A Users Guide for The **Xerion** Neural Network Simulator Version 3.1

Drew van Camp
Department of Computer Science
University of Toronto

May 3, 1993

Acknowledgements

Xerion was developed by Drew van Camp, Tony Plate and Geoffrey Hinton and is funded by a strategic grant from the Natural Science and Engineering Research Council of Canada. Evan Steeg contributed the **bm**, **mft**, and **fem** modules. Sue Becker contributed the **hcl**, **scl**, and **kcl** modules. Brion Dolenko contributed the **cascor** module.

Contents

0 Introduction **4**

I Xerion Tutorial **4**

1 Starting the Simulator **4**

2 Network objects **5**

3 The XOR network **5**
3.1 Building the Network 6
3.2 Initializing the Net 7
3.3 Adding Examples 9
3.4 Training the Net 9
3.5 Freezing Weights 10
3.6 Constraining Weights 11
3.7 Saving and Restoring Nets 11

4 Xerion Displays **12**
4.1 Activation Display 12
4.1.1 Updates 15
4.2 Connection Display 15
4.3 Learning Methods 16
4.3.1 Parameters 18
4.4 Group Types 20
4.5 Gaussian Mixture 21
4.6 Graphs 22
4.7 Variables/Objects 23
4.8 Commands for Manipulating Displays 24

II The Xerion Command Interface **25**

5 Command Syntax **25**
5.1 Startup 25
5.2 Comments 25
5.3 Parameter substitution 25
5.4 Command substitution 26
5.5 Quoting 26
5.6 Script files 27

6 Built-in Commands **27**
6.1 Conditionals 27
6.2 Parameter and Variable Manipulation 28
6.3 Traces 29
6.4 Miscellaneous Commands 30

7 Miscellaneous **32**
7.1 Automatically Created Parameters 32
7.2 Blank Interpretation 33
7.3 Output Redirection 33

III Programming in Xerion **34**

8 Writing New Simulators **34**
8.1 Creation and Destruction Hooks 34
8.2 The Main Function 35
8.3 Extension Records 35
8.4 Methods 38

9 Writing New Commands **41**

Appendices **43**

A Commands for Working with Networks **43**
A.1 Commands for creating Networks 43
A.2 Commands for Training Networks 44
A.3 Miscellaneous Commands 45

B Different Types of Units **45**

C A Sample Script File **47**

List of Figures

1 Main Window 12
2 Activation Display 13
3 Display Updates Dialog 15
4 Connection Display 15
5 Learning Methods (Short Display) 17
6 Learning Methods (Long Display) 19
7 Group Types 20
8 Mixture Dialog 21
9 Mixture Display 21
10 Graph Dialog 22
11 Graph 22
12 Variables/Objects 23
13 Main procedure for **Xerion** 35
14 Extension Records for Backprop Simulator 36
15 Create and Destroy Hooks for the Net 37
16 Error Update Method for the Net 38
17 Error and Derivative Update Method for the Net 39
18 Unit Methods 40
19 Template for writing a command 42
20 A script file for showing a unit's incoming links. 48

0 Introduction

Xerion is a collection of C libraries that can be used to implement many different Neural Network paradigms. The libraries contain routines for constructing networks, displaying them, and training them. On its own, **Xerion** is *not* a Neural Network simulator. It *is* a collection of routines that can be used to code complex, and experimental networks simply.

A simulator written using the **Xerion** libraries has access to a command line interface with built in commands for: creating and training networks, examining and modifying data structures, redirecting output using streams, as well as miscellaneous utilities. It also has a graphical interface for displaying network activations on specified examples, and for examining connection weights between units.

Several simulators have been built with **Xerion**. Among them a Boltzmann Machine, Mean Field Theory Machine, Hard and Soft Competitive Learning Networks, a Kohonen Network, Cascade Correlation, Back Propagation and Recurrent Back Propagation Networks. These simulators are available with the standard **Xerion** distribution.

This document is an overview of the **Xerion** Neural Network simulator. It contains a tutorial on how to build a network using the Back Propagation simulator, a reference for the command line interface, and a summary of how to write your own simulator with **Xerion**.

Part I
Xerion Tutorial

This part of the document will describe how to use the Back Propagation module in **Xerion**. It will walk you through starting the simulator, building a network, adding example patterns to the network, and then training on these examples. As well, it will describe the graphical displays that can be used to view the network topology and learning parameters.

1 Starting the Simulator

Before running **Xerion** your shell environment must be configured properly. First set the environment variable `XERIONDIR` to the directory where the **Xerion** configuration files and standard networks have been installed. Then source the standard **Xerion** setup file. If you use the C Shell, do the following:

```
unix> setenv XERIONDIR <directory>
unix> source $XERIONDIR/config/setup.csh
```

If you use the Bourne or Korn Shell, do the following:

```
unix> XERIONDIR=<directory>
unix> export XERIONDIR
unix> .  $XERIONDIR/config/setup.sh
```

You will probably find it convenient to put these commands in your .cshrc (C Shell) or .profile (Bourne and Korn Shell) file.

$XERIONDIR must not contain any of the special characters mentioned in Section 5.5. If it does, the simulator may exit while initializing itself, and print cryptic error messages.
When **Xerion** was installed on your machine, it may have been configured to use the environment variable $BINTYPE. You should check with the person who installed it to see if there are any site-specific setups you must do.

Once your environment is configured, the Back Propagation module can be started from a **UNIX** prompt by typing the command:

```
unix> bp
```

Alternately, it can be run in its own xterm window by using the command:

```
unix> run bp
```

There are man pages for both `run` and `bp`. These can be viewed using **sman** (the **Xerion** equivalent of **man**) at the **UNIX** prompt.

```
unix> sman sman
```

Once the simulator is running, it takes input both from a command line and from a graphical interface. For now we will consider only the command line interface. **Xerion** has commands specifically for building networks as well as others that can be used as standard utilities. All commands have online help that can be obtained by typing `help <command>`.

```
bp-> help help
```

To quit the simulator type either `quit` or `exit`.

