

This network detects a sign wave in noise. This example illustrates the use of the `user_init.c` file to create your own data generators.

IRIS

The data set to be used was published by Fisher [Fisher, 1936] and has been used widely as a testbed for statistical analysis techniques. The sepal length, sepal width, petal length, and petal width were measured on 50 iris specimens from each of 3 species, Iris setosa, Iris versicolor, and Iris virginica.

MONK's Problems

The MONK's problem were the basis of a first international comparison of learning algorithms. The result of this comparison is summarized in "The MONK's Problems - A Performance Comparison of Different Learning algorithms" [Thrun, 1991].

The MONK's problems are derived from a domain in which each training example is represented by six discrete-valued attributes. Each problem involves learning a binary function defined over this domain, from a sample of training examples of this function. Experiments were performed with and without noise in the training examples.

The example included is the backpropagation neural network trained on the three problem sets.

NETTALK

This example contains the data used in "Parallel networks that learn to pronounce English text" [Sejnowski, 1987]. The network is trained on a database of text to phoneme mappings. The original system described in [Sejnowski, 1987] could output the phonemes to a DECTalk[2] system for actual audio playback of the text. Read the file README.nettalk for a full description.

This example illustrates the use of the `user_init.c` file to build your own data generators. Also, a complete stand-alone application called `Performance` is included. This program links to the simulation and measures the performance of the network using the "Best Guess" metric.

PERF

This example is a large network. It is used only for benchmarking

RINGING

This example illustrates the use of an autoregressive [Leighton, 1991](see appendix K) network to learn a time varying function. In this case a exponentially decaying sinusoid impulse response.

SEQUENCE

This example illustrates the use of an autoregressive network to learn to recognize sequences of events. The training set consists of sequences of tokens. The

[2]Digital Equipment Corporation, *DTC-01-AA*.

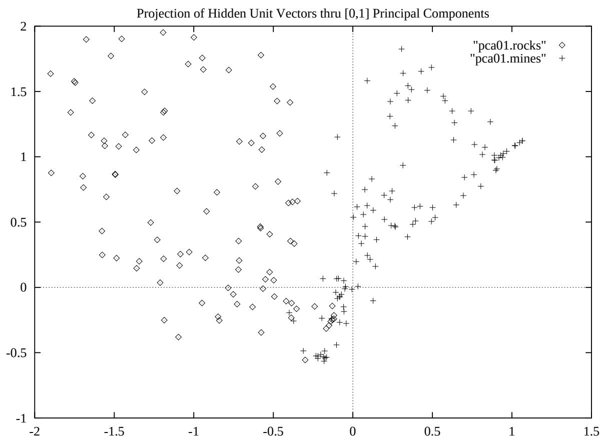


Figure 6.1: Principal Components Analysis of Hidden Layer for Sonar Example

network is trained using the AR backpropagation algorithm [Leighton, 1991] (see appendix K) to recognize a particular sequence within the training data. This cannot be done using the *same* architecture without AR nodes. It is tested for generalization on the test set.

A simple feedforward network that will do this is in the file ff.aspirin. Notice that delays are required on the input. This is the typical approach to recognizing sequences, but scales badly. If you need to recognize a very long sequence (or your data is highly sampled) then the number of delays (and 1st layer weights) can grow very large. The feedforward network has 10 ($\sim$30%) more weights than the AR network. Normally both input delays and AR delays are used in AR backprop networks for temporal recognition.

The idea of AR backprop is that the input delay window can remain small and AR "memories" can be used recognize the temporal characteristics of the signal.

The best solution for some problems is to use both (an ARMA network).

In this example, the AR network can separate [3] the data with no delays on the input. The feedforward network with delays separates the data very well, but requires an input retina as long as the sequence.

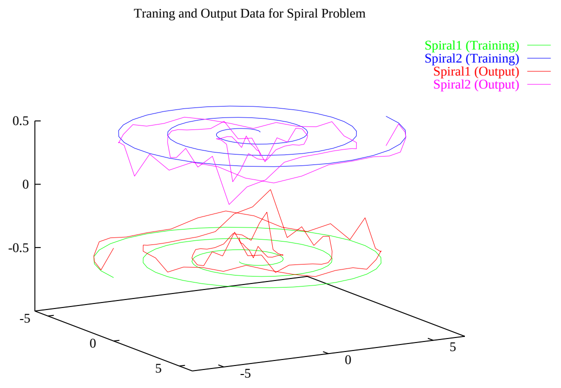


Figure 6.2: The Spiral Problem

SONAR

This example contains the data used in "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets" [Gorman, 1988]. In this example the neural network learns to separate processed sonar returns from rocks and mines on the ocean floor. Read the file `README.sonar` for a full description.

This example makes use of the `analyze` program to perform principal components analysis and canonical discriminate analysis to visualize the clustering performed by the hidden layer.

SPIRAL

This an example of a very hard pattern recognition problem. The data in this example are the x and y coordinates of two spirals. In the actual data file the coordinates for each spiral have an additional number associated with them which denotes to which spiral the coordinate belongs to (i.e., 0.5 for one spiral and -0.5 for the other). The two spirals coil three times around the origin and around one another. The goal of this example is to train the network to map an x,y coordinate into the proper spiral. The network is trained by giving it x and y and the target classification for some of the points along these intertwined spirals. This problem was originally conceived by this group at MITRE and is

[3]There exists a threshold that will separate the classes.

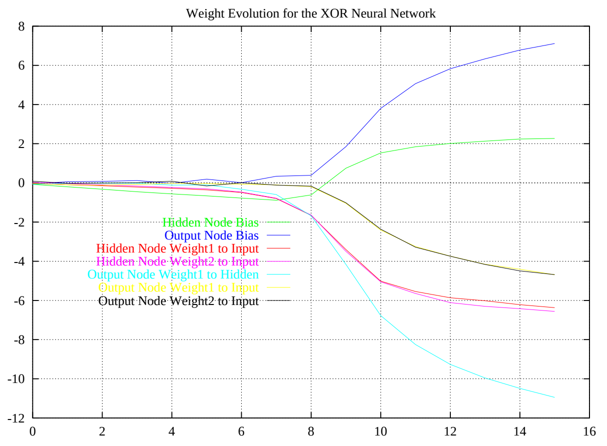


Figure 6.3: Weight Evolution (from Gnuplot 3)

described by Lang, *et al* in [Lang, 1988]. Read the file `README.spiral` for more details.

This example illustrates the use of user defined nodes and user defined error functions.

XOR

This is an example taken from [Rumelhart, 1986] whose historical significance dates back to Minsky and Papert's book `Perceptrons` [Minsky, 1969]. The largest criticism of the perceptron was the inability of this method to converge when the classes are not linearly separable. The exclusive-or (XOR) problem is to map 1 0 to 1, 0 1 to 1, 0 0 to 0 and 1 1 to 0. This problem is not linearly separable. Since the backpropagation method can solve this problem it has overcome a major shortcoming of perceptrons.

This example illustrates a simple Aspirin file used with an ASCII data file. Notice the skip level connections. The `.df` file controls the loading of the data files into the network. The MIGRAINES file `xor.cmd` causes the weights to be dumped to file as a function of learning iteration.

Chapter 7

ASPIRIN LANGUAGE

Aspirin is a declarative language used to specify neural network architectures. Associated with Aspirin are compilers that generate computer code to simulate the network(s) described in an Aspirin file. Aspirin enables a user to describe a neural network at a very high level. Commonly this code is used in conjunction with MIGRAINES to create a flexible, but efficient, neural network simulation. However, the Aspirin-generated simulations do not require the MIGRAINES interface. The code that is generated from the Aspirin description of a network can be linked into more application-specific systems.

The Aspirin language and the Aspirin compilers are decoupled in the sense that an Aspirin description may be understood by some compilers but not others. This allows the Aspirin network description language to be extensible without having to extend all of the associated compilers. Each compiler generates code to simulate a specific neural network paradigm. For example, an application may compile the Aspirin descriptions of two different neural networks using two different compilers. The generated code could then be linked into an application which controls the two networks, thus creating a single large heterogeneous system. Conversely, a particular network may be compiled by several different compilers with the goal of comparing various learning algorithms on a single network.

Aspirin is organized around the concept of a *black box* description of a neural network. A black box neural network is an abstract unit which (optionally) receives external input and (necessarily) produces some output. A complete neural network is one example of a black box. Another example of a black box is a subnetwork component of a larger, complex neural network system. The compilers generate functions that enable you to control each black box individually, as well as all of the black boxes declared in a single aspirin file. Code is also generated to enable programs to query the network about its current state and the functions it is currently using.

7.1 An Example Aspirin File

As a simple example of a Aspirin description we return to the encoder/decoder problem described on page 13. In this problem, the network takes eight input values, pushes that through a bottleneck of three nodes, and then recreates the input values on eight output nodes. An Aspirin file equivalent to the one discussed below can be found in the file `$NNTOOLS/examples/encode/encoder.aspirin`.

An aspirin description of the encoder/decoder network is:

```
DefineBlackBox Encoder
    {
         OutputLayer->   Output_Layer
         InputSize->       8
         Components->
             {
                 PdpNode Bottleneck 3
                     {
                         InputsFrom-> $INPUTS
                     }
                 PdpNode Output_Layer 8
                     {
                         InputsFrom-> Bottleneck
                     }
             }
    }
```

Let us examine this file line by line:

`DefineBlackBox Encoder`
: This statement tells the Aspirin compiler that a new black box named "Encoder" is being defined. For this network, as for most simple networks, a black box and a network are synonymous terms.

`{ and }`
: Curly brackets are used throughout Aspirin to denote the beginning and end of a definition. In this case the statements within the { } are the definition of the black box Encoder.

`OutputLayer-> Output_Layer`
: Since Aspirin does not require that the description of the network components be formatted or ordered in any particular manner, it is necessary to specify the name of the layer that produces the output of this black box. In this example the layer named `Output_Layer` is the output layer.

`InputSize-> 8`
: If the network receives external input, you must specify the number of

input elements. Whenever possible, the resulting application will check that the number of input data elements is correct [1], but for some special cases (such as user-defined data generators; see page 82) this is not possible. An incorrect `InputSize->` argument may cause unpredictable results.

`Components->`
: This section describes the make-up of this black box in terms of its layers and their connections.

`PdpNode Bottleneck 3`
: This means that the layer named `Bottleneck` is composed of three items of type `PdpNode` (sigmoidal nodes with output in the range of [0,1]). The size of a layer, as with the size of the inputs, can be expressed by the equivalent notation `[3]` or `[3 x 1]` to indicate that it is really a vector of three items.

`InputsFrom-> $INPUTS`
: This specifies the connections into the current layer, in this example `Bottleneck`. Without other qualification, the default is a full connection. That is, every node in `Bottleneck` receives input from each of the eight `$INPUTS`. The word `$INPUTS` is a special word in Aspirin (as are all words beginning with `$`) and indicates inputs external to the network. For example, these inputs might be supplied by user-provided data patterns or user-written data generators.

`PdpNode Output_Layer 8`
: In a similar manner, this defines a layer named `Output_Layer` composed of eight `PdpNodes`.

`InputsFrom-> Bottleneck`
: This declares that the `Output_Layer` receives input from (and is fully connected to) the layer `Bottleneck`. The order that layers are defined is not important. (It would have been equally correct to define `Output_Layer` with its reference to `Bottleneck` before `Bottleneck` had been declared).

Once an Aspirin file has been created, it can be compiled by typing `bpmake`, see section 9 on page 74.

[1] This assumes that the standard application program is used by compiling the application with **bpmake**. If you elect to write your own program to use the Aspirin-generated routines directly, it is your responsibility to perform checks on the number of input and output data elements.

7.2 Compiling an Aspirin File

Aspirin files are generally compiled with utility `bpmake`. `Bpmake` automatically generates a generic application linked to the MIGRAINES interface code. This section describes how `bpmake` compiles a simulation generated by an Aspirin compiler.

First `bpmake` (if necessary) calls the utility `aspirin` to compile the ".aspirin" file[2] to create a ".c" C file and a ".h" C header file. The Aspirin parser `aspirin`, which is located in the `$NNTOOLS/bin/$MACHTYPE` directory, takes two file names as its arguments: the name of the Aspirin file and a file name for the generated simulation files. As the third argument, an Aspirin compiler must be specified with the `-c` flag. Any additional flags for that specific compiler come last.

As an example, compile the Aspirin file `encoder.aspirin` by typing:

```
aspirin encoder.aspirin encoder -c backprop
```

In this example, two files are created: a header file named `encoder.h` and a C file named `encoder.c`. The header file is intended to be `#included` into application programs. It lists the functions that are generated. The C file contains the actual routines that an application program will use. The contents of these files depends on the particular compiler (indicated by the parameter following the `-c` flag). In this example the standard backpropagation algorithm has been specified.

The resulting code is then compiled by a C compiler and linked with application code. `bpmake` (see page 74) creates an application linked to MIGRAINES by executing something like:

```
gcc -D_sun_sparc__-D_sunos__ -O2     -I$NNTOOLS/include  \
    -c encode.c ;
gcc -D_sun_sparc__-D_sunos__ -O2     -I$NNTOOLS/include  \
    $NNTOOLS/migraines/bp/Backprop.c  -o encode encode.o \
    -L$NNTOOLS/lib/sun_sparc
    -lBpUi -lBpIo -lBpDatafile -lBpConverge  -lBpSim  -lAmOs -lm ;
```

Note that the files generated by Aspirin require that they are linked with a number of libraries and a UNIX math library (-lm).

If you are writing your own application code to use Aspirin-generated C routines, it is *very* important that the floating point guide to your system be read carefully. There are usually compiler flags that allow a C compiler to take advantage of floating point hardware. In addition to compiler flags, functions may need to be inserted into the application program to set up the floating point hardware. For example, on Sun 4 systems running Sun OS 4.0, you must call the function `abrupt_underflow_()`, once in your program, to force floating point underflows to zero. If this is not done, the exception is handled by the operating

[2]By convention, Aspirin files have a `.aspirin` suffix.

system, which will result in the program running slower than cold molasses. On Sun 3 systems be sure to have the environment variable `FLOAT_OPTION` set to `f68881`. In addition, some machines have high performance vector libraries (e.g., Cray, Convex) that Aspirin simulations require. These libraries need to be included when the application is compiled.

7.3 Format of Aspirin Files

An Aspirin file consists of one or more black box definitions. Each black box represents a network or subnetwork to be simulated. The code generated by the compiler controls each black box. All black boxes have names, components, and produce some output(s). They may also have data inputs from the outside world (inputs to the system) or from the outputs of other black boxes.

A black box can be connected to other black boxes by referring to the name of each of those black boxes. Since each black box is intended to truly be a "black box," a layer in one black box can only connect to the *outputs* of another black box.

The format of Aspirin files is reminiscent of C. There is no requirement to format the text in any particular manner, aside from providing code that is easily read by humans. Files may contain any number of C-style comments; that is, anything between `/*` and `*/` is ignored by the compiler as a comment ("recursive" comments are not allowed). In addition, the Aspirin compiler preprocesses the Aspirin description file with the C macro preprocessor (cpp). This allows C-style `#define`s of constants and macros to be used.

Names are specified by strings[3] beginning with letters `a-z` or `A-Z` followed by alpha-numeric characters[4]. Numbers are specified as strings using the characters `0-9`. Numbers may have either `+` or `-` prepended and may also use a decimal point.

7.4 Aspirin Syntax

This section describes the syntax used in Aspirin files. Not all of the constructs will necessarily work with all compilers. See the particular compiler description for information on the supported Aspirin constructs. The directories under `$NNTOOLS/examples` of the software distribution contain some examples of simple applications using Aspirin.

7.4.1 Black Boxes (`DefineBlackBox`)

Conceptually a black box is either a network or a subnetwork. An Aspirin file is composed of a series of one or more black box descriptions. A black box is

[3]Maximum length 14.

[4]The characters `!`,`$`,`/`,`-`,`_` and `.` are also allowed.

defined by a `DefineBlackBox` statement. A black box is required to have certain statements, and allowed to have many others. Each black box must contain a declared `OutputLayer->` and a list of `Components->`. Optionally, a black box may receive input from the outside world via `$INPUTS`. The `$` character indicates that the token is a special symbol internal to the current black box. The token `$INPUTS` refers to the inputs of the black box that the application program controls.

A black box is declared by the `DefineBlackBox` statement, followed by a name for the black box, and a declarative description of the component elements:

```
DefineBlackBox <name>
    {
         OutputLayer-> <layer-name>      /* required */
         Static->                        /* optional */
         InputSize-> <number-of-inputs>  /* optional */
         InputFilter-> <C function>      /* optional */
         OutputFilter-> <C function>     /* optional */
         Components->                    /* required */
             {
                  <list-of-component-layers>
             }
    }
```

Words bracketed by < and > are to be replaced by appropriate values; for example `<layer-name>` might be replaced by `Encoder`.

Output Layer (`OutputLayer->`)

All black boxes *must* have an output layer defined.

Static (`Static->`)

The `Static->` statement turns off the learning in the black box. This has the effect of significantly reducing the memory requirements for the simulation. This option is usually used for pre-processing black boxes (see section 7.4.1). Also, once a network has learned, recompiling the network with all black boxes declared `Static->` will result in a more efficient simulation that is suitable for inclusion within application-specific programs. If a black box is declared static which receives input from another *non-static* black box then the code generator may not be able to optimize the simulation due to the dependence. For example, the back-propagation code generator will produce code to propagate the error through the static black box to the *non-static* black box, although the weights in the static black box are never changed.

Input Data (`InputSize->`)

If the `InputSize->` statement is used, then this black box connects to the outside world. If this statement is not specified, then the layers in the black box must connect to the outputs of other black boxes defined in the Aspirin file.

Filtering Input Signals (`InputFilter->`)

If the `InputFilter->` statement is used, then the inputs will be processed by the C function named. For example, `InputFilter-> PowerSpectrum` might be declared. The user's C function called `PowerSpectrum` would be linked into the executable code. The simulator will pass the inputs to the `InputFilter->` which will return a pointer to the new inputs. These new inputs will be used as the actual inputs to the black box. The `InputFilter->` option is useful if you need to process input data (e.g., from a .df file, see page 75) but do not want to write your own data generators.

The `InputFilter->` C function should take three arguments and return a pointer to a floating point array. The first argument is a pointer to the inputs (floating point array pointer). The second argument is the width (an integer). The third argument is the height (an integer). The `InputFilter->` C function must be linked with the executable. This is most easily done by putting it in the `user_init.c` file (see page 82) and using the `bpmake` utility (see page 74).

Filtering Output Signals (`OutputFilter->`)

If the `OutputFilter->` statement is used, then the output layer values will be processed by the C function named. For example, `OutputFilter-> Threshold` might be declared. The user's C function called `Threshold` would be linked into the executable code. The simulator will pass the output layer values to the `OutputFilter->` which will alter the value of the output layer.

The `OutputFilter->` C function should take three arguments and return *void*. The first argument is a pointer to the outputs (floating point array pointer). The second argument is the width (an integer). The third argument is the height (an integer). The `OutputFilter->` C function is used to alter the output layer values in the array passed as the first argument. The `OutputFilter->` C function must be linked with the executable. This is most easily done by putting it in the `user_init.c` file (see page 82) and using the `bpmake` utility (see page 74).

Component Layers (`Components->`)

Each component of a black box is described by four fields: the node type, a unique name for that layer, the size of that layer (the dimensions), and connection information (a description of how that layer is connected to other layers). There are also three optional fields discussed later.

Node types are dependent on the compiler that is being used. For example, the backpropagation compiler in V7.0 of Aspirin has three kinds of sigmoidal nodes. (`PdpNode` or `PdpNode1` produces output in the range $[0, 1]$. `PdpNode2` has an output range of $[-0.5, 0.5]$, and `PdpNode3` in $[-1, 1)$]. Also supported are linear nodes, quadratic nodes, and user-defined nodes. These are described on page 58.

The size declaration describes how many nodes are in a layer and how they are arranged. Aspirin presently supports 1 and 2 dimensional arrays of nodes. Size can be described in one of the following ways:

1-dimensional layer:

width Example: `PdpNode layer1 100`
[*width*] Example: `PdpNode layer2 [100]`
[*width* x 1] Example: `PdpNode layer3 [100 x 1]`
[1 x *height*] Example: `PdpNode layer4 [1 x 100]`

2-dimensional layer:

[*width* x *height*] Example: `PdpNode layer5 [100 x 100]`

Connection information is described in terms of the inputs to each of the layers using the `InputsFrom->` statement. If a layer is specified without any additional connection information the default is a full connection between the two layers. For example:

```
Components->
    {
        PdpNode Layer-1 [100 x 20]
            {
                InputsFrom-> Layer-2 and Layer-3
            }
        PdpNode Layer-2 [5 x 4]
            {
                InputsFrom-> Layer-3
            }
        PdpNode Layer-3 [10 x 10]
            {
                InputsFrom-> $INPUTS
            }
    }
```

In this example `Layer-1`, which has 2,000 `PdpNodes` arranged in a 100 by 20 array, is fully connected to `Layer-2` and `Layer-3`. Therefore, each node in `Layer-1` receives 121 inputs: one input from each of the 20 nodes in `Layer-2`, plus one input from each of the 100 nodes in `Layer-3`, plus the node's bias weight.

The complete syntax for a black box component description is:

```
<node-type> <optional-node-args> <name> <size>
       <optional-order> <optional-arclip> <optional-initial-bias-spec>
    {
        <connection-specifications>
    }
```

The `<optional-node-args>` is used for information specific to the node type. The backpropagation code generator uses this for user defined nodes called `UserNodes` (see section 7.4.3).

The `<optional-order>` field has the form [5]:

```
Order = <integer>
```

refers to the number of linear feedback delays that are on the node. This is used to implement autoregressive nodes used in the autoregressive backpropagation algorithm (see Appendix K) [Leighton, 1991]. The default is 0, which implies that there are no linear feedback delays.

The `<optional-arclip>` is used to set the clipping on the autoregressive feedback delays.

```
ArClip = <number>
```

This *only applies to autogressive nodes.* For `PdpNode1, PdpNode2, PdpNode3` this defaults to ten times the dynamic range of the node. For all other node types there is no default and clipping is only done if `ArClip` is specified.

The `<optional-initial-bias-spec>` field has the form:

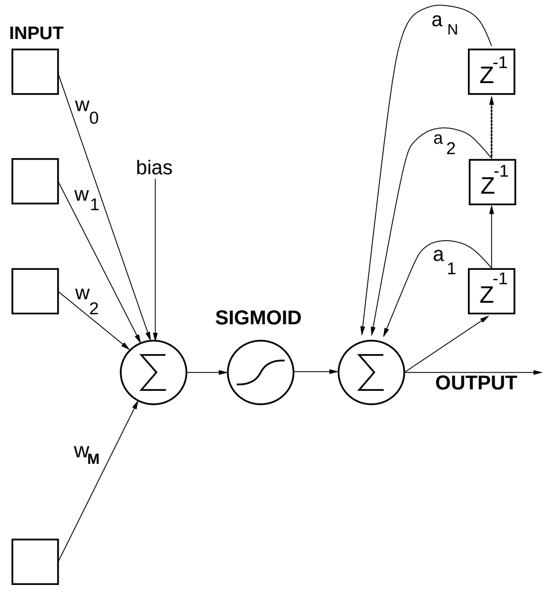
`Bias = <number>` *or* `Bias = <C function>`

where `<number>` is a real number and `<C function>` is the name of a user-defined C function that is linked into the simulation. This C function should take a single integer argument (the node index) and return a floating point number. If a real number is specified then all bias weights in that layer are set to the specified bias. If a C function is specified, then this is called repeatedly for each node's bias weight at initialization time. The ability to set a node's bias weight is very useful for creating black boxes that act as preprocessing filters (see page 51).

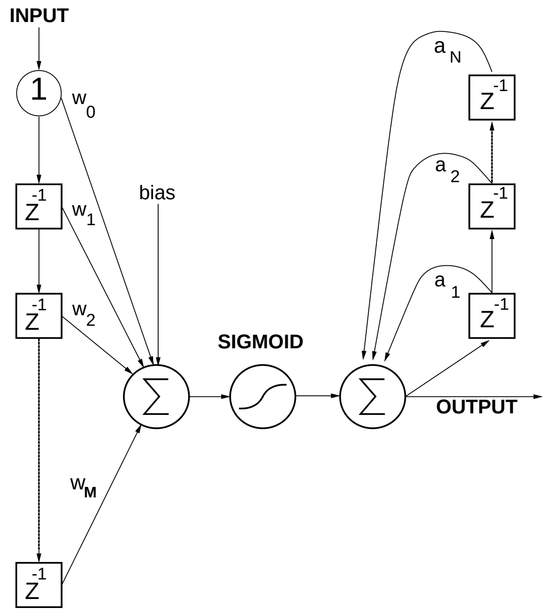
Connection Specification

The `<connection-specifications>` field describes the inputs into the current layer. The statement `InputsFrom->` precedes a list of layer names, the names of other black boxes, or the `$INPUTS` token separated by the token `and`. Each layer (or black box) in this list has a full connection to the current output of the layer unless additional connection information is specified. The format of a complete connection specification is:

[5] $0 \leq integer \leq 2$



AR Model Neuron



AR Model Neuron as a Digital Filter

Figure 7.1: Autoregressive Neurons

```
InputsFrom-> <connection> and <connection> and ...
```

where each `<connection>` is specified by

```
  <name> <optional-time> <optional-connection-description>
```

where the field `<name>` is the name of the source layer, black box or `$INPUTS` token and the fields`<optional-time>` and `<optional-connection-description>` are described on pages 46 and 47, respectively.

In a multiple black box network, a layer may connect to the output of any other black box. This is denoted by prepending '`!`' to the name of that black box. For example:

```
 PdpNode layer1 [5 x 5]
  {
             InputsFrom-> !BlackBox5
  }
```

This fully connects `layer1` to the output of a black box named `BlackBox5`. Connections can only be made to the output of a black box, not to any of the internal layers.

Delayed Node Values

Generally, the input to a layer is the current output of another layer or black box. But for some applications the network is processing data where the temporal nature of the data is itself important. For these applications it may be desirable for a layer to receive as inputs the values that were produced at a previous time [Waibel, 1987].

Aspirin provides this capability by allowing you to specify the time inputs to a layer are to be sampled. This is specified by following the token `$INPUTS`, layer name, or black-box name by the `<optional-time>` field. The format for `<optional-time>` is: `@ (Time = <delay>)`, where `<delay>` is an integer that is less than or equal to zero. Not specifying a delay is equivalent to saying `@ (Time = 0)`. To see the previous value you would use `(Time = -1)`; etc. Note that `<delay>` may not be a positive number (thus referring to future outputs of a node).

A moving average connection may be implemented using this delay feature by specifying a *range* of temporal values to connect the layer:

```
            @ (Time = [<initial delay>,<final delay>]).
```

For example, if a you wanted to connect to a layer called `Hidden1` with a moving average style connection you might express this by using: `InputsFrom-> Hidden1 @ (Time = [0 , -10])`.

Aspirin creates storage to hold all of the values for every delay. While there is no restriction on how far back you can have the network store values, memory requirements can quickly become prohibitive if you are working with large networks and long delays.

The following Aspirin code describes a network with delays:

```
DefineBlackBox Delay_Example
 {
        OutputLayer-> OUT
        InputSize-> 32
        Components->
         {
                PdpNode FIRST  3
                 {
                   InputsFrom-> $INPUTS and
                                /* a moving average style connection */
                                $INPUTS @ (Time = [0,-20])
                 }  /* end FIRST */
                PdpNode OUT  1
                 {
                   InputsFrom-> FIRST and
                                FIRST @ (Time = -1) and
                                FIRST @ (Time = -2)
                 }  /* end OUT */
         }  /* end components */
  }  /* end Delay_Example */
```

Each node in the layer FIRST receives 65 inputs: 32 from the current input to the network, 32 from the moving averages of the last 21 inputs to the network and one from the bias weight. Similarly, the node in layer OUT receives 10 inputs: its bias weight and the three outputs of layer FIRST at the current time, plus FIRST's outputs the last time the network fed forward, plus FIRST's outputs the time before that.

Connection Description

Each connection description, if it exists, begins and ends with parentheses:

```
( <optional-connection-description> )
```

Within these parentheses you describe the kind of connection and declare an optional initialization procedure. Currently, full connections and tessellation (i.e., limited receptive fields, see section 7.4.1) are the only connection types implemented, but in the future other types may be included (e.g., second order (quadratic), random, explicit enumeration).

Tessellation

Tessellation (tiling) connections allow each node in a destination layer to be connected only to a subset of the nodes in a source layer. These connections are created by a *regular* tiling of the source layer by the nodes in the destination layer. Tessellation creates nodes with limited receptive fields.

A tile of connections (the receptive field), for each destination node, must connect to a rectangular patch in the source layer. These rectangular tiles are described by their width, height, `Xoverlap`, `Yoverlap`, initial `Xoffset`, and initial `Yoffset`. Nodes in the destination layer must connect to some (not necessarily all) of the nodes in the source layer. The tiling cannot exceed the dimensions of the source layer; that is, connections may not "hang over the edges" of the source layer.

The following Aspirin description and figure 7.2 illustrate the use of a one-dimensional tessellation of the inputs by a layer of 3 nodes. Each node has 8 inputs. The "receptive field" of each node overlaps with its neighbors by 4 (except the end nodes). The tiling begins with the first node, node 0.

```
DefineBlackBox Tessellation_Example
 {
        OutputLayer-> LAYER
        InputSize-> [16 x 1]
        Components->
         {
                PdpNode LAYER 3
                 {
                   InputsFrom-> $INPUTS ( with a [8 x 1] Tessellation
                                          using a 4 Xoverlap )
                 }  /* end LAYER */
         }  /* end components */
  }  /* end Tessellation_Example */
```

It is not necessary for the tiling to start with the first node. The following Aspirin description, shown in figure 7.3, begins the tiling with the second node in the source layer.

```
DefineBlackBox Tessellation_Example_With_Offset
 {
        OutputLayer-> LAYER
        InputSize-> [16 x 1]
        Components->
         {
                PdpNode LAYER 3
                 {
                   InputsFrom-> $INPUTS ( with a [8 x 1] Tessellation
```

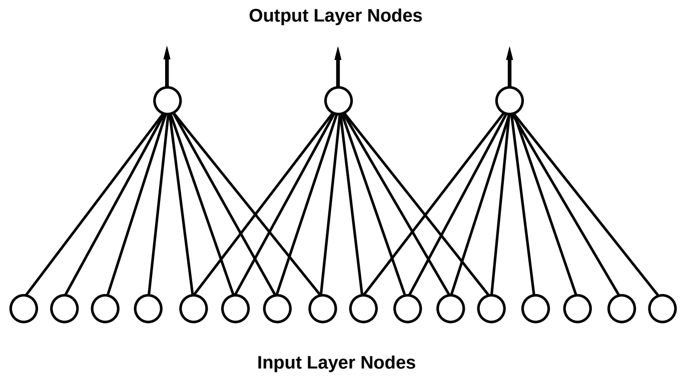


Figure 7.2: Using a 1D Tessellation

```
                                          using a 5 Xoverlap
                                          with a 1 Xoffset )
                }  /* end LAYER */
        }  /* end components */
  }  /* end Tessellation_Example_With_Offset */
```

Note the addition of the phrase `with a 1 Xoffset` in the connection description for `LAYER` and the difference between figures 7.3 and 7.2. In this example the first and the last input are not connected to anything.