2 Network objects

Networks in **Xerion** are composed of sets of objects. Building a network requires creating these objects and connecting them together with the proper structure.

The lowest level objects are **Unit**s and **Link**s. A Unit is an object that has an input, an output and a function for transferring the input to the output. It may also have a target value if it is an output unit. Units are connected to one another by Links. Links have modifiable connection weights, derivatives for the weights, and a function for calculating these derivatives. A Link can connect two, and only two, Units together.

At the next level of objects is the **Group**. A Group is a set of similar Units. All Units in a Group have the same method of activation, and are, in general, of the same "type".

Above a Group is a **Net**. Nets contain a set of Groups and a list of all the Links connecting the Units in the Groups. Nets are the highest level objects in the simulator. Nets, as well as containing Groups, contain objects called **ExampleSets**. ExampleSets are lists of examples (input and target vectors) used to train, or test the network.

3 The XOR network

To demonstrate the creation of a network in **Xerion**, we will build a simple XOR network (a network that learns the exclusive-or problem). It will have a single hidden layer of 2 units. To build this network we will be using many of the commands built into xerion; however, we will by no means use them all. For a complete list of the commands available for building and manipulating networks see the command interface reference in Part II of this document, Appendix A, and the online help once inside **Xerion**.

All the files used to build this network can be found in **$XERIONDIR/nets/bp**. The readpath variable (used by the read command) is automatically initialized to search this directory.

Before building the network you must start the back propagation simulator. Type:

```
unix> bp
```

at the **UNIX** prompt to start it. Once inside the simulator, you can start creating the network.

3.1 Building the Network

The first step in building the network is to create an empty net. Use the command:

```
bp-> addNet "Xor2"
```

Xor2 will now be the name by which we can access the network from the command line.

Names of nets, groups, units, etc. must be quoted if they contain spaces or special characters (see Section 5.5). If they do not contain spaces or special characters, the quotes are optional.

All commands dealing with nets act on a variable called the currentNet. To set the current net use the command[1]:

```
bp-> useNet "Xor2"
```

The next step in building the XOR network is to add groups of units to the network. The syntax for the addGroup command is:

```
addGroup [-type <mask>] <name> <n>
```

The first argument (-type <mask>) is used to tell the simulator what type of units to create. Different masks that can be used are INPUT, OUTPUT, or BIAS. INPUT units have their outputs clamped when an example is presented to the network. OUTPUT units have a target value, also set when an example is presented. BIAS units always have an output of 1 and never change their activation. The -type argument can be omited if the unit is none of these types. For a complete description of the different types of groups you can create, see Appendix B.

When you create a new net, a bias group (called **Bias**) is automatically created and added to it. It has type BIAS and a single unit whose output is always 1. As you add new groups (or units) to the net the bias unit is automatically connected to them.

The final two arguments to addGroup are: the name to give the group, and the number of units to initially put in it. For this network, enter:

```
bp-> addGroup -type INPUT Input 2
bp-> addGroup Hidden 2
bp-> addGroup -type OUTPUT Output 1
```

[1] The command addNet automatically sets currentNet to be the newly created one, so useNet is not really necessary in this case.

The order in which you create the groups is important. A forward pass through the network activates groups in the order they were created. If you want to change this order use the command `orderGroups`.

After the groups have been added to the net you must connect them properly. Remember that the automatically created bias unit was connected to all the groups as they were added to the net.

```
bp-> disconnectGroups Bias Input
bp-> connectGroups Input Hidden
bp-> connectGroups Hidden Output
```

These commands take two arguments: the names of the groups to act on. Since `INPUT` units are always clamped, the bias unit need not be connected to them (disconnecting them is not really essential, but it is good form). The other groups should be fully connected forward.

The two commands `connectGroups Input Hidden` and `connectGroups Hidden Input` are *not* the same. The first command will cause the output activations of the units in `Input` to be passed as the inputs to the units in `Hidden`. The second command will do the reverse.

You have now completely built the XOR network.

3.2 Initializing the Net

Once the network is built you can initialize fields in it. The most important thing to initialize is the weights in the network.

```
bp-> randomize 1.0
```

This will randomize all the weights in the network on the range (-1,1).

Xerion has built in commands that allow you to look at and modify the data structures in the simulator (similar to a debugger). To examine the fields in the current net look at the data object `currentNet` using the `show` command.

```
bp-> show currentNet

   currentNet = {
               numGroups = 4
                    name = "Xor2"
                   error = 0
                    cost = 0
            currentEpoch = 0
               batchSize = 0
              minorWidth = 29
             minorHeight = 29
             minorMargin = 3
              majorWidth = 131
             majorHeight = 67
             majorMargin = 4
           activityWidth = 29
          activityHeight = 29
          activityMargin = 3
```

```
     trainingExampleSet = struct ExampleSetRec 0x100c8828
      testingExampleSet = struct ExampleSetRec 0x100c8888
   validationExampleSet = struct ExampleSetRec 0x100c88e8
                     mz = struct minimizeRec 0x100c8620
             weightCost = 0
              costModel = struct CostModel 0x100c8948
                  group = array [numGroups] of struct 0x100c89c8
              unitGains = 1
             timeSlices = 1
            currentTime = 0
              extension = struct NetExtensionRec 0x10092dac
}
```

You can examine the various pointers and other objects in the network all the way down to individual links.

```
bp-> show currentNet.group[2].unit[0].incomingLink[0]

   currentNet.group[2].unit[0].incomingLink[0] = {
                   name = "Bias -> Hidden.0"
                 weight = -0.898973
            deltaWeight = 0
                  deriv = 0
                preUnit = struct UnitRec 0x100be7c0
               postUnit = struct UnitRec 0x100cdef8
            variableIdx = 0
            scaleFactor = 1
              weightRow = {0 1}
           weightColumn = {0 3}
              extension = struct LinkExtensionRec 0x0000
   }
```

Many of the structures and data fields have online help available with the `whatis` command. If you want to know what a field is used for, or what its default values are, use this command.

```
bp-> whatis zeroErrorRadius

   zeroErrorRadius:
   Interval of acceptance for agreement between desired output and
   actual output of a unit.  The degree to which a "near miss" will
   count as a "hit" (ignored by units that use cross entropy error).
   "zeroErrorRadius" is a field of type "Real" in the structure "NetExtensionRec"
```

To change the values of these (or any other fields in the simulator), use the `set` command.

```
bp-> set currentNet.weightCost = 0.001
bp-> set currentNet.extension.zeroErrorRadius = 0.1
bp-> set currentNet.mz.epsilon = 0.05
bp-> set currentNet.mz.alpha = 0.9
```

This will set the fields in the network to the values specified. For a complete description of all the fields and their meanings, see the man pages for **Net**, **bp**, and **minimize**.

3.3 Adding Examples

Up to this point the net has been created and initialized, but it contains no examples to train or test on. To add examples to the net use the command `addExamples`:

```
bp-> addExamples -type TRAINING xor2.ex
```

This command takes a mask describing the type of examples to add (one of `TRAINING`, `TESTING`, or `VALIDATION`) and the name of a file containing the examples.

The format of the example files is:

```
i i i ...  i,
t t t ...  t;
   ⋮
i i i ...  i,
t t t ...  t;
```

where `i` are the (floating point) input values to be presented to the `INPUT` units, and `t` are the (floating point) target values to be presented to the `OUTPUT` units. Inputs are separated from targets by a comma, and each example is terminated with a semicolon. There must be the same number of `i`'s as `INPUT` units in the net, and `t`'s as `OUTPUT` units. There may be any number of input-target pairs in the example file. The file `xor2.ex` contains:

```
0 0, 0;
0 1, 1;
1 0, 1;
1 1, 0;
```

3.4 Training the Net

Once the example set has been added to the net, it can be trained. There are many commands in **Xerion** for training a network, but all are different versions of one principal command: `minimize`.

`Minimize` is a complex command. It can train a network using: steepest descent, momentum descent, delta-bar-delta, quick-prop, and conjugate gradient with various line searches. By default it performs conjugate gradient. To train using minimize type:

```
bp-> minimize -rep 5 -iter 100
```

This will train the network for a total of 100 iterations (weight updates) using conjugate gradient optimization. Every five iterations (called a repetition) it will print a report on the progress of the training. For a complete description of the `minimize` command, see its man page[2].

All of the other training commands are simple front ends to `minimize`. To learn how to use them, see their online help. The list of commands that can be used to train a network are:

cg - train a network using conjugate gradient descent

checkGrad - check partial derivatives for accuracy.

deltaBarDelta - train a network on the examples using deltaBarDelta.

[2] The online help for the `minimize` command is very sparse, since it is far too complex to describe in a few words. The man page contains everything (and more) that you might want to know about its use.

initMinimize - initialize the minimize record that is used for optimizing the weights.

minimize - train a network on the examples using one of the minimization methods.

momentum - train a network on the examples using momentum descent.

quickProp - train a network on the examples using quickProp descent.

steepest - train a network on the examples using steepest descent.

train - train a network on the examples using momentum descent.

Once a training method has been chosen and used (for example quickProp), the `minimize` command will use that as the default training method instead of conjugate gradient.

Notice that all the `minimize` commands process a *batch* of examples before each weight update. The size of a batch is determined by the field `currentNet.batchSize`. If this value is greater than zero, then each training iteration will process that number of examples from the training set. If it is set to a value less than or equal to zero, then each iteration will do one pass through the example set. To put it more simply: to do online learning, set `currentNet.batchSize` to 1; to do true batch learning, set it to 0; and to do any other size batch, set it to that value.

The conjugate gradient optimization routines work only when you use batch learning. All the others work for batch and online learning.

To test or validate the network there are commands similar to `minimize` called `test` and `validate`. To use them you must add examples to the appropriate example set.

3.5 Freezing Weights

If you don't want training to modify certain weights in the network, you can freeze them using `freezeLink`, `freezeUnit`, or `freezeGroup`.

`freezeLink` freezes the weight of a specific link at some value. Its value will not be changed during network training; however, you can change it using the `set` or `setLink` commands. `freezeUnit` and `freezeGroup` can be used to freeze the weights of all incoming links of the specified units or groups. As an example, freeze all of the weights coming into the first hidden unit using:

```
bp-> freezeUnit "Hidden.0"
```

There are corresponding commands `unfreezeLink`, `unfreezeUnit`, and `unfreezeGroup` for unfreezing the weights.

Names of units are generated automatically from the name of the group containing them. Units in the group `Hidden` would be named `Hidden.0`, `Hidden.1`, …, up to the number of units in the group. Names of links are generated from the names of the two units they connect. The name of the link connecting units `Input.1` and `Hidden.3` would be `Input.1 -> Hidden.3`.

3.6 Constraining Weights

Xerion also has the ability to implement linear weight constraints. Once all the units in a network have been connected, you can constrain any number of the weights to be linear multiples of one another. The commands used to do this are: `constrainLink`, `constrainUnit`, and `constrainGroup`. Each command takes the names of two objects, plus an optional argument specifying the linear constraint between the weights. If no multiple is specified, 1 is assumed.

The command `constrainLink` constrains the weight of one link to be a multiple of another's. The command `constrainUnit` constrains all incoming links of the first unit to be linear multiples of the corresponding incoming links to the second unit. The command `constrainGroup` is equivalent to calling `constrainUnit` for all corresponding pairs of units in the two groups.

> As with the connect commands, order of arguments is important here. Before constraining, any prior constraints on the first argument are broken. The first argument is then constrained with the second. Any constraints on the second argument are retained.

3.7 Saving and Restoring Nets

Once a net has been built and trained, you may wish to save it. The best way to save a net is to build it using a read file that builds the net, initializes its fields, and adds examples. If you do this, then all that you have to save is what is changed during training: the weights. There are commands called `saveWeights` and `loadWeights` which can be used to save and restore the weights of a network.