Aspirin enables you to make two-dimensional tessellations:

```
DefineBlackBox Two_D_Tess
 {
        OutputLayer-> LAYER
        InputSize-> [6 x 6]
        Components->
         {
                PdpNode LAYER [2 x 2]
                 {
                        InputsFrom-> $INPUTS ( with a [2 x 2] Tessellation
                                               using a 0 Xoverlap
                                               and a 0 Yoverlap
                                               with a 1 Xoffset
                                               and a 1 Yoffset )
                 }  /* end LAYER */
        }  /* end components */
 }  /* end Two_D_Tess */
```

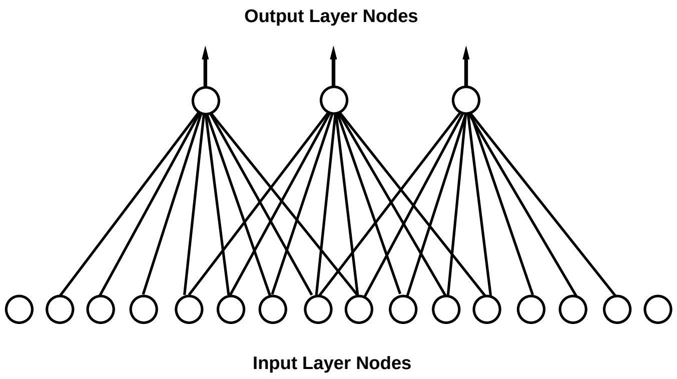


Figure 7.3: 1D Tessellation with Offset

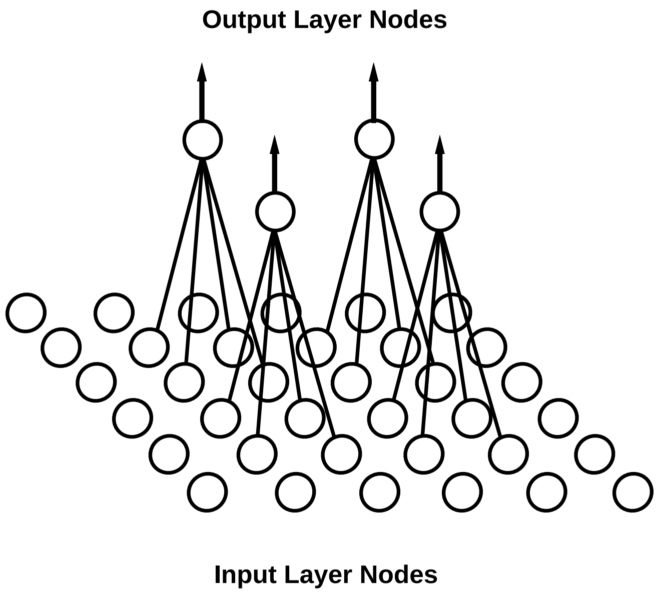


Figure 7.4: Using a 2D Tessellation with Offset

Figure 7.4 is similar to figure 7.3, but generalized for two dimensions.

It is possible for you to create networks that use tessellations of tessellated layers to form pyramid-style networks:

```
DefineBlackBox bb3
 {
        OutputLayer-> OUPUT-LAYER
        InputSize-> [512 x 512]
        Components->
         {
                PdpNode OUTPUT-LAYER [2]
                 {
                    InputsFrom-> LAYER2 and LAYER1
                 }  /* end OUTPUT-LAYER */
                PdpNode LAYER2 [3 x 3]
                 {
                    InputsFrom-> LAYER1 ( with a [15 x 15] Tessellation
                                          using a 8 Xoverlap
                                          and a 8 Yoverlap )
                 }  /* end LAYER2 */
                PdpNode LAYER1 [29 x 29]
                 {
                    InputsFrom-> $INPUTS ( with a [32 x 32] Tessellation
                                           using a 16 Xoverlap
                                           and a 16 Yoverlap )
                 }  /* end LAYER1 */
         }  /* end components */
  }  /* end bb3 */
```

In the above network, LAYER1 tiles the input and LAYER2 tiles LAYER1. Tessellated networks learn much faster than more fully connected networks; however, fewer weights mean a lower resolution.

Initialization of Weights and Shared Tessellations

Two optional parameters to a tessellation declaration are:

- whether or not this is a shared tessellation (default: *not shared*)
- whether or not there is a user-defined initialization function (default: initialize weights to small random values [6])

A shared tessellation is a tiling exactly as described above but where the weights for each node are the *same* weights. It must be emphasized that these

[6]The exact values depends on the compiler.

are not weights with the same value, but literally the exact same weights. Any change to the input weights to one node affects the input weights to all of the nodes. These shared weights may be learned or they may remain fixed, as would be the case for an input filter. This is an even greater reduction in storage over simple tessellations, and has the interesting advantage that the network is automatically shift invariant at this level. Clearly it also limits what the network can compute, but in filtering or preprocessing applications this is often exactly what is desired.

You may optionally specify a C function by name that will be used to initialize the weights. If the connection is to a one-dimensional layer, this function should take a single integer argument. If the connection is to a two-dimensional layer, this function should take two integer arguments. These arguments correspond to the x and y index of a weight in the particular weight matrix. In both the one- and two-dimensional cases, the indices range from 0 to *dimension-size*−1. This C function must return a single floating point number which will be used as the (initial) value for that weight. The syntax is:

```
initialized with <user-supplied-C-function>
```

As an example, consider the following black-box description which creates a filter for preprocessing an image (presumably before passing it to a neural network in a subsequent black box, not shown here):

```
DefineBlackBox Laplace
 {
   OutputLayer-> Filter
   InputSize-> [32 x 32]
   Components->
    {
         LinearNode Filter [30 x 30] Bias = 0.0
          {
                 InputsFrom-> $INPUTS ( with a Shared [3 x 3] Tessellation
                                        using a 2 Xoverlap and 2 Yoverlap
                                        initialized with a Laplacian )
          }
    }
 }  /* end Laplace */
```

where `Laplacian` is a user-defined C function of the form:

```
/*******************************************************
 **    Laplacian: Set arc weights to 3x3 Laplacian    **
 *******************************************************/

/**
 **  Arc weights for a 3x3 Laplacian.
 **/

float Laplace_Kernel[3][3] = { { -1.0, -1.0, -1.0 },
                               { -1.0,  8.0, -1.0 },
                               { -1.0, -1.0, -1.0 } } ;

/**
 **  Function to return the weights.
 **/

float Laplacian(x, y)
   int x, y;
{
  return( Laplace_Kernel[x][y] );
}  /* end Laplacian() */
```

This definition would be placed in the `user_init.c` file (see page 82) which is automatically compiled and linked into an application when using `bpmake` (see page 74).

Full Tessellation Syntax

The full syntax for a tessellation description is:

```
  ( with a <optional-'Shared'-token> <size> Tessellation
    <optional-overlap>  <optional-offsets> <optional-initialization> )
```

- The `<size>` is:

  ```
  [ <width> x <height> ]
  ```

  (or `[ <width> ]` or simply `<width>`)

- The `<optional-overlap>` is specified (if y dimension is 1) by:

  ```
  using a <integer> Xoverlap
  ```

  or (if the y dimension is greater than 1) by:

  ```
  using a <integer> Xoverlap and a <integer> Yoverlap
  ```

- The `<optional-offsets>` is specified (if y dimension is 1) by:

  ```
  using a <integer> Xoffset
  ```

  or (if the y dimension is greater than 1) by:

```
        using a <integer> Xoffset and a <integer> Yoffset
```

- The `<optional-initialization>` is specified:

```
        initialized with <user-supplied-C-function>.
```

7.4.2 Loading Black Boxes

All Aspirin compilers allow you to save the state of the network in a "dump" file. The Aspirin language allows the state of individual black boxes to be extracted from a dump file during network initialization. This facility enables you to build new networks from black boxes created in other networks via the `LoadData` command.

For example, consider a large network composed of two black boxes. After the network has been trained for a number of hours, the user decides to change one of the two black boxes. The network could be allowed to start learning again from scratch and make the user wait another couple of hours, or alternatively the unchanged black box could be loaded from a dump file and only the second black box created from scratch. This approach will generally speed convergence of the network's weights due to the previously learned weights of the first black box.

When loading a black box from a dump file, the black box in the Aspirin file may have a different name than the black box stored in the dump file. However, all other names and sizes internal to the two black boxes *must* be identical. For example:

```
LoadData into detect_Red_Plane from detect_MIG29 in Detect_MIG29.dump
```

At initialization (that is when `<aspirin-file-name>_init_network()` is called) the black box called `detect_MIG29` in the file `Detect_MIG29.dump` will be located and the weights will be loaded into a black box defined in the current aspirin file called `detect_Red_Plane`. Note that the names of the black boxes were different (`detect_Red_Plane` and `detect_MIG29`); this is allowed, but otherwise they must be identical.

The syntax is:

```
LoadData into <black-box-name> from <black-box-name> in <path-name>
```

The `LoadData` statements should be placed at the end of the Aspirin file.

7.4.3 Compilers

Aspirin can support different types of networks and different learning algorithms by having a library of different compilers. V7.0 of Aspirin contains one backpropagation compiler in its library. This compiler allows for a large number of optional flags which enable many of the more common backpropagation network types to be created.

Backpropagation

This compiler implements the error backpropagation (BP) [Rumelhart, 1986] algorithm and the autoregressive backpropagation [Leighton, 1991] algorithm. The BP algorithm can use only feedforward networks [7] Delays and moving averages are supported. The method of learning involves presenting to the network an input pattern and then propagating the values forward through the network. The final values of the output vector are compared to a target vector and an error signal is generated. The error signal is propagated back through the network, changing each weight proportional to the degree that weight contributed to the error. This is repeated for all of the input/output patterns many times. The final goal is to train the network to reproduce the input/output mappings to within an acceptably small error.

This compiler can be invoked from aspirin with the flag `-c backprop` in the command:

```
aspirin network.aspirin network -c backprop <backprop-flags>
```

The following *optional* `<backprop-flags>` are supported with the backpropagation compiler:

`-bpthreshold <float>`
: This flag causes the simulator to *not* update the weights if the total mean error (at the output of the network) is less than `<float>`. The result is that the simulation spends its time adapting to only those patterns which cause significant errors.

`-biasdfdt <float>`
: This flag causes the simulator to add a small random number `<float>` to the calculation of the derivative. This results in an error signal bias, which usually speeds convergence, see [Fahlman, 1988].

`-interface`
: This flag causes the code generator to produce high level network controlling functions with generic names. This is used with the generic application code that the `bpmake` utility compiles. All of these functions begin with the name `network_`.

Your Own Error Calculations (`ErrorFunction->`)

You may replace the mean-squared-error calculation with your own error function. This keyword only applies to BP simulations. To do this, just as with the `InputFilter->`, you specify a C function to be linked with the simulation. For

[7]The authors are aware of, and have some involvement in, the recent recurrent neural network research, but this version of the Aspirin/MIGRAINES does not address these networks.

example, you might declare `ErrorFunction-> CrossEntropy`. The C function `CrossEntropy` will be called every time the output error calculation is required.

The `ErrorFunction->` C function should return a float which corresponds to the total error. For example, if your error function is mean-squared-error, then this function would return the sum of the squares divided by 2. The C function should take four arguments. The first is a pointer to an array of floats. This argument is the target vector. The second is a pointer to an array of floats. This argument is the output vector. The third is a pointer to an array of floats. This argument is the credit vector. The credit vector *must* be filled with the credit for the corresponding nodes (i.e., $-\frac{\partial E}{\partial o_i}$). The fourth argument is an integer. This argument is the length of the vectors. The `ErrorFunction->` C function must be linked with the executable. This is most easily done by putting it in the `user_init.c` file (see page 82) and using the `bpmake` utility (see page 74).

The Update Interval

By default the weights are updated after each presentation of data. If the `UpdateInterval` is specified for the black box, then the weights are updated every specified interval. For example:

```
DefineBlackBox xor
{
      InputSize-> 2
      OutputLayer-> Output
      UpdateInterval-> 4  /* because there are 4 patterns */
        .
        .
        .
        .
}
```

This interval is usually set to the number of patterns in the training set (called an epoch). However, the authors' experience suggests that weight update after each presentation (the default) is the most effective, however there are situations when the weight changes should be averaged over a number of patterns for better performance.

Experience suggests that the best way to train a network is to begin with a low update interval and resume training with gradually longer and longer update intervals.

Line Search

The line search option is used to adaptively choose a learning rate. This option is typically used with the conjugate gradient option (see 7.4.3). One normally uses the line search to fine tune a network after it has converged. The normal methodology is to train the network using an update interval smaller than an entire epoch to insure quick convergence and then to fine tune with an update interval of an epoch using a line search (possibly with conjugate gradient optimization). To specify a line search the following option should appear in the aspirin file *outside* the black box descriptions:

```
LineSearch <update interval> <timeout>
```

where `<update interval>` specifies the update interval for the whole network and `<timeout>` specifies the maximum number of attempts to decrease the error in the line search. You may also specify:

```
LineSearchVerbose <update interval> <timeout>
```

which is the same as the `LineSearch` option but will print out information every time the weights are updated. The line search is an *inexact* line search, where the current learning rate is used to change the weights. If that change results in a *decrease* in the error for input data used in that update interval, then the weight change is accepted. If the error is *increased* then a new weight change is made that is one half the last weight change. This process is repeated until there is a decrease in the error or it is repeated `<timeout>` times. If the line search is repeated `<timeout>` times then the last learning rate used is accepted.

A heuristic has been added to change the initial learning rate. The simulation is begun using a user specified learning rate. If the first weight change is accepted five times in a row, then the initial learning rate is doubled. If the line search times out five times in a row then the initial learning rate is halved. This heuristic results in an optimization that proceeds quickly down local minima. Therefore, it is recommended that this only be a fine tuning step after the network has converged to some nominal level of performance.

The line search is implemented by keeping a table as large as the update interval of pointers to the input and target data. This implies that you *must* ensure the data patterns are in separate segments of memory. You *cannot* use the line search if you:

1. Use the -u or -n options to add noise to `bpmake` generated simulations. This is because each input vector is copied to a single buffer and noise is then added. There is no way for the simulation to keep track of the data.
2. Use a user defined generator that does *not* set the inputs with pointers to different segments of memory.