To save the weights of the XOR network, type:

```
bp-> saveWeights bp.wt
```

To restore the network the next time you start the simulator, read in the supplied file that contains all the commands that build the net (`$XERIONDIR/nets/bp/xor2.in`), and load the weights that you saved:

```
bp-> read xor2.in
bp-> loadWeights bp.wt
```

> Saved weights cannot be loaded into a network that has had its structure modified in any way. Groups, units, and links cannot be added or deleted between saving and loading. Example sets may be changed.

You may also use the command `show -set` to save individual structures. This command will print out an objects name and value in a format that can be read in later to reset the field.

```
bp-> show -set currentNet.extension > foo
bp-> sh cat foo
set currentNet.extension.zeroErrorRadius = 0
```

> Do not use `show -set currentNet`. This will cause *huge* files to be generated, and may recurse infinitely.

4 Xerion Displays

When **Xerion** is started, it pops up a control window containing several push buttons (see Figure 1). Clicking the mouse on any of these buttons will open another window (display) which can then be used to view or modify the network, learning methods, or simulator variables.

Figure 1: Main Window

The seven standard displays controlled by the **Xerion** main window are the:

Activation Display - used to view the activations of the network and process individual examples.

Connection Display - used to view the connection weights of the network.

Learning Methods - used to select the learning method and set parameters that affect it.

Group Types - used to change the activation functions, error measures, and cost models of groups of units.

Gaussian Mixture - used to view a mixture of Gaussians cost model (used in soft weight sharing) as it changes during training.

Graphs - used to plot values as they change through time (usually during training).

Variables/Objects - a list of system variables and objects used by the simulator. The values can be viewed and modified.

The final button in the window (**Quit**) can be used to exit the simulator. You can add new displays to **Xerion**, and some of the modules (e.g. hcl) have done this. To find out more about how to do it try `sman addUserDisplay`.

The appearance and behavior of the displays is highly modifiable via X resources. If you want to change the default appearance (e.g. colors, fonts, labels, sizes) the application-defaults file `$XERIONDIR/config/Xerion` contains many examples to work from. There are also sman pages that list resources for the NetworkWidget (Hinton diagrams) and GraphWidget.

4.1 Activation Display

This display is used to show the network activation for specific examples. It is broken down into several areas (see Figure 2). The first area is a pull-down menu bar across the top. Clicking the mouse on any of the buttons in it will pop up a menu with more items to select from. The **Window** menu contains buttons for refreshing the display, selecting updates (see Section 4.1.1), hiding targets, and, most importantly, closing the display. The **Example Sets** menu allows you to select which example set is displayed in the window. The **Field** menu allows you to select which field of the units is displayed. The **Network** menu allows you to change the size, background, and other miscellaneous parameters of the network display.

The next area of the display is a list of examples on which to train or test the network. The list of examples currently displayed (Training, Testing, or Validation) can be chosen from the **Example Sets** pull-down menu. Clicking the mouse on one of the examples in the list will cause that example to be presented to the network, and the network to be activated using those inputs and targets. The list can be traversed and examples selected either with the mouse, or with the up and down arrow keys (↑ and ↓). Shift ↑ will go to the top of the list and shift ↓ will go to the end of it.

The final area of the display is used to show the network activation for the example you have selected from the example list.

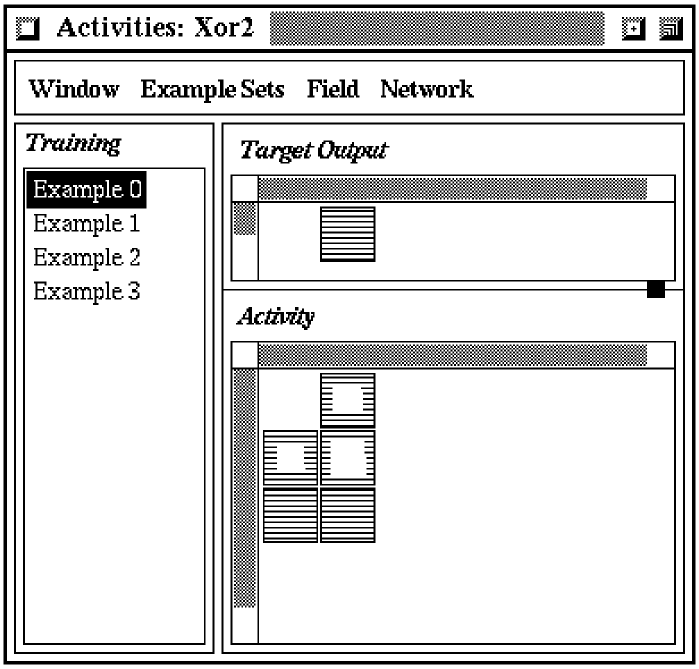

Figure 2: Activation Display

This area uses Hinton diagrams to show both the network activations and the target values for the selected example. In these diagrams each unit is represented by a square with a shaded background. The activation level of the unit is represented by a smaller square inside the larger one. The area of the inside square is proportional to the magnitude of the activation. The larger the inside square, the greater the magnitude of the activation. By default, if the entire square is filled the activation is one. Positive values are represented by white, while negative values are represented by black.

Besides unit activations, you can display other fields from the units. To change the field being displayed, use the **Field** menu.

Clicking the left mouse button on one of the units in the display will cause a short message about the activation to be printed. Clicking the middle mouse button on an input unit will cause its activation to be decremented by a small value, and the network to be reactivated using that input. Clicking the right mouse button will increment the unit's activation. These actions allow you to introduce some noise into your data, and observe how the network performance changes.

Since network activations are all calculated from the input activations, you can only introduce noise to the input units.

To specify the layout of the units in this display, **Xerion** uses a special layout language. To build the layout for the XOR network use the command `doLayout`. This command requires the name of a file containing layout commands.

```
bp-> doLayout xor2.layout
```

The commands for building the activity layout in `xor2.layout` are of the form:

```
activity-layout cell-size <c> margin <m> per-row <n>
<layout commands>
<layout commands>
    ⋮
```

where `<c>`, `<m>`, and `<n>` are integers.

The activity-layout command specifies the size of each unit in the activation display (in pixels), the margin, or space, between the units (in pixels), and the maximum number of units in each row of the display. This command is followed by a series of layout commands. Each line of layout commands lays out one line of units in the activation display. The targets use the same layout as the activations. Valid layout commands are:

`all <group name>`
: Places all of the units in the specified group on a line.

`next <group name> <n>`
: Places the next n units in the specified group on a line.

`section <group name> <m> <n>`
: Places units m through n (inclusive) of the specified group on a line. Units are numbered starting at 0.

`blank <n>`
: Places n blank spaces on a line.

`fill`
: Places enough blank spaces on a line to evenly fill it out to the per-row number.

The activity layout for the Xor2 network is built with the layout commands:

```
activity-layout cell-size 29 margin 3 per-row 2
blank 1, all Output
all Hidden
all Input
```

If the Recurrent Backprop simulator is being used, the activation display is repeated horizontally. There will be one network shown for each time slice in the net.

4.1.1 Updates

The **Updates** option of the **Window** pull-down menu will pop up the dialog shown in Figure 3. This dialog is used to set when the activation display is automatically redrawn (reflecting the current state of the network). In the **bp** network it can be redrawn before minimization starts, after each weight update, after each progress report is printed, or after minimization completes. It can also be redrawn during the `doExamples` command.

You can redraw the display at any number of these points. Simply click the mouse on the toggle buttons for the desired points, then on the **OK** button at the bottom of the dialog. The **Updates** dialog is used by several of the other **Xerion** displays, including the Connection, Graph, and Gaussian Mixture displays.

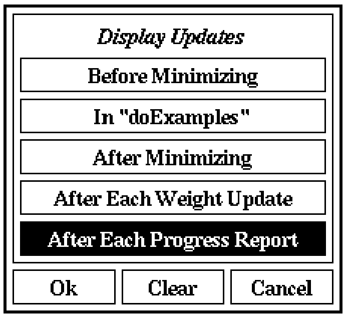

Figure 3: Display Updates Dialog

4.2 Connection Display

This display is used to show the connections in the network. It uses the same Hinton diagrams as the Activation Display to accomplish this. This display can also be broken down into several areas (see Figure 4).

The first area is, again, the pull-down menu pane. The **Window** pull-down menu contains items that refresh the Connection Display, pop-up the **Updates** window, and close the display. The **Field** menu allows you to select what value is shown in the display (again, using Hinton diagrams). You can display one of: **Weight**, the connection weights of the links; **Weight Change**, the most recent change in the weights; or **Derivative** the derivative of the weights with respect to the network error and cost. The **Network** menu allows you to change the size, background pattern and miscellaneous other parameters of the network display.

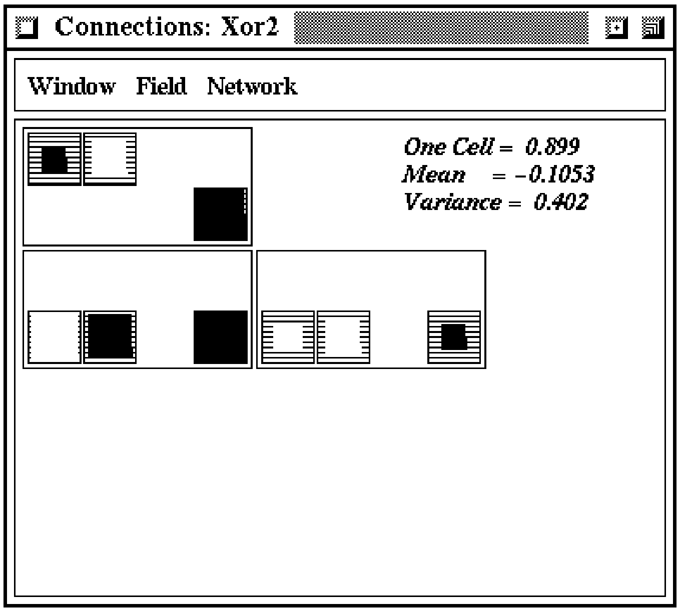

Figure 4: Connection Display

The next area of the display contains the representation of the connection weights. This is a more complex display than in the Activation window. In the top right corner is a line of statistics about the weights (or other values) being displayed. The network display shows, for each unit, the weights of its incoming connections. To do this, the display is broken down into a minor and a major layout. The minor layout specifies how the incoming connections to a unit are to be displayed. The major layout specifies how to lay out each of these minor layouts.

In Figure 4 the major layout is made of the three large boxes. The `major-layout` command specifies where these large boxes are placed. The large box in the top left shows the incoming connections for the single output unit. The two large boxes below it show the incoming connections for the two hidden units. The `minor-layout` command specifies how each of these smaller networks are laid out.

The minor layout is described by the commands:

```
minor-layout cell-size <c> margin <m> per-row <n>
<layout commands>
<layout commands>
   ⋮
```

where `<c>`, `<m>`, and `<n>` are integers. This command follows exactly the format of the `activity-layout` command.

The major layout is specified by:

```
major-layout margin <m> per-row <n>
<layout commands>
<layout commands>
   ⋮
```

where `<m>`, and `<n>` are integers. The layout commands are the same as for both `activity-layout` and `minor-layout`. All of these commands should be placed in the same file and read in by `doLayout`. The complete file for the XOR network is:

```
activity-layout cell-size 29 margin 3 per-row 2
blank 1, all Output
all Hidden
all Input

minor-layout cell-size 29 margin 3 per-row 4
all Hidden, fill
all Input, fill, all Bias

major-layout margin 4 per-row 2
all Output
all Hidden
```

4.3 Learning Methods

This display lets you choose a training method for the network, modify the parameters that affect the method you chose, and then train using it. It is the graphical equivalent of the `minimize` command.