Conjugate Gradient

To specify a conjugate gradient optimization the following option should appear in the aspirin file *outside* the black box descriptions:

```
ConjugateGradient
```

This *must* be used with the line search option (see section 7.4.3). This should be done with the update interval set to an epoch.This option should only be done to fine tune the network.

Node Types

Six kinds of nodes are currently supported in the backpropagation paradigm:

`PdpNode` *or* `PdpNode1`
: Takes a weighted sum of its inputs, plus a bias weight and passes it through the sigmoid transfer function:

$$F(x) = \frac{1}{1+e^{-x}}$$

resulting in an output value in the range $[0, 1]$. This is implemented as a table look-up, [8] with *no* interpolation. If you require more accuracy define a `UserNode`.

`PdpNode2`
: Similar to PdpNode, but the outputs are offset by -0.5 in order to be symmetric around zero. The output values are in the range $[-0.5, 0.5]$. This is implemented as a table look-up, with *no* interpolation. If you require more accuracy define a `UserNode`.

`PdpNode3`
: Similar to PdpNode, but with a range of $[-1, 1]$. This is implemented as a table look-up, with *no* interpolation. If you require more accuracy define a `UserNode`.

`LinearNode`
: Takes a weighted sum of its inputs, plus a bias weight and returns this sum. You can think of this as a `PdpNode` without a transfer function. Be warned, these nodes have an unbounded output range!

`QuadraticNode`
: Takes a weighted sum of its inputs, plus a bias weight and returns this sum squared. This is often helpful when you do not want the sign of the input to matter, or dealing with second order properties such as size. Be warned, these nodes have an unbounded output range!

`UserNode`
: These nodes enable the user to specify the name of a function for computing the transfer function and the name of a function for computing

[8] A table of 1024 values is used.

the derivative of the transfer function. In order to use the `UserNode` you must specify the `<optional-node-arguments>`, which, in this case are the name of a C function that is the transfer function and the name of the C function that is the derivative of the transfer function. The format is:

```
    UserNode <C-transfer-function> <C-deriv-of-transfer-function>
```

Both of these functions should take a single argument of type `float`, which will be the weighted-sum input to the node, and return a single argument of type `float`, the output of the node. Aspirin will use these names when writing the code for the network. The application programmer must insure that these functions are properly defined and able to be linked in. The recommended place for these function definitions is in the `user_init.c` file (see page 82), which is automatically compiled and linked into an application by `bpmake`. For example, a cubic transfer function could be defined in the `user_init.c` file by the code:

```
/*****************************************************************
 **    Definitions for nodes with a cubic transfer function    **
 *****************************************************************/

/**
 **  Transfer function (for use during propagation forward)
 **/

float Cubic_Node_Function( x )
      float x;
{
  return( x * x * x );
}  /* end Cubic_Node_Function() */

/**
 **  Derivative of the transfer function (for backprop/learning)
 **/

float Cubic_Node_Function_Prime( x )
      float x;
{
  return( 3.0 * x * x );
}  /* end Cubic_Node_Function_Prime() */
```

Once a cubic node has been defined in the `user_init` file, it can be used in the same way that any other node type would be used. For example:

```
UserNode Cubic_Node_Function Cubic_Node_Function_Prime
  Cubic_Layer [3 x 3]
    {
       InputsFrom-> $INPUTS
    }
```

The tessellated connections, as described above, are supported for all six node types.

Compilation

The compiler generates two files: a header file with a `.h` suffix and a C file with a `.c` suffix. The C file contains the functions declared in the `.h` file and is linked to application code. The application code should `#include` the generated `.h` file. For example, your application code might contain the line:

```
#include mynetwork.h     /* generated from mynetwork.aspirin */
```

Note that the generated files need header information from the `$NNTOOLS/include` directory. This is done with the `-I` flag for the C compiler. Also, common routines are located in libraries in the directory `$NNTOOLS/aspirin/lib/$MACHTYPE`:

```
libAmOs.a
libBpSim.a
```

which must be linked with the final application. For example, the following will generate C simulation code in "`speech.c`" from the Aspirin file called `network.aspirin` and then compile it with full (e.g., `-O2`) optimization[9] and link it with the math library (e.g., `-lm`):

```
aspirin network.aspirin speech -c backprop
gcc -D_sun_sparc__-D_sunos__ -O2  -I$NNTOOLS/include -c speech.c
gcc -D_sun_sparc__-D_sunos__ -O2  -I$NNTOOLS/include Speech.c -o Speech \
 speech.o -L$NNTOOLS/aspirin/lib/sun_sparc -lBpSim.a -lAmOs.a -lm
```

The backpropagation compilers generate code that calls functions in the math library, therefore the final compile *must* link to the math library with `-lm`. If you are not interested in writing application code, the `bpmake` utility will do this automatically.

The `.h` file contains declarations of the exported functions available to the application. Each black box has a set of functions that controls it. The name of the particular black box (denoted here as `<bb-name>`) is prepended as part of the name of each controlling function. The following are descriptions of the functions generated by the backpropagation compiler to control each black box. As an example, if the name of your black box (as declared immediately

[9]The `-O2` flag is not required, but is highly recommended.

after the `DefineBlackBox` statement) was `xor`, a function generated would be `xor_set_input()`.

`<bb-name>_set_input`
: Takes one argument, a pointer to a vector of floating point numbers with length `InputSize->`.

`<bb-name>_get_input`
: Takes no arguments. Returns a pointer to a vector of floating point numbers which is the current input. Note that before the input has been set this will return `(float *)NULL`.

`<bb-name>_set_target_output`
: Takes one argument, a pointer to a vector of floating point numbers the size of
`OutputLayer->`. This is used to generate the error signal at the output.

`<bb-name>_get_target_output`
: Takes no arguments. Returns a pointer to a vector of floating point numbers that is the current target output. Note that before the target output has been set this will return `(float *)NULL`.

`<bb-name>_propagate_forward`
: Takes no arguments. Propagates the activations through the network. Must have used `<bb-name>_set_input` before calling.

`<bb-name>_calc_error`
: Takes no arguments. This returns a floating point number corresponding total error at the output layer.

`<bb-name>_calc_grad`
: Takes no arguments. Returns void. Calculates the black box's contribution to the gradient.

`<bb-name>_update_weights`
: Takes a single float as an argument which is typically the inert¡ia. Returns void. Updates the weights of the black box.

`<bb-name>_get_backprop_counter`
: Takes no arguments. Returns an integer which is the number of times
`<bb-name>_propagate_backward` was called.

`<bb-name>_set_backprop_counter`
: Takes an integer argument which resets the number of times
`<bb-name>_propagate_backward` was called.

In addition, the compiler generates a set of functions that globally control all of the black boxes. The name of the Aspirin file (denoted as `<aspirin-file-name>`) is prepended as part of the name of the controlling function [10]. The following are the functions generated by the backpropagation compiler for global control of networks defined in an Aspirin file:

`<aspirin-file-name>_init_network`
: This takes no arguments. This function MUST be called before any other functions (except functions that set parameters). Returns 0 if there were no errors, an error code otherwise. If a non-0 number is returned, the function `<aspirin-file-name>_error_string` can be called which returns a string describing the error.

`<aspirin-file-name>_set_learning_rate`
: This function takes one floating point argument that is the learning rate of the network described by the Aspirin file. This defaults to 0.2. This function returns `void`.

`<aspirin-file-name>_set_inertia`
: This function takes one floating point argument that is the inertia (sometimes called the momentum) term. This defaults to 0.95. This function returns `void`.

`<aspirin-file-name>_set_random_init_seed`
: This function takes one long integer argument that is the seed for the random number generator used to initialize the weights. This value defaults to 123. Since each run defaults to starting with the same random seed, it will take the same number of iterations to converge, etc. To use different random number generator seeds (generated from the clock) you could use `<aspirin-file-name>_set_random_init_seed(time(0))`. This function returns `void`.

`<aspirin-file-name>_set_random_init_range`
: This function takes one floating point argument that is the range about zero for the numbers that initialize the weights and thresholds. Defaults to 0.1 with weights uniformly distributed in [-0.1, 0.1). This function returns `void`.

`<aspirin-file-name>_dump_network`
: This takes a string as an argument that will be the filename of the dump file. The state of the network's weights are written to a file. Returns 0 if there were no errors, an error code otherwise. If a non-0 number is returned, the function `<aspirin-file-name>_error_string` can be called which returns a string describing the error.

[10] You should only use Aspirin filenames that can be used in C function names.

`<aspirin-file-name>_load_network`
: This function takes a filename of a file previously dumped network with the name `<aspirin-file-name>_dump_network`. Returns 0 if there were no errors, an error code otherwise. If a non-0 number is returned, the function
`<aspirin-file-name>_error_string` can be called which returns a string describing the error.

`<aspirin-file-name>_asciidump_network`
: This function takes no arguments. It prints the information included in the dump file to the screen. This function returns `void`.

`<aspirin-file-name>_query_network`
: This function takes two integers as its arguments. The first integer specifies the black box index, and the second integer specifies the layer index. (A programmer should not rely on any relationship between the order of black boxes or layers in an Aspirin file and the indices assigned to them. Aspirin may rearrange black boxes and layers for greater efficiency.) The function returns an `LB_PTR` (layer buffer pointer) with all of the information about that layer. This allows access to the simulation's data structures and functions [11].

`<aspirin-file-name>_error_string`
: Takes no arguments. Returns a string describing the latest error. Generally called immediately after one of the above signals an error.

[11] See `$NNTOOLS/include/aspirin_backprop.h` for the structure definition and see the source in `$NNTOOLS/src/aspirin/libBp/io` for examples of getting network information.

Chapter 8

MIGRAINES INTERFACE

The MIGRAINES interface was developed in order to allow the neural network researcher to get *inside* the neural network. Rather than treating the neural network as a mystical black box, the MIGRAINES interface allows the researcher to probe the neural network using available analysis tools. Before describing the MIGRAINES interface commands, a few examples are given to illustrate how to visualize the innards of a neural network.

8.1 Visualization Examples

Figure 8.1 uses the apE 2.1 visualization package to illustrate the evolution of the first layer weights as a function of time for a network learning to recognize characters independent of rotation (the network is described in the file `$NNTOOLS/examples/characters/simple/simple.aspirin`).

Similary, figure 8.1 uses Mathematica to display the weights into a node in a network that has learned to recognize butterfly images (see figure 8.1) independent of rotation.

Figure 8.4 uses plots from the **BAYES** example generated by gnuplot (see page 30). The upper plot is a contour plot of the class distributions H1,H2,H3,H4 and the edges of the decision surface learned by the neural network. Notice that despite the large overlap of the classes, the network learned a very good separating surface which runs along the valleys between the classes. The lower plot is another way of looking at the same data, where the class distributions and the neural network decision surface have been superimposed.

Figure 8.5 contains two plots generated from data extracted from Aspirin neural networks. The plot on the top was generated by projecting the values in a hidden layer through the first two principal components (see section 10). The bottom plot graphs the output of a neural network versus the targets in three dimensions.

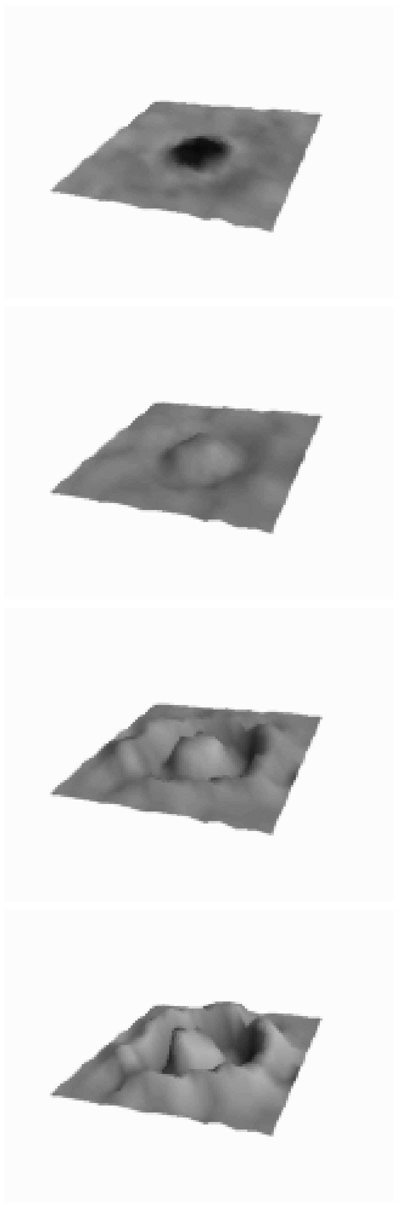


Figure 8.1: Weights from the input to the a node in the hidden layer as a function of time during learning.

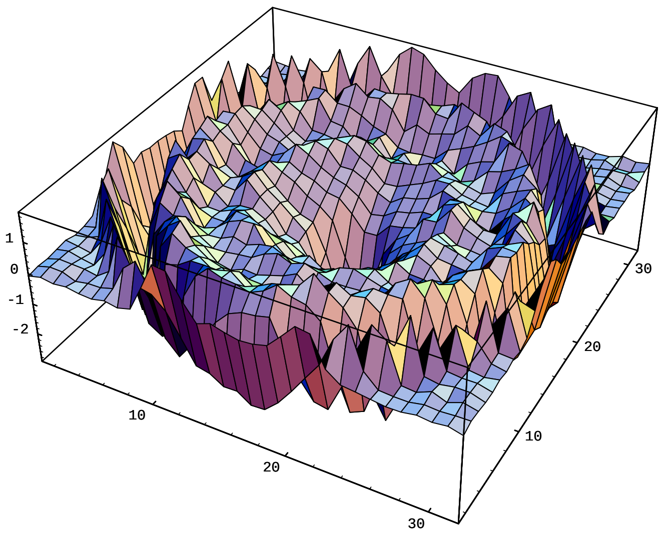


Figure 8.2: Butterfly Weights (from Mathematica)

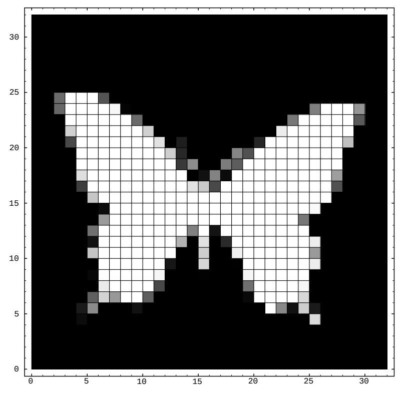
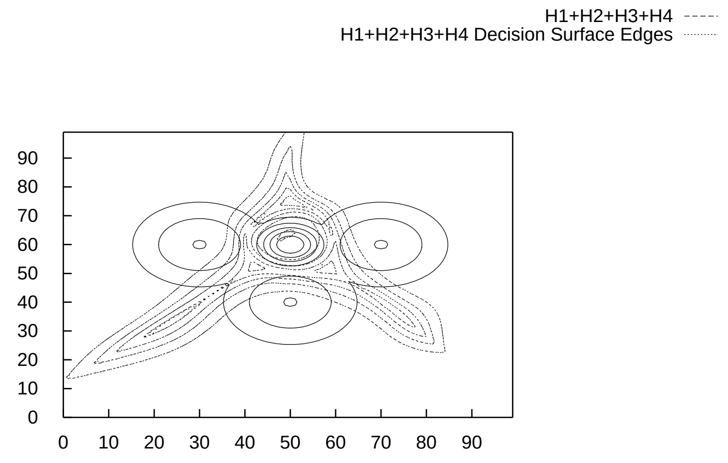


Figure 8.3: Butterfly Inputs (from Mathematica)

8.2 Moving Through the Neural Network (Contexts)

An Aspirin file contains one or more neural network modules called *black box*es. Each black box is composed of a set of component layers, connection vectors between layers, and optionally there may be external inputs and target vectors. The MIGRAINES interface allows you to change the *context* by using the commands `push`, `pop` and `poproot`.