When you first pop-up the **Learning Methods** display, it should look like Figure 5. It has a menu pane across the top, four editable fields below that, and several toggle buttons below that. This is the *condensed* **Learning Methods** display.

The pane across the top of the display contains two pull-down options and three push-buttons. The **Window** pull-down menu allows you to refresh the display, to close the window, and to enlarge the window to the one shown in Figure 6 (see Section 4.3.1). The **Network** pull-down menu allows you to randomize the network weights, test, and validate the network, or modify the frequency with which weights are saved.

The remaining items on the menu pane are push-buttons. To start training the network, click the mouse button on the **Do it!** button. If the minimize routine gets stuck, you can attempt to restart it using the **Reset Search** option.

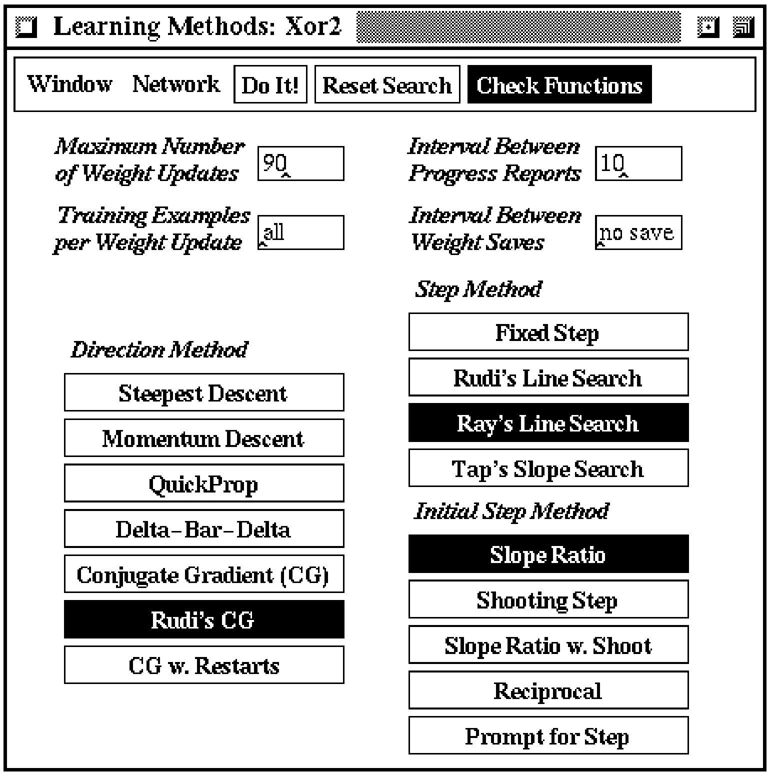

Figure 5: Learning Methods (Short Display)

Before minimizing, the functions used to evaluate the network error and cost are checked for consistency. If you don't want to check them, turn off the **Check Functions** toggle. It is probably a good idea to leave this on until you are certain that your network is training properly (and even after that).

If you change any of the minimization or network parameters, you *must* reset the search before continuing the minimization. If you don't, the search will terminate prematurely.

Below the menu pane are four editable fields that can be used to control the progress of the training. They all have default values, but they are very easy to tailor for a specific network. The fields are:

Maximum Number of Weight Updates is the maximum number of times to change weights during training (beyond the number of times it has already been done). Training terminates once this number of updates have been done.

Training Examples per Weight Update is the number of examples that are processed each time the weights are changed. By default, all the examples in the example set are processed during each weight update; that is, batch training is used. This field can be set to contain any positive integer (set it to 1 to do on-line training).

The conjugate gradient optimization routines (and any of the other routines with a line search) work only if batch learning is being used, i.e. ***Training Examples per Weight Update*** must be set to `all`.

Interval Between Progress Reports determines how frequently to output minimization statistics, and (as a side affect) how frequently to trigger the ***After Each Progress Report*** window update.

Interval Between Weight Saves is how frequently during training, the network weights are saved to a file. The default weight file is `xerion.wt`, but it may be changed via the **Network** pull-down menu. By default, the weights are never saved.

The ***Direction Method*** area contains a list of toggle buttons that can be used to select the training method. The labels on the methods should be self explanatory[3]. The ***Step Method*** area contains a set of toggles that can be used to select the step method to be used for the already chosen learning method. It will default to the best method given your choice of direction method. The ***Initial Step Method*** area can be used to select the type of initial step to take. See the sman page for `minimize` for a description of the learning methods available from this display.

4.3.1 Parameters

The **Show Parameters** option of the Learning Methods **Window** pull-down menu window will change the display from its appearance in Figure 5 to that in Figure 6. This display is the *expanded* **Learning Methods** display.

The top of the display is the same as in Figure 5. The bottom of the window, however, contains three lists. These lists contain parameters that control, or fine tune, the learning methods.

All the parameters have default values. Unless you have a clear understanding of a parameter, it is not wise to change it from its default value.

Each of these lists has a title, a list of parameters, an edit field and two push buttons. Selecting an item from the list of parameters will show its current value in the edit field. You can change the value of the selected parameter by editing this field, then saving it by clicking on the **Set Value** button, or by typing `<Return>`.

You can get a short help description of the parameter by selecting it from the list, then clicking on the **Help** button. The help message will be printed in the window where the command line is read. These lists can also be traversed using the mouse or arrow keys. The parameters in the three lists are:

Direction Parameters - These are parameters used by the direction methods.

Step Parameters- These are parameters used by the step methods.

Misc. Parameters - These are parameters specific to the simulator being used, or that are used by the network (as opposed to the minimize command).

For descriptions of these parameters either see the appropriate sman page (`minimize`, `Net`, or `bp`), or use the **Help** button.

[3] If you want to know more about the individual methods, see the sman page for `minimize`.

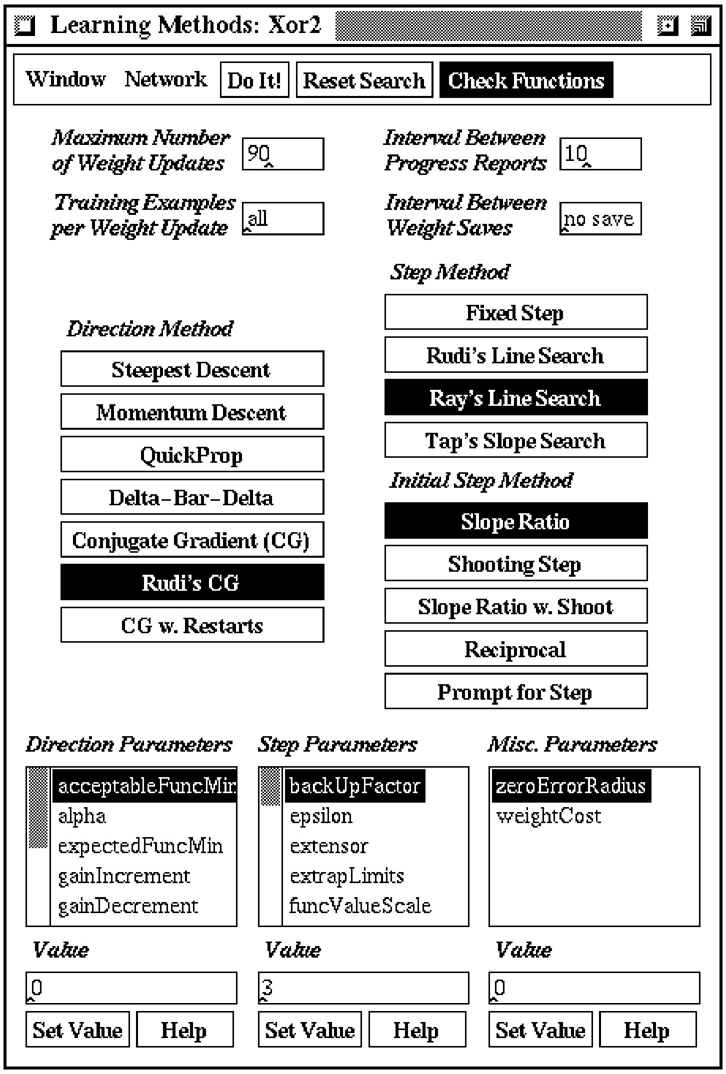

Figure 6: Learning Methods (Long Display)

4.4 Group Types

This display is only present in the **bp** module. It allows you to change the type mask of a group, which in turn changes its activation function, error measure , etc. As well you can change the cost function used by the network or an individual group.

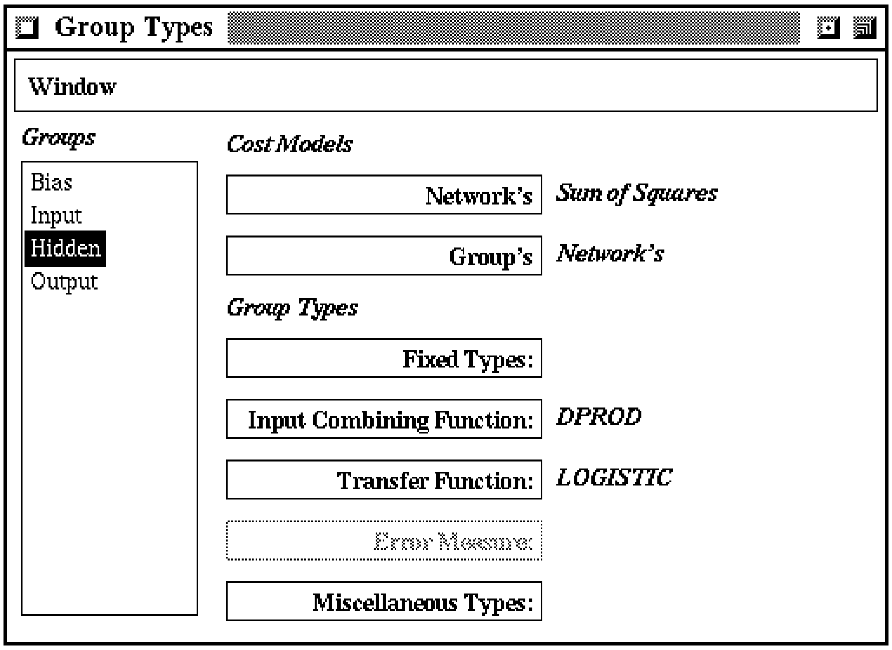

Figure 7: Group Types

The display in Figure 7 has a simple pull-down menu across the top which allows you to refresh and close the display. Down the left of the display is a list of all the groups in the current network. The rest of the display reflects the state of the group selected from the list (as well as the state of the current network).

The right side of the display consists of several pull-down menus with labels beside them. The first two menus deal with the cost models used by the network. These models are used to control the complexity of the network. The network may use a sum of squares cost model, a mixture of Gaussians cost model, or no cost model. Individual groups may have their own cost model, or use the network's. By default all groups use the network's.

The remaining pull-down menus are used to manipulate the type masks of the groups. Each menu manipulates a subset of the mask. The **Fixed Masks** cannot be changed. They are a read-only subset. The **Input Combining Function** set is used to control how the total input of each unit of a group is calculated. It contains masks such as DPROD and DISTANCE. The **Transfer Function** set controls how the total input is transferred to an output. It contains options such as LOGISTIC, TANH, and SOFTMAX. The **Error Measure** set (only for OUTPUT units) controls what error measure is used. It contains options such as SUMSQUARE, and CROSSENTROPY. The **Miscellaneous Types** set (obviously) controls masks which don't fit into any of the other sets. See Appendix B for a more complete discussion of these type masks.

4.5 Gaussian Mixture

If you are not using soft weight-sharing, you can safely skip this section, ignore this display, and go to Section 4.6.