The default context is always the root context. You can `push` into any subcontext by typing `push <subcontext name>`. By typing the `?` command you can display the available subcontexts. From the root context, the subcontexts consist of all of the declared black boxes. If you have used the `-d <datafile>` option on the command line, then there will also be a subcontext called `TestingContext` (see section 8.2.1). The `?` command also displays all relevant commands to the current context. The command `pop` will move you back up to the parent context and the command `poproot` will move you to the root context.



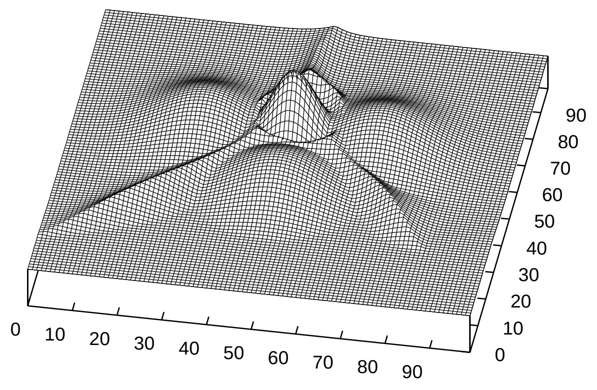
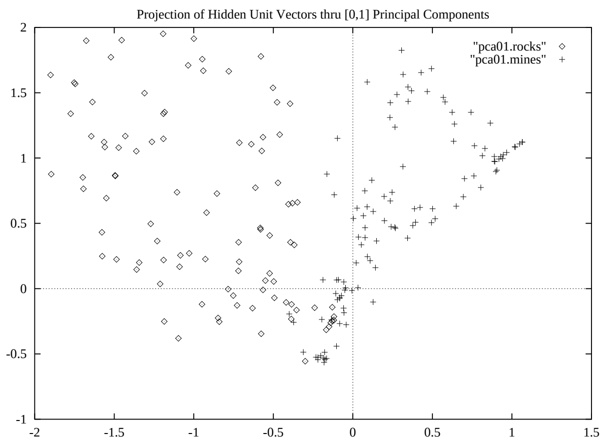


Figure 8.4: Edges of the neural network decision surface and class distributions



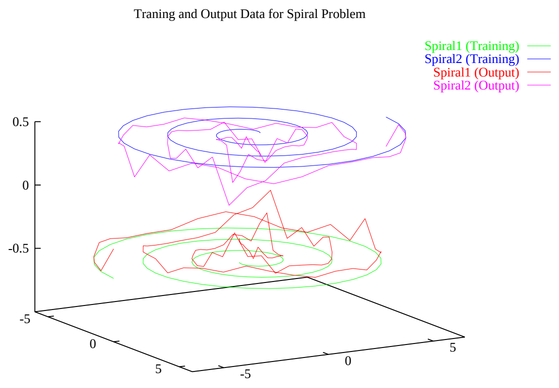


Figure 8.5: Data can be Extracted and Analyzed

8.2.1 Testing Context

This context allows you to do simple testing of networks. The simulation will cycle until an output vector exceeds the error bound. It then prints out which input caused the error and updates any open Unix pipes to node, input or target data.

This will only appear as a subcontext if you have read in data using the `-d` option with a `.df` or have defined a data generator [1]. The commands within this context are:

`Info`
: Print out the current error bound and iteration count.

`SetIterations <iterations>`
: Set the iteration count `<iterations>`.

`SetBound <bound>`
: Set the error bound to `<bound>`.

`StartTesting`
: Cycle through `<iterations>` of data until an input causes the maximum absolute value of the difference between a node value and its target is greater than `<bound>`. Update all open Unix pipes to node, input or target data.

8.3 Global Commands

`cycle <iterations>`
: Present `<iterations>` data patterns to the network. All pipes open to node, target or input data are updated. This will only appear as an option if you have read in data using the `-d` option with a `.df` or have defined a data generator [2].

`pverbose`
: Print messages when dumping data to pipes (default).

`psilent`
: Do not print messages when dumping data to pipes.

`pbinary`
: Use native binary as the data format when piping data.

`pascii`
: Use ASCII as the data format when piping data (default).

[1] See section 9.4.2.
[2] See section 9.4.2.

`pnoheader`
: Use no header when piping data.

`pheader`
: Use a header when piping data (default).

`pinfo`
: Print information about currently open pipes.

`pclose <pipename>`
: Close pipe called `<pipename>`.

`popenWeights <x> <y> <pipename> <command>`
: Only applicable in a connection context. Pipe data in the connection vector that connects to node[`<x>`][`<y>`] into `<command>`.

`popenInputs <pipename> <command>`
: Only applicable in a $INPUTS context. The `cycle` command must have been called at least once to get data into the inputs. Pipe data in $INPUTS to `<command>`.

`popenTargets <pipename> <command>`
: Only applicable in an output layer context. The `cycle` command must have been called at least once to get data into the targets. Pipe data in target vector to `<command>`.

`popenBiases <pipename> <command>`
: Only applicable in a layer context. Pipe data in bias vector to `<command>`.

`popenAr1 <pipename> <command>`
: Only applicable in an autoregressive layer context. Pipe data in ar1 vector to `<command>`.

`popenAr2 <pipename> <command>`
: Only applicable in an autoregressive layer (of order 2) context. Pipe data in ar2 vector to `<command>`.

`popenNodes <pipename> <command>`
: Only applicable in a layer context. Pipe data in node vector to `<command>`.

`load <weightfile>`
: Loads the `<weightfile>`. All open pipes to weight data are updated.

`push <subcontext name>`
: Change contexts to `<context name>`.

`pop`
: Go up a context.

`poproot`
: Go to root context.

`source <filename>`
: Evaluate `<filename>` as if it was typed interactively.

`! <unix command>`
: Execute the `<unix command>` in a subshell. For example, `! ls` will print the contents of the current directory.

`echo <string>`
: The `echo` commands will print all characters following the command to the standard output.

`quit`
: Exit MIGRAINES.

`?`
: Display all commands relevant to this context.

8.4 Pipes

The commands `popenWeights, popenInputs, popenTargets, popenBiases, popenAr1, popenAr2 and popenNodes` allow you to export neural network data to a Unix pipe. These commands only apply in the appropriate context. For example, suppose you had `push`ed into a layer context, then typing `popenNodes Nodes cat > nodes.ascii` would result in the current values of the nodes being written to the program `cat`, which in turn would write the data to the file `nodes.ascii`. Any program that reads data from `stdin` can be used in a popen command. For example, a statistics program could be invoked that collected the output values of the network and reported the performance. Most of the examples contain `.cmd` files that open pipes to extract data from the neural network. Refer to these files for examples of using pipes.

All open pipes should be closed before exiting the MIGRAINES interface. The `pclose <pipename>` command is used to close a pipe.

8.4.1 Data Stream Format

Currently, all data that is written to pipes is floating point data. If the `pbinary` command is executed, then any pipes opened subsequently will use the native single precision (float) floating point format. If the `pascii` command is executed, then any pipes opened subsequently will use ASCII.

An ASCII header is used to describe the data. Its form is:

```
<format> <type> <xdim> <ydim>
```

where `<format>` can be either `ascii` or `binary`, and `<type>` must be `float` and, `<xdim>` `<ydim>` are the x and y dimensions of the data vector, respectively. If the `pnoheader` command is executed, then any pipes opened subsequently will not use a header. If the `pheader` command is executed, then any pipes opened subsequently will use a header.

Unix filters can be written to translate data in this format to formats usable by a plotting/analysis package. Example filters are included with this distribution [3]:

am2gnuplot Convert data to Gnuplot format.

am2math Convert data to Mathematica format.

am2matlab Convert data to Matlab format.

am2ascii Convert data to ASCII matrices (row major).

am2raw Convert data to file of raw single precision floats (strips header).

All of the above filters take an optional `-s` flag that will cause the filter to run in silent mode (no messages). To use these filters you must be using headers on the data streams.

[3]Source located in `$NNTOOLS/src/aspirin/libBp/ui`

Chapter 9

THE `bpmake` UTILITY

`Bpmake` is a compilation tool for automatically creating BP neural network simulations with the `aspirin` neural network code generator. These simulations are automatically linked with the Aspirin simulation library, the MIGRAINES interface library and the data file reading library. `Bpmake` is implemented as a shell script that uses the UNIX `make` utility.

Your environment should have the following variables set before using `bpmake`:

`NNTOOLS`
: set to the path where the Aspirin/Migraines software is installed.

`MACHTYPE`[1]
: set to the kind of machine you are using.

`MANPATH`
: should have `$NNTOOLS/man` appended to it (e.g., if `NNTOOLS=/u1/nntools` then `setenv MANPATH $MANPATH:/u1/nntools/man`).

Your path should include `$NNTOOLS/bin/$MACHTYPE`, where `$MACHTYPE` expands to the type of machine you are using. If you do not know the proper setting for `MACHTYPE`, then look in `$NNTOOLS` to see what the bin directories are called on your system (e.g., `ls $NNTOOLS/bin/*`).

A single Aspirin file should exist in the current directory with a `.aspirin` suffix. By executing `bpmake` the Aspirin software will be invoked using the default simulation code generator (currently backpropagation with full optimization). The resulting C code simulation is then compiled by `bpmake`.

If there is a file `user_init.c` in the directory then this file is also compiled. The `user_init.c` file is the place where the user can alter simulation parameters and preprocess data. This file MUST contain a function called `user_init()` that

[1]Look in `$NNTOOLS/bin` for a list of installed machines at your site.

returns void and takes no arguments. The final executable simulation will call this function before it has initialized the simulation. It is in this function that you can change parameters (e.g., learning rate) and define your own generators for suppling data to the simulation. See section 9.4.1.

Finally, a generic simulation program located in `$NNTOOLS/migraines/bp` called `Backprop.c` will be compiled and linked to the previously compiled files. The executable will be named by using the prefix of the aspirin file (e.g., speech.aspirin => speech).

See the directory `$NNTOOLS/examples` for examples of using `bpmake` with an aspirin file.

9.0.2 Environment Variables and Parameters for bpmake

The Makefile used for the compilation with `bpmake` allows the user to override the internal variables by environment variables or variables supplied on the command line if the environment variable `USE_ENV` has been set to `-e`. For example, the compiler flags can be overridden by setting the `OPTIMIZE` environment variable.

A very important environment variable is `USER_LIBS`. By setting this to a list of objects and/or libraries, the user can link the `user_init.c` file to other functions. For example, if you are reading images using Pixrects in the `user_init.c` file, then you would set the `USER_LIBS` to -lpixrect so the Pixrect routines would be linked.

9.1 Modifying the Base Makefile

The Makefile for `bpmake` is located in `$NNTOOLS/migraines/bp` and is called `Makefile.bp`. This file can be modified for your particular installation. For example, you may need to link to special libraries or might want to alter the compiling instructions to use a multi-pass optimizing compiler.

9.2 The `Backprop.c` File

This file contains the `main()` executable program for simulations built by `bpmake`. It is a very generic application that supports a number of flags (see section 9.5.) This program can be used as an example for writing your own applications that do not use the `bpmake` facility. It is located in the directory `$NNTOOLS/migraines/bp`.

9.3 Input Data

The simulations generated by `bpmake` have the ability to read a number of data file formats. Using data files requires two kinds of files. First, the data needs to be in one of the supported formats (listed below). Second, a *data format* file `(.df)` is needed. A `.df` file controls the reading of the data file. Keywords allow pre-processing options.

The data is supplied to the simulation by presenting one pattern from a data file and then moving to the next data file. Note that backpropagation learning is extremely sensitive to the ordering of these data files.

The small amount of overhead associated with using this method of data input can be eliminated by defining your own data generators, see section 9.4.2.

9.3.1 Data Format Files

A data format file `(.df)` is an ASCII file of C style comments, `#defines`, and `#includes`, `ReadFile`, `UseFile` and `RandomSwitch` commands. The `ReadFile` command will read the specified file. Many data files can be read by using multiple `ReadFile` commands. The `ReadFile` command takes the following form:

```
ReadFile ( <pathname> <type> <options> )
```

Where `<pathname>` is the path name of the data file and `<type>` is its type (see section 9.3.1). These are required arguments to the `ReadFile` command `<options>` field is composed of a list of keywords some of which may be required by the file reader, depending on the data file type. The format for `<options>` is a list of keyword-argument pairs:

```
(keyword-> [<arguments>])
```

The following lists available keywords:

`(Bias-> <real: bias>)`
: Takes a real number as an argument. This will cause the data in each input pattern to have `<real: bias>` added to each element. If `Range->` is being used then the data will be rescaled and shifted to have `<real: bias>` as the center of the range.

```
(Bias-> 2.0)
```

`(Cache-> <integer: npatterns>)`
: Use a data cache of size `<integer: npatterns>` input/target patterns. The data is read from disk as needed into the data cache. This is *very* useful if you have large amounts of data. You declare a file cached and specify how many data patterns are to be in the cache. Instead of reading all of the patterns in the file, the software just fills the cache. When you take a cache "hit" (the next pattern is out of cache), then the cache is filled from the open file. Basically this is virtual memory for your data files. Users that have huge amount of data (much more than physical

memory) will find this useful because it allow networks to run, without stopping, on large data sets and prevents the computer's virtual memory from thrashing. On coprocessor boards (e.g., Mercury i860) this is useful if you have more data than memory. The disadvantages are that cache hits slow you down (but not much if the caches are large) and you are limited by the maximum number of open files on your system.

`(ClearDelays->)`
Takes no arguments. Clear the delay buffers after a pattern has been completely scrolled through. This is usually necessary when using networks that encode state information (e.g., TDNN's[Waibel, 1987]).

`(HighClip-> <real: clip>)`
If an input data point is $>$ `<real: clip>` then set it to `<real: clip>`.

`(LowClip-> <real: clip>)`
If an input data point is $<$ `<real: clip>` then set it to `<real: clip>`.

`(Math-> <string: function> )`
Takes a single argument that specifies a function to be applied to each data point in each input pattern after the data is read. Possible functions are:

- abs[x]
- square[x]
- log10[x]
- log10[x+1]

Only a single `Math->` statement may be used per `ReadFile`. Usage: `(Math-> log10[x+1])`.

`(Normalize->)`
Takes no argument. Causes the each data pattern to have its mean subtracted and its length normalized to 1.0.

`(Range-> <real: range> [<real: threshold>])`
Takes a real number as an argument. This will cause all data patterns to be rescaled on a per pattern basis with a range of `<real: range>` with 0.0 at the center. Rescaling to a small range about zero is recommended for data patterns with very large dynamic ranges.

```
(Range-> 2) /* rescale data between -1 and 1 */
```

If the optional argument `<real: threshold>` is specified, then if the maximum value in a pattern is less than `<real: threshold>` then `<real: threshold>` is used as the maximum for the rescaling. This threshold can prevent the rescaling of noise.

`(Scale-> <real: scalar>)`
: Takes a real number as an argument. This will cause the data in each pattern to have each element scaled by `<real: scalar>`.

`(SubMean->)`
: Takes no argument. Causes each data pattern data to have its mean subtracted.

`(SwapBytes->)`
: Takes no argument. Causes the data to have the byte order reversed. Used when reading data from a machine with opposite byte order.

`(Switch-> [<integer: npatterns>|string: All])`
: Takes an optional integer as an argument. Recall that, the normal mode of presentation is to input one pattern to the network, then to go to the next data file's pattern. The `Switch->` allows you to control when the input to the network switches from one file to the next. The argument corresponds to the number of presentations of data that are to be done before switching to the next file. `<integer: npatterns>` defaults to 1. To delay switching for the whole file use the symbol `All` as the argument.

```
(Switch-> 10) /* switch to next file after 10 presentations
                              */
```

or

```
(Switch-> All) /* switch to next file after the whole file
                              */
```

`(Target-> <integer-list: list of numbers>|string: Self|<string: filename>)`
: Takes a list of ASCII numbers, `Self`, or `<string: filename>` as an argument. These numbers form the target output for all black boxes for this file.

```
(Target-> 1.0 1.0 0.0)
```

If `Self` is used, then the target is the input data pattern itself. This is used for networks learning the identity mapping.

```
(Target-> Self)
```

The ASCII vector and `Self` are required (and only applicable) for Type 1 and ProMatlab format files. The `<string: filename>` argument is only used for Type5 files.

The order of application of pre-processing is:

1. `SwapBytes->`
2. `Math->`
3. `ClipLow->`
4. `ClipHigh->`
5. `SubMean->`
6. `Normalize->`
7. `Range->`
8. `Scale->`
9. `Bias->`

Data Files

The following lists the currently supported formats for data files:

`ASCII Data Format` **(Ascii)**
The format of an **Ascii** file is:

pattern
target pattern
pattern
target pattern
...

A *pattern* or *target pattern* is composed of ASCII numbers with white spaces, carriage returns, tabs or C-style comments between them. Data is stored in row major order[2] for two-dimensional arrays.

`Type 1 Data Format` **(Type1)**
The format of a **Type1** file is:

pattern
pattern
...

A *pattern* is composed of *binary* single precision floating point numbers. Data is stored in row major order for two-dimensional arrays. The target pattern must be declared in the `.df` file using the `Target->` keyword.

[2]Two dimensional data is stored: *row,row,row...*

`Type 2 Data Format` **(Type2)**

The format of a **Type2** file is:

target pattern
pattern
pattern
...

A *pattern* or *target pattern* is composed of *binary* single precision floating point numbers. Data is stored in row major order for two-dimensional arrays. The *target pattern* is the first pattern in the file and applies to all patterns in the file.

`Type 3 Data Format` **(Type3)**

The format of a **Type3** file is:

pattern
target pattern
pattern
target pattern
...

A *pattern* or *target pattern* is composed of *binary* single precision floating point numbers. Data is stored row major for two dimensional arrays.

`Type 4 Data Format` **(Type4)**

The format of a **Type4** file is:

pattern
pattern
...
target pattern
target pattern
...

A *pattern* or *target pattern* is composed of *binary* single precision floating point numbers. Data is stored row major for two dimensional arrays.

`Type 5 Data Format` **(Type5)**

The format of a **Type5** file is:

pattern
pattern
...

A *pattern* is composed of *binary* single precision floating point numbers. Data is stored row major for two dimensional arrays. The target patterns are read from a separate file of the form:

target pattern
target pattern
...

The target pattern file can be declared in the `.df` file using the `Target->` keyword (e.g., `(Target-> data.targets)`). If the `Target->` keyword is *not* used then the software assumes that the targets are contained in a file of the same name as the data but with the word `.targets` appended.

`Matlab Format` **(Matlab)**
Files that are the output of a Matlab program . The target pattern must be declared in the `.df` file using the `Target->` keyword (in the same way as Type1 files). Only one input pattern per file.

The order of the `ReadFile->` statements implies the order of presentation to the neural network. Note that the number of examples of each class in a classification problem (priors) can influence performance. Sometimes it is desirable to reference a file more than once in order to normalize the priors or affect the ordering. Multiple `ReadFile` statements could be used but this would cause the data to read into memory multiple times. Instead, the command `UseFile` is used to reference a file already read into the system. The format for the `UseFile` command is:

```
UseFile(<pathname>)
```

By using the `RandomSwitch` command in a `.df` file you can cause the selection of the next data file to be pseudo-random. The format for the `RandomSwitch` command is:

```
RandomSwitch
```

9.4 Including Application-Specific Code

Users can include their own C code to preprocess the input or to set parameters. This is done by including their own C code in a file called `user_init.c`. The `bpmake` utility will automatically compile and link this code into the executable simulation.

9.4.1 The user_init File

In this file there *must* be a function called user_init() which takes no arguments and returns void. This function is called by the simulation generated by bpmake as the first step of initializing the simulation.

In this file you define functions that were used in the .aspirin file for initializing connections or biases (thresholds). This is where you can define the transfer functions for UserNodes.

You can read and process your own data, using the user_init() function. By defining your own generators, you can control the presentation of the data to the network (see section 9.4.2).

The following is an example that sets the learning rate for a simulation called speech:

```
void user_init()
{
    speech_set_learning_rate(0.01);
}/* end user_init */
```

9.4.2 User-Defined Data Generators

You can define your own generators using the define_generator function. A generator is a C function that takes one argument and returns void. It should set all of the inputs and target outputs for all black boxes in the simulation. By defining your own generators, you can pre-process data as it is presented to the network (e.g., adding noise with a particular distribution).

The define_generator function takes three arguments. The first is the name of the generator function. The second is a pointer to a data structure that you would like passed to the generator when the simulation calls the generator (this can be a NULL pointer). The third argument is a string that you want to have associated with the generator.

See Appendix 6 for a description of the example networks that come with the Aspirin/MIGRAINES software. Many of these examples declare generators in user_init.c files.

9.5 EXECUTING bpmake-CREATED SIMULATIONS

The simulation created by bpmake can be run in a number of different modes and controlled by a number of command-line arguments. Assuming a network is defined in an Aspirin file named <network-name>.aspirin, the executable simulation code will be placed in the file <network-name> by bpmake. The simulation can be run using a command of the form

```
<network-name> [options] [dump file]
```

where the `dump file` is the path name of a file that contains the network state saved in a previous run and `options` may be selected from the following list: [3]

```
[-d][-datafile <datafile>] read datafile (a .df file)
[-a][-alpha <learning rate>] set learning rate
[-i][-inertia <inertia>] set inertia
[-F][-Filename <dump file name>] ("Network" default)
[-l][-learn] learn without graphics
[-s][-save <iterations>] save to "Network.save" every <iterations> (5000 default)
[-#] Append the current iteration number to the save file name
[-t][-test <iterations> <passes> <bound>] test for convergence
 every <iterations> (5000 default) by going <passes>(default 100)
 through generators without an error exceeding +/- <bound>(default 0.1)
[-N][-Notest ] never test for convergence
[-I][-Iterations <max_iterations>] exit after max_iterations (default is unlimited)
[-n <mean> <variance>] add normally distributed noise to inputs(very slow)
[-u <mean> <variance>] add uniformly distributed noise to inputs(slow)
[-f][-forward <iterations>] go forward <iterations> (used for stats and benchmarking)
[-E][-Epoch] go forward one epoch (1 pass thru all data)
(does not apply to user-defined generators!)
[-p][-print] print outputs and targets (used with -f)
[-P][-Pdpfa <threshold>] Calculate Pd and Pfa for -f <iterations>
 using <threshold> for detection threshold (L2 norm)
[-A][-AsciiDump] print out all the weights and thresholds
[-AsciiDumpNoFmt] print out all the weights and thresholds (no formatting, use with -L)
[-L][-LoadAscii] read from stdin the results of -AsciiDumpNoFmt
[-h][-help] this message
```

If neither the `-l` or `-f` option is used, then the simulation will use the MIGRAINES interface. The only flags that have an effect when using MIGRAINES are: `-d`, `-n`, and `-u`. The other flags control the simulation when not using MIGRAINES.

9.5.1 -[d]datafile `<.df filename>`

This allows you to specify a data format file (see section 9.3). If there is no `.df` file, then you must supply a generator via the `user_init.c` file.

9.5.2 -a[alpha] `<learning_rate>`

Override the learning rate with `<learning_rate>`.

9.5.3 -i[inertia]`<inertia>`

Override the inertia with `<inertia>`.

[3]Obtained by using the `-help` option

9.5.4 -l[learn]

Learn.

9.5.5 -F[Filename] `<filename>`

Overrides the default prefix ("Network") to the dump file names.

9.5.6 -s[save] `<iterations>`

The number of iterations between saving the state to `Network.save`.

9.5.7 -#

If this flag is used then the iteration number is appended to the save file name. This is useful if you want to evaluate the performance of the network as it has learned. Be careful, if your simulation runs for a long time you might fill up your disk!

9.5.8 -t[test] `<iterations>` `<passes>` `<bound>`

This flag controls the testing for convergence. The way this works is that every `<iterations>` the simulation stops and propagates forward `<passes>` or until the test fails using `<bound>`. The test is by default the maximum absolute difference. For example, using `-t 1000 500 0.1` will cause the simulation to stop every 1000 iterations and test for convergence. The test runs through 500 input/output pairs. If the maximum absolute error for all output nodes is less than 0.1 for all patterns then the network is said to have converged. Upon successful completion the state of the network is written to a file named `Network.Finished` and the simulation exits.

9.5.9 -N[Notest]

Never test for convergence.

9.5.10 -I[Iterations] `<iterations>`

Exit learning after `<iterations>`(default is unlimited). The state of the network is written to a file named `Network.save`.

9.5.11 -n `<mean>` `<variance>`

Add normally distributed noise to inputs (very slow).

9.5.12 -u `<mean>` `<variance>`

Add uniformly distributed noise to inputs (slow).

9.5.13 -f[forward] `<iterations>`

Go forward `<iterations>` (used for statistics gathering and benchmarking).

9.5.14 -E[Epoch]

Go forward through all of the data patterns in all the data files (i.e.,one epoch). This only applies to data files, *not* to user-defined generators.

9.5.15 -p[print]

Print outputs and targets (used with -f).

9.5.16 -P[Pdpfa] `<threshold>`

Calculate an estimate of the probability of detection (Pd) and probability of false alarm (Pfa) for -f `<iterations>` using `<threshold>` for detection threshold (l_2 norm). The program counts the number of times that the Euclidean distance between the output vector and the target vector is less than `<threshold>`. The proper way to use this option is to run all of the classes of data *separately.* The output is a listing of the max, min, mean and variance for all outputs as well as the Pd and Pfa ratios.

9.5.17 -A[AsciiDump]

Print out the state of the network to the screen. It is best to redirect this output to a file!

9.5.18 -AsciiDumpNoFmt

Print out the state of the network to the screen (unreadable). It is best to redirect this output to a file! This option is useful if you need to move a network between machines with incompatible word formats. The results of -AsciiDumpNoFmt can be read into the simulation by -LoadAscii.

9.5.19 -L[LoadAscii]

Read from stdin the results of -AsciiDumpNoFmt and produce a dump file on the current platform. For example, dump the weights to an ASCII file on a Cray and read them in on a Sun:

```
cray% speech Network.Finished -AsciiDumpNoFmt > weights.ascii
< ftp weights.ascii to a Sun and convert these weights to a local dump file>
sun% speech -LoadAscii < weights.ascii
```

Moving weights between machines with different word sizes or different processors is not recommended. Due to the nonlinearities within the network and the subtle differences in math libraries, rounding rules and floating point processors, a network with the same weights on two different kind of machines is *not* necessarily going to behave the same.

9.5.20 -h[help]

In case you forget.

Chapter 10

THE `analyze` UTILITY

An integrated utility is provided called `analyze` which is a program inspired by the Dennis and Phillips tools [Denis, 1991]. `Analyze` is used to understand the training data as well as how the hidden layers separate the data, in order to optimize the network architecture. The `analyze` program does principal components analysis (PCA), canonical discriminant analysis (Fisher space analysis)(CDA), projections, and histograms. It uses the same data file formats as are supported by `bpmake` simulations and the output data can be translated to a variety of formats. Associated with this utility are shell scripts that implement data reduction and feature extraction.

The techniques of PCA and CDA are well established methods of data analysis. Rather than delve into a detailed discussion of the techniques, the reader is referred to[Duda, 1973, Denis, 1991]. The numerical techniques used in this implementation were modified versions from Numerical Recipes[Press, 1986].

Once you have analyzed the data by PCA or CDA you should consider the following:

- If the projection distributions and histograms indicate that there is separation, then the projection vectors (principal components) might make good feature detectors, or good initial weight values for neural network nodes connecting to the input of the network.

- If you did PCA and there were many small eigen values (principal values) then your data has much redundancy and you should use the projected data as input to a classifier because the dimensionality has been reduced.

To translate a projection file produced from `analyze` into a Type1 file recognized by `bpmake` generated neural network simulations use `am2raw` which strips the header off the file leaving a flat file of single precision floats. For example: `am2raw < mydata.prj > mydata.type1`

10.1 Sonar Example

The Learn script in the sonar example [1] illustrates the use of the analyze utility to analyze the hidden unit values of the neural network. After learning, the hidden unit values are collected into files using the MIGRAINES interface and processed by PCA and CDA. The CDA analysis produces projection vectors (eigen vectors) that best separate the classes.

In fact, the first canonical variate (first eigen vector) can be used as a linear discriminate to separate the classes. This indicates that only one more layer is necessary to separate the classes. If the network had any more hidden layers after the first hidden layer, we would know by this analysis that they are unnecessary. Further, examination of the projection vector indicates that not all of the hidden units are necessary to separate the classes. The near-zero elements of the projection vector do not contribute very much to the classification decision. The nodes that correspond to the near-zero elements can be eliminated without loss of performance for a classifier based on the first hidden layer and the CDA derived linear discriminate. This also indicates that the neural network could be retrained with a smaller number of hidden units.

10.2 Characters Example and DataReduce

The Learn script in the characters example [2] illustrates the use of the analyze utility to analyze the training data. The DataReduce script is used to run PCA and CDA on the training data to reduce the dimensionality of the data. This is frequently a useful step of preprocessing because, if the dimensionality can be reduced, then the training time will be reduced and robustness increased.

The DataReduce script [3] Uses the analyze utility as well as gnuplot to analyze data stored in .df files. DataReduce is an interactive program that requires input from the user, however in the characters example these answers are provide by program so that the Learn script can be run without user input. If the reader is interested in repeating the analysis interactively, the responses are contained in the files YourPcaResponses and YourCdaResponses.

When the Learn script is run two networks are trained and some weight vectors are displayed. Then PCA is performed by DataReduce followed by CDA. Plots are displayed of the projection of the data through the first two and three principal components of the PCA and CDA analysis. The eigen values in the PCA and CDA analyses indicate that the data could be reduced considerably without loss of much information. Further, the CDA analysis indicates that a neural network is not required. Projection through the first two canonical variates combined with some threshold logic can separate the four classes. This

[1] Located in $NNTOOLS/examples/sonar.
[2] Located in $NNTOOLS/examples/characters.
[3] Located in $NNTOOLS/bin/$MACHTYPE.

is illustrated best by the histogram plots. Two thresholds can be drawn to separate the B and the D classes using the first canonical variate. If the value of the projection through the first canonical variate falls between these two thresholds then it is either the A or the C class, so single threshold on the projection through the second canonical variate can separate these classes. One should note the similarity of the plots of the canonical variates and the weight vectors of the neural network.

10.3 Usage

The `analyze` program reads .df files and allows different analysis methods to be applied to the data.

Required arguments:

`-d <df file>`

`-InputsXdim <xdim>`

`-InputsYdim <ydim>`

`-TargetsXdim <xdim>`

`-TargetsYdim <ydim>`

These arguments describe where to get the data and the dimensionality of the data.

Analysis methods:

`-pca`
: Options:

  `-P <prefix>`
  : Use `<prefix>` for prefix of generated files (default: analyze)

  `-range <n>`
  : Save 1-n components

  `-group`
  : Divide data into sets, run pca separately

    `-edit`
    : Ask user to confirm the use of every group

`-unique`
All data with unique targets are grouped (default)

`-max`
All data with the same index of max are grouped

`-cda`
Options:

- `-P <prefix>`
  Use `<prefix>` for prefix of generated files (default: analyze)
- `-range <n>`
  Save 1-n components
- `-group`
  Divide data into sets, run pca separately
  - `-edit`
    Ask user to confirm the use of every group
  - `-unique`
    All data with unique targets are grouped (default)
  - `-max`
    All data with the same index of max are grouped

`-project`
Project the data thru 1-range eigen vectors (looks for files from `<prefix>.<component>.<suffix>`).
Options:

- `-P <prefix>`
  Use `<prefix>` for prefix of generated files (default: analyze)
- `-S <suffix>`
  Use `<suffix>` for suffix of generated files (default: pcs)
- `-range <n>`
  Read 1-n components
- `-autohistogram <n>`
  Compute an n bin histogram autoscale to max/min
- `-histogram <n> <min> <max>`
  Compute an n bin histogram in the interval [`<max>`,`<min>`]

`-div`
This is used to divide data read from the .df file into groups. Options:

- `-P <prefix>`
  Use `<prefix>` for prefix of generated files (default: analyze)
- `-edit`
  Ask user to confirm the use of every group

`-unique`
: All data with unique targets are grouped (default)

`-max`
: All data with the same index of max are grouped

Appendix A

INSTALLING A/M V7.0

As of this writing, the following UNIX systems are supported:

1. Convex
2. Cray
3. DecStation
4. IBM RS/6000
5. Intel 486/386 (Unix System V, Linux)
6. HP 9000
7. NeXT
8. News
9. Silicon Graphics
10. Sun

Installation begins by loading the compressed tarfile containing the Aspirin/MIGRAINES into a newly created `nntools` directory using the UNIX utility `tar`. The shell script `install.script` is then invoked to do the rest. The following text describes this procedure in greater detail.

1. Create a directory to hold the Aspirin/MIGRAINES software. For example, create the directory `/usr/nntools`. To accomplish this task type the following UNIX commands:

   ```
   cd /usr
   mkdir nntools
   ```

2. Uncompress the distribution by typing: `uncompress am7.tar.Z` which will produce the file `am7.tar` .

3. Use the UNIX utility `tar` to read the tarfile into this newly created directory.

```
cd nntools
tar xvf am7.tar
```

4. Set the environment variables described in Appendix B. These variables must be in effect before beginning the next step.

5. Fix the known bugs (see Appendix C).

6. Read the file README.install and follow instructions.

This should result in the recompilation of all of the subdirectories.

Please notify Russell Leighton (russ@elegant-software.com) of any changes you make to the code so we may consider incorporating them in future releases. Also, any questions, comments, or bug reports should be directed to Russell Leighton. Changes, additions and bug fixes are welcome at any time.

Appendix B

YOUR UNIX ENVIRONMENT

Certain conventions have been established to facilitate Aspirin/MIGRAINES programming. For Aspirin/MIGRAINES software execution you will need to declare the UNIX environment variables required by the software. If you have installed Aspirin/MIGRAINES in the directory `/usr/nntools` you will need to include the statements:

```
setenv NNTOOLS  /usr/nntools
setenv MACHTYPE sun_sparc
setenv MANPATH /usr/nntools/man:$MANPATH
```

in either your `.cshrc` or your `.login` file (assumming csh).
Shared scripts need to in your path:

```
set path = ($path $NNTOOLS/share)
```

Finally, the executable images are located in `$NNTOOLS/bin/sun_sparc` (if you are running on a sun w/sparc). More generally, the binary files are in `$NNTOOLS/bin/$MACHTYPE`, where `$MACHTYPE` expands into the kind of machine you are running on. Your `.cshrc` should include the following line after the rest of your path descriptions:

```
set path = ($path $NNTOOLS/bin/$MACHTYPE)
```

If you have a coprocessor (e.g., mc_i860) then you should put the bin area for the coprocessor *after* the host:

```
set path = ($path  $NNTOOLS/bin/sgi_r3k $NNTOOLS/bin/mc_i860)
```

Appendix C

RELEASE NOTES, KNOWN BUGS AND WORK-AROUNDS

There seems to be a problem with convergence testing when am7 is compiled with Sun's ANSI C compiler. This has not yet been fixed. I suspect it has something to do with passing function pointers and prototyping. To work around this, get the `gcc` compiler [1] (see `$NNTOOLS/README.ansiC`).

Some people have been having problems with header files and `gcc`. This is a `gcc` installation problem. The probelm seems to be that `cpp` is getting the `/usr/include` files *not* the `gcc` ANSI files. Proper installation of `gcc` will prevent this.

[1]The `gcc` compiler an excellent ANSI C compiler that works across many platforms and can be gotten free via anonymous ftp or on a CD from the Free Software Foundation.

Appendix D

PORTING TO NEW PLATFORMS

Follow these steps to configure V7.0 for a new Unix platform:

1. Name your system. The naming convention is <**platform**>_<**processor**>[_<**os**>]. The _<**os**> is optional, since Unix is assumed. Set the environment variable MACHTYPE to this name. For example:

   ```
   setenv MACHTYPE ds_alpha
   ```

2. Find the *closest* system in the `./config` directory.

3. Copy the *closest* system file to `$MACHTYPE`.config.

4. Change the compiler flags, libraries etc. for your system.

5. Check the file `.src/os/header.h` to see if the defaults for `#includeing` headers and math functions are correct. This file always seems to need to be modified. If you can avoid modifying this file, please do. Many of the macros can be overridden by defining a file `$MACHTYPE.h` in the `$NNTOOLS/config` directory and adding `-DAM_MACHTYPE_HEADER="\"$NNTOOLS/config/$MACHTYPE.h\"'` to the `CFLAGS`. This causes this header to be `#included` in the master header file and can override macros. Some of the system configurations use this option. See these as examples.

   Check the C files in `.src/os` to make sure your OS supports all of the system level functions required. For example, many coprocessors do not allow the use of `system()`. There is a conditional compile in `am_system()` that causes an error statement printed instead of the system call. If your system does not support a function used and a conditional compile already exists, then define the appropriate constant in your file `$NNTOOLS/config/$MACHTYPE.h` file. If there is no conditional compile, then add one and update your `$MACHTYPE.h` file.

If you are having problems with underflows you should modify the file `$NNTOOLS/src/os/am_traps.c` to call a floating point hardware configuration function to force underflows to zero without trapping. Underflows *must* go to zero.

There has been an attempt to put all system dependent parameters in the files `$NNTOOLS/config/$MACHTYPE.config`, `$NNTOOLS/config/$MACHTYPE.h`, and `$NNTOOLS/src/os/*.c`. You should only need to change the `$MACHTYPE.config` and the `$MACHTYPE.h` files. In rare cases, you might need to change the `$NNTOOLS/src/os/*.c` files.

6. Type: `$NNTOOLS/Makify` .

   In each directory to be compiled there is a basic Makefile (`Makefile.am`) and optionally a file with specific macro definitions (`local.am`) for that directory. The script `$NNTOOLS/Makify` creates a makefile for your platform by concatenating the files `local.am`, `$NNTOOLS/config/$MACHTYPE.config`, and `Makefile.am` into a file in the directory called `$MACHTYPE.mk`. In this way, all of the machine specific configuration information is included into the makefile.

7. Type: `make -f $MACHTYPE.mk install`

   This compiles and then copies the executable programs (aspirin, etc) into `$NNTOOLS/bin/$MACHTYPE` and the libraries into `$NNTOOLS/lib/$MACHTYPE`.

8. Goto `./examples` and compile *all* examples. Type: `make -f $MACHTYPE.mk` Test all of the examples.

9. If you have problems, go back to 2-4 and then re-make.

10. Update the file `./config/Platforms` with the new machine. This is done for future installations.

11. Send a copy of `$MACHTYPE.config` to **russ@elegant-software.com** for inclusion in future distributions.

Try running the script ByteOrder in the `$NNTOOLS/porting` directory if the byte order is unknown. This is only relevant for the characters example, see the `.df` file.

If you have a vector machine (e.g., convex, cray, i860 machines), you need to link in the matrix routines if you want this software to run fast. Do the above *vanilla* port, and test it before doing the vector port.

Go to the directory `./src/aspirin/libBp/sim` .

In the .c files you will see conditional compiles to use vector routines instead of *vanilla* C. Modify these files for your machine and vector libraries. Change the `$MACHTYPE.config` to be like the cray or meikoi860 files. The use of vector libs can be turned on and off by selectively commenting the `SLIBS` and the `VECTORLIBS` variables:

```
# Added in Classpack from Kuck&Associates (BLAS)
VECTORLIBS=-DBLAS
SLIBS = -lm -L$$MEIKOHOME/860/hlib -lkmath

# Plain C (if you don't have BLAS)
#SLIBS =  -lm
```

Appendix E

USING COPROCESSORS

Generally, one can obtain a considerable performance increase by using a coprocessor that is specially tailored to floating point vector operations. As of this writing there are many vendors selling add-in boards for workstations using Intel's i860 microprocessor. Given that the vendor supplies a good vector library, Aspirin generated simulations can be expected to run extremely efficiently. For example, the Mercury i860 coprocessor will run the nettalk network (`$NNTOOLS/examples/ntalk`) at more than 12.4 million connections per second (25+ MFLOPS) without learning [1].

If you are using a coprocessor then follow the instructions in Appendix B for setting up your environment and search path.

To compile a simulation for a coprocessor first set for environment variable `MACHTYPE` to the name of the coprocessors. For example:

```
setenv MACHTYPE mc_i860
```

for a Mercury i860. If you are unsure of what your coprocessor is called search the directory `$NNTOOLS/bin`. As of this writing, the following coprocessors are supported:

1. Mercury i860 (40MHz)
2. Meiko Computing Surface w/i860 (40MHz) Nodes

Then edit your `.aspirin` file and run `bpmake`. This should result in a simulation that is compiled for the coprocessor.

To run a simulation on a coprocessor you should use the loader program for the product. This is typically done by executing a loader program followed by the normal program and arguments. For example, on the Mercury i860:

```
runmc xor -l -d xor.df -t 50 4 0.1
```

[1]This was fast in 1991, you are probably laughing given todays processors.

on the Meiko i860:

```
mrun xor -l -d xor.df -t 50 4 0.1
```

Some coprocessors may expect that the program and its arguments be compiled into an executable, so one more step may be required before running the program.

Some coprocessors do not allow some system calls back to the host (e.g., Mercury i860). This implies that `cpp` may not be able to be run on your .df files. If this is the case, and you are using macros (i.e., `#define` statements) run `cpp` manually on the .df file to produce a macro expanded .df file. You can then use this new .df file with the simulations.

Most coprocessors will require that you use an optimized vector library to attain peak speeds. See section D.

Appendix F

PARALLEL COMPUTERS

There is a myth that “neural network algorithms” are a parallel computing paradigm. This is only a true statement to the degree that it is true for digital signal processing (DSP) or matrix algorithms.

With respect to backpropagation learning, parallelizing backpropagation is *not* an efficient undertaking except in 2 special cases:

- If you have a *fine* grain parallel/vector machine then parallelize the matrix/vector routines. Aspirin *can* take advantage of this style of parallelism using vector libraries. See section D.
- If you have huge amounts of data, and long update intervals, then distribute the training data across processors and broadcast the weight updates. This version of Aspirin does not support this kind of learning.

Once a network has been trained, running multiple copies of the network on different processors to process different data channels can be a very efficient method of parallelization.

Appendix G

GNUPLOT

The examples use a plotting package called `gnuplot`. In general, `gnuplot` is available as the file gnuplot?.?.tar.Z. Please obtain gnuplot from the site nearest you. Many of the major ftp archives world-wide have already picked up the latest version, so if you found the old version elsewhere, you might check there. In NORTH AMERICA, anonymous ftp to dartmouth.edu (129.170.16.4), fetch `pub/gnuplot/gnuplot?.?.tar.Z`, in binary mode.

Appendix H

Tk/Tcl and wish

In order to run **NNinspect** you need the **wish** shell. Install both Tk and Tcl by anonymous ftp from harbor.ecn.purdue.edu in `pub/src`.

Appendix I

NET TOOLS

We have include a simple set of analysis tools by Simon Dennis and Steven Phillips. They can be used with the examples to illustrate the use of the MI-GRAINES interface with analysis tools, although the `analyze` utility largely replaces these tools. The package contains three tools for network analysis:

- gea - Group Error Analysis
- pca - Principal Components Analysis
- cda - Canonical Discriminants Analysis

Group Error Analysis (gea)

Gea counts errors. It takes an output file and a target file and optionally a groups file. Each line in the output file is an output vector and the lines in the targets file are the corresponding correct values. If all values in the output file are within criterion of those in the target file then the pattern is considered correct. Note that this is a more stringent measure of correctness than the total sum of squares. In particular it requires the outputs to be either high or low rather than taking some average intermediate value. If a groups file is provided then gea will separate the error count into the groups provided.

Principal Components Analysis (pca)

Principal components analysis takes a set of points in a high dimensional space and determines the major components of variation. The principal components are labeled 0-(n-1) where n is the dimensionality of the space (i.e. the number of hidden units). The original points can be projected onto these vectors. The result is a low dimensional plot which has hopefully extracted the important information from the high dimensional space.

Canonical Discriminants Analysis (cda)

Canonical discriminant analysis takes a set of grouped points in a high dimensional space and determines the components such that points within a group form tight clusters. These points are called the canonical variates and are labeled 0-(n-1) where n is the dimensionality of the space (i.e. the number of hidden units). The original points can be projected on to these vectors. The result is a low dimensional plot which has clustered the points belonging to each group.

TECHNICAL REPORT

`$NNTOOLS/doc/NetTools.ps.Z` [Denis, 1991] is a technical report which demonstrates the results which can be obtained from pca and cda. It outlines the advantages of each and points out some interpretive pitfalls which should be avoided.

Appendix J

THE BACKPROPAGATION ALGORITHM

The following is a derivation of the generalized delta rule (backpropagation) for learning in feedforward networks.

The weighted sum input into the jth node from n other nodes is

$$net_j = w_{j0} + \sum_i^n w_{ji} o_i$$

This is passed through a transfer function f_j to produce the output value for the jth node [1]

$$o_j = f_j(net_j)$$

The weight change of a particular weight should be proportional (α) to the contribution of that weight on the total error (E).

$$\Delta w_{ji} = -\alpha \frac{\partial E}{\partial w_{ji}}$$

By the chain rule

$$-\frac{\partial E}{\partial w_{ji}} = -\frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

Let the "credit" for the jth node be $\delta_j = -\frac{\partial E}{\partial net_j}$, then

$$\Delta w_{ji} = \alpha \delta_j o_i$$

[1]A typical transfer function is $f_j(net_j) = \frac{1}{1+e^{-net_j}}, f_j'(net_j) = f_j(net_j)(1 - f_j(net_j))$

By the chain rule

$$\delta_j = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} = -\frac{\partial E}{\partial o_j} f'_j(net_j)$$

If the jth node is an output unit then the error E is defined

$$E = \frac{1}{2}(t_j - o_j)^2$$

where t_j is the teaching signal (target value) for the jth node. Then

$$\frac{\partial E}{\partial o_j} = -(t_j - o_j)$$

Therefore the "credit" for the jth node (an output) is

$$\delta_j = (t_j - o_j) f'_j(net_j)$$

If the jth node is a hidden node then the "credit" is

$$\frac{\partial E}{\partial o_j} = \sum_k^{\kappa} \frac{\partial E}{\partial net_k}\frac{\partial net_k}{\partial o_j} = -\sum_k^{\kappa} \delta_k w_{kj}$$

$$\delta_j = f'_j(net_j) \sum_k^{\kappa} \delta_k w_{kj}$$

where $k - \kappa$ are the indices of nodes *to which* the jth nodes connects.

The following are some extensions that can be added to improve convergence time and avoid local minima.

An *inertia* (also called *momentum*) term can be added to smooth the weight changes over time. The weight change equation is modified to include an inertia constant (γ) in the interval [0,1):

$$\Delta w_{ji}(t) = \alpha \delta_j(t) o_i(t) + \gamma \Delta w_{ji}(t-1)$$

$$w_{ji}(t) = w_{ji}(t-1) + \Delta w_{ji}(t)$$

Scott Fahlman suggests that biasing the derivative speeds convergence [Fahlman, 1988]. This avoids cases where the derivative is equal to exactly zero.

Backpropagation learning is very sensitive to the ordering of the training patterns. This can be used to advantage on difficult problems by using a "shaping" [Wieland, 1988] schedule in the learning. A shaping schedule is an ordering of the data during learning. Initially the network only learns with the subset of the training data that is easily separated. The simulation is stopped, more data is added to the current training data and the simulation resumed. Gradually more data is introduced (thus "shaping" the decision surface) until the whole data set has been learned. This technique is not *necessary* on many problems, however, the authors have experienced many cases where the difference between successfully learning and local minima was the training schedule.

Appendix K

THE AUTOREGRESSIVE BACKPROPAGATION ALGORITHM

In backpropagation learning [Rumelhart, 1986], the inputs to a neuron are multiplied by feedforward weights and summed, along with a node bias term. The sum is then passed through a smooth sigmoidal transfer function, producing the neuron's output value. This neural model has no memory, because the output value is independent of previous inputs or outputs.

The memoryless model described above has been extended to include an autoregressive(AR) memory, $m_i(t)$ (figure 7.4.1), a form of self-feedback where the output depends on the current output plus a weighted sum of previous outputs [Leighton, 1991]. The AR node equations are:

$$\begin{aligned} o_i(t) &= f(net_i(t)) + m_i(t) \\ net_i(t) &= bias_i + \sum_{j=1}^{ninputs_i} w_{j,i} o_j(t) \\ f(x) &= \frac{1}{1+e^{-x}} \\ m_i(t) &= \sum_{n=1}^{order_i} a_{i,n} o_i(t-n), \end{aligned}$$

where $o_i(t)$ is the output of node o_i at time t, $w_{j,i}$ is the feedforward weight from node j to node i [1], $a_{i,n}$ is the feedback weight from the nth delay of node i,

[1] Node i must be contained in a layer farther from the input than the layer containing node j.

ninputs$_i$ is the number of input nodes to the ith node and j is an index through this set of input nodes, and *order*$_i$ is the number of feedback delays for the ith node. The feedback weights ($a_{i,n}$) and the feedforward weights ($w_{j,i}$) are adaptive. The backpropagation learning algorithm can be modified such that each model neuron can change its $w_{j,i}$'s and $a_{i,n}$'s to minimize the output error. In this way, networks can "remember" past events and make context-sensitive decisions.

K.0.1 Gradient Descent

Error backpropagation is a learning algorithm for feedforward neural networks that attempts to minimize the mean squared error on the output by adjusting the feedforward weights. AR backpropagation extends this algorithm to discrete time-varying systems by including feedback weights that are also adjusted to minimize the mean squared error. The output error measure is defined as:

$$\begin{aligned} E &= \sum_{t=T_0}^{T_1} E(t) \\ E(t) &= \frac{1}{2}\sum_i (d_i(t) - o_i(t))^2, \end{aligned}$$

where o_i is the value of the ith output node and d_i is the desired value of the ith output node. The weight changes are accumulated between T_0 and T_1. The weights are then changed and the process is repeated. The interval from T_0 to T_1 is called the *update interval.*

In order to perform gradient descent in weight space, each feedforward weight, $w_{j,i}$, and each feedback weight, $a_{i,n}$, need to change proportionally to the negative of the gradient with respect to that weight:

$$\begin{aligned} \Delta w_{j,i} &= -\alpha \frac{\partial E(t)}{\partial w_{j,i}} \\ \Delta a_{i,n} &= -\alpha \frac{\partial E(t)}{\partial a_{i,n}} \end{aligned}$$

where α is a constant of proportionality.

The partial derivatives with respect to the feedforward weights can be expanded to:

$$-\frac{\partial E(t)}{\partial w_{j,i}} = -\frac{\partial E(t)}{\partial o_i(t)} \frac{\partial o_i(t)}{\partial w_{j,i}}$$

where

$$\frac{\partial o_i(t)}{\partial w_{j,i}} = \frac{\partial f(net_i(t))}{\partial w_{j,i}} + \frac{\partial m_i(t)}{\partial w_{j,i}}$$

$$\begin{aligned} &= \frac{\partial f(net_i(t))}{\partial net_i(t)} \frac{\partial net_i(t)}{\partial w_{j,i}} + \frac{\partial m_i(t)}{\partial w_{j,i}} \\ &= f'(net_i(t)) o_j(t) + \sum_{n=1}^{order_i} a_{i,n} \frac{\partial o_i(t-n)}{\partial w_{j,i}}. \end{aligned}$$

The *credit* for an output node o_i at time t is defined as:

$$\delta_i = -\frac{\partial E(t)}{\partial o_i(t)} = d_i(t) - o_i(t).$$

Brute force calculation of the δ_i for a hidden node is computationally undesirable. However, assuming that the error at the output changes slowly, the recursive calculation used in standard backpropagation may be used. If the error changes slowly, the δ_i for a hidden unit o_i is:

$$\delta_i = -\frac{\partial E(t)}{\partial o_i(t)} \approx -\sum_{\kappa=1}^{noutputs_i} \frac{\partial E(t)}{\partial o_\kappa(t)} \frac{\partial o_\kappa(t)}{\partial o_i(t)} = \sum_{\kappa=1}^{noutputs_i} \delta_k \frac{\partial o_\kappa(t)}{\partial o_i(t)}.$$

The index of a node receiving input from o_i is represented as κ. The partial derivative, $\frac{\partial E(t)}{\partial o_\kappa(t)}$, has been calculated previously. The partial derivative, $\frac{\partial o_\kappa(t)}{\partial o_i(t)}$, is calculated as :

$$\begin{aligned} \frac{\partial o_\kappa(t)}{\partial o_i(t)} &= \frac{\partial f(net_\kappa(t))}{\partial o_i(t)} + \frac{\partial m_\kappa(t)}{\partial o_i(t)} \\ &= f'(net_\kappa(t)) w_{i,\kappa} + \sum_{n=1}^{order_k} a_{k,n} \frac{\partial o_\kappa(t-n)}{\partial o_i(t)}. \end{aligned}$$

Therefore, $w_{j,i}$ is updated to minimize E:

$$\begin{aligned} \Delta w_{j,i}(t) &= \alpha \left(-\frac{\partial E(t)}{\partial o_i(t)} \right) \frac{\partial o_i(t)}{\partial w_{j,i}} = \alpha \delta_i \left[f'(net_i(t)) o_j(t) + \sum_{n=1}^{order_i} a_{i,n} \frac{\partial o_i(t-n)}{\partial w_{j,i}} \right] \\ w_{j,i}^{new} &= w_{j,i}^{old} + \frac{1}{T_1 - T_0} \sum_{t=T_0}^{T_1} \Delta w_{j,i}(t). \end{aligned}$$

The $bias_i$ terms are considered feedforward weights from nodes with values of 1.0 and updated by the above equations.

The partial derivative, $-\frac{\partial E(t)}{\partial a_{i,n}}$, can be expanded to:

$$-\frac{\partial E(t)}{\partial a_{i,n}} = -\frac{\partial E(t)}{\partial o_i(t)} \frac{\partial o_i(t)}{\partial a_{i,n}},$$

where

$$\begin{aligned} \frac{\partial o_i(t)}{\partial a_{i,n}} &= \frac{\partial f(net_i(t))}{\partial a_{i,n}} + \frac{\partial m_i(t)}{\partial a_{i,n}} \\ &= o_i(t-n) + a_{i,n} \frac{\partial o_i(t-n)}{\partial a_{i,n}}. \end{aligned}$$

The δ_i for o_i has already been calculated above. Therefore, $a_{i,n}$ is updated to minimize E:

$$\begin{aligned}
\Delta a_{i,n}(t) &= \alpha\left(-\frac{\partial E(t)}{\partial o_i(t)}\right)\frac{\partial o_i(t)}{\partial a_{i,n}} = \alpha\delta_i\left[o_i(t-n) + a_{i,n}\frac{\partial o_i(t-n)}{\partial a_{i,n}}\right] \\
a_{i,n}^{new} &= a_{i,n}^{old} + \frac{1}{T_1 - T_0}\sum_{t=T_0}^{T_1}\Delta a_{i,n}(t).
\end{aligned}$$

K.0.2 Stability Issues

The AR node is asymptotically stable if the poles of the feedback transfer function are located within the unit circle of the Z-plane. The AR1 node has only one pole, which is identical to the feedback weight. Therefore, the stability criterion is:

$$|a_{i,1}| < 1.$$

Feedback weights are initialized to small random values that do not violate the stability criterion. At the end of an update interval, if $a_{i,1}^{new}$ violates the stability criterion, it is recalculated. The change from $a_{i,1}^{old}$ to $a_{i,1}^{new}$ is halved until $a_{i,1}^{new}$ satisfies the stability criterion. From a computational standpoint, it is not desirable to compute the poles for higher-order nodes (i.e., AR2+). Stability checks can be made using only the feedback weights and their respective desired changes. These tests are based upon the Routh-Hurwitz stability criterion [Pandit, 1983]. As in the first-order case, if the stability criterion is violated, the changes in the feedback weights are halved until the resulting weights satisfy the stability criterion.

K.0.3 Weight Update Issues

The evaluation of $\frac{\partial o_i(t)}{\partial w_{j,i}}$ during learning represents the effect of $w_{j,i}$ through past time. Similarly, $\frac{\partial o_i(t)}{\partial a_{i,n}}$ represents the effect of $a_{i,n}$ through past time. At the end of each update interval, the weights are changed. The partial derivatives[2] associated with each adaptable parameter are set to zero to ensure that the new sums reflect only the effect of the current update interval. The network can be tuned to patterns of a particular duration by choosing an update interval roughly equal to the duration of the patterns of interest. In order to improve convergence, the AR backpropagation equations can be modified to include a momentum term [Rumelhart, 1986].

[2]The partial derivatives are implemented as running sums.

Bibliography

[Denis, 1991] Denis, S. and S. Phillips, 1991, *Analysis Tools for Neural Networks*, Technical Report 207, Department of Computer, University of Queensland.

[Duda, 1973] Duda, R. and P. Hart, 1973, *Pattern Classification and Scene Analysis*, Wiley and Sons.

[Fahlman, 1988] Fahlman, S., 1988, "Faster-Learning Variations on Backpropagation: An Empirical Study," In *Proceedings of the 1988 Connectionist Models Summer School.*

[Fisher, 1936] Fisher, R. A., 1936, "The use of mutliple measurements in Taxonomic Problems," *Annals of Eugenics*, Vol. 7, pp. 179–188.

[Gorman, 1988] Gorman, R. P. and T. J. Sejnowski, 1988, "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets," *Neural Networks*, Vol. 1, pp. 75–89.

[Kernighan, 1988] Kernighan, B. and D. Ritchie, 1988, *The C Programming Language*, Prentice-Hall.

[Lang, 1988] Lang, K. and M. Witbrock, 1988, "Learning to Tell Two Spirals Apart," In *Proceedings of the 1988 Connectionist Models Summer School.*

[Leighton, 1991] Leighton, R. and B. Conrath, 1991, "The Autoregressive Backpropagation Algorithm," In *Proceedings of the 1991 International Joint Conference on Neural Networks.*

[Minsky, 1969] Minsky, M. and S. Papert, 1969, *Perceptrons*, MIT Press.

[Pandit, 1983] Pandit, S. and S. Wu, 1983, *Time Series and System Analysis with Applications*, John Wiley and Sons.

[Press, 1986] Press, W. H. et al., 1986, *Numercial Recipes*, Cambridge University Press.

[Rumelhart, 1986] Rumelhart, D., J. McClelland, et al., 1986, *Parallel Distributed Processing*, MIT Press.

[Sejnowski, 1987] Sejnowski, T. J. and C. R. Rosenberg, 1987, "Parallel networks that learn to pronounce English text," *Complex Systems*, Vol. 1, pp. 145–168.

[Thrun, 1991] Thrun, S. B. et al., 1991, *The MONK's Problems - A Performance Comparison of Different Learning Algorithms*, CS-CMU-91-197, Carnegie-Mellon Univeristy.

[Waibel, 1987] Waibel, A., T.Hanazawa, G. Hinton, K. Shikano, and K. Lang, 1987, *Phoneme Recognition Using Time-Delay Neural Networks*, TR-I-0006, ATR Interpreting Telephony Research Laboratories.

[Wieland, 1988] Wieland, A. and R. Leighton, 1988, "Shaping Schedules as a Method of Accelerated Learning," In *Abstracts of the First INNS Meeting*, page 231.

[Wieland, 1988] Wieland, A., R. Leighton, and G. Jacyna, 1988, *An Analysis of Noise Tolerance for a Neural Network Recognition System*, MP-88W00021, The MITRE Corporation, McLean, Virginia.

[Wieland, 1988] Wieland, A., R. Leighton, and W. Morgart, 1988, "Aspirin for MIGRAINES," In *Proceedings of the 1988 International Neural Network Society Conference.*

Index