Selecting the **Gaussian Mixture ...** option of the main display, will pop-up a dialog similar to the one shown in Figure 8. This dialog contains a list of all the Gaussian mixture cost models in the current network. These cost models are used to implement soft weight-sharing (try `help -k cost` to find out more about the cost models). The entire network may contain a single mixture cost model, and/or individual groups may have their own models. The name of each model (as shown in the list) will be the same as the network or group that contains it.

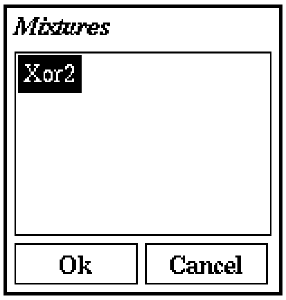

Figure 8: Mixture Dialog

To show the distribution of a cost model, select its name from the list, then click on **Ok**. The dialog will pop down, and a display similar to Figure 9 will pop up.

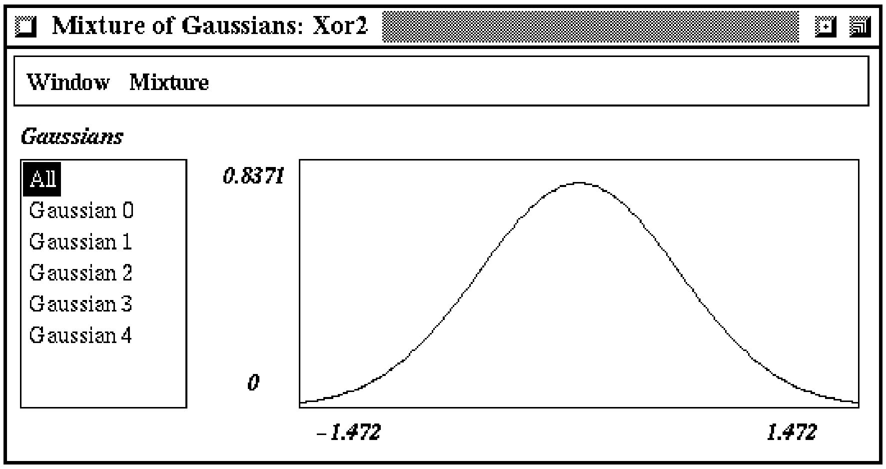

Figure 9: Mixture Display

As with all of the other displays, this one has a pull-down menu across the top. The **Window** pull-down menu controls redrawing, and closing of the display. There is an **Updates...** option, as in the **Activation**, and **Connection** displays. The **Mixture** pull-down allows you to change which mixture you are showing (if there are multiple ones in the network), and control miscellaneous aspects of the way the display is drawn.

The left side of the display contains a list of the Gaussians that make up the mixture. By selecting different Gaussians from this list, you can see the overall distribution, or the individual distributions of each Gaussian.

The central part of the display shows the distribution of the mixture (or a Gaussian in the mixture). If a single Gaussian is being shown, then the mean, variance and proportion of that Gaussian are also displayed.

4.6 Graphs

The next displays are **Graphs**. These are used to plot values from the simulator as they change through time (usually during training). Multiple graphs can be open at one time. Selecting the **New Graph...** button on the main window causes the dialog in Figure 10 to pop up.

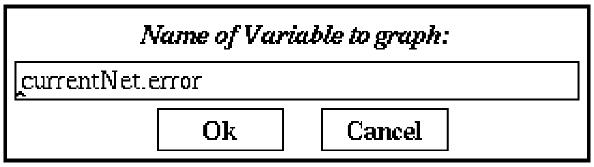

Figure 10: Graph Dialog

This dialog prompts for the name of the variable you wish to plot. The variable name should be entered in the same format as for the `show` or `set` command. Once the name has been entered, click the **Ok** button (or hit `<return>`). This causes the dialog to pop down, and the graph to pop up (see Figure 11).

The graph contains two areas: the menu pane, and the graph itself. The **Window** menu contains the same options as the other displays. The **Graph** menu contains items for:

- making the graph rescale itself vertically (if necessary) to better fit the data.
- shifting all values in the graph to the left, allowing new points to be added to the end.
- clearing the graph and resetting its vertical scale.
- setting miscellaneous parameters that affect the graph's appearance.

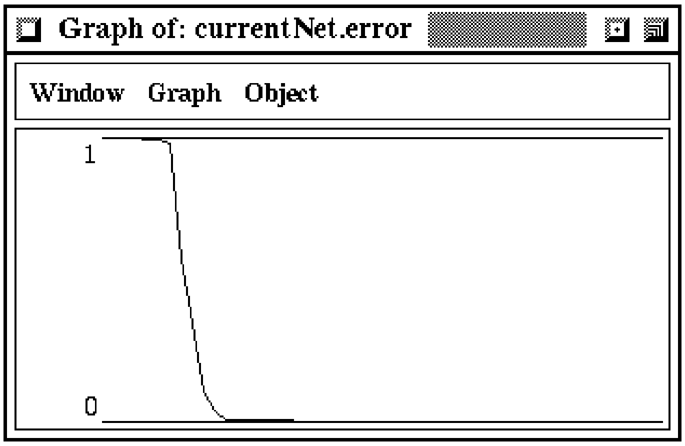

Figure 11: Graph

The **Object** menu allows you to plot the current value of the object on the graph, or to change what object is being plotted.

To plot the graph, select one of the options in the **Updates** dialog box, then train the network. At each update, a new point will be added to the graph.

4.7 Variables/Objects

This window (Figure 12) provides a graphical interface to the `show` and `set` commands. It can be used to examine and set simulator variables, fields within the network data structure, or other fields within any other data structure.

The pull-down menu pane across the top contains a menu for refreshing and closing the display. Below the menu pane is a read-only text field that contains the name of the object/variable currently being displayed. You can use the mouse to copy the variable name from this field and paste it into either an editor or the command line.

The main part of the display contains a scrolled window with the names of variables paired with a field containing either the type of, or value contained in, the variable. Depending on your current state, this window will contain different variables. You can traverse entire data structures by selecting items within this display.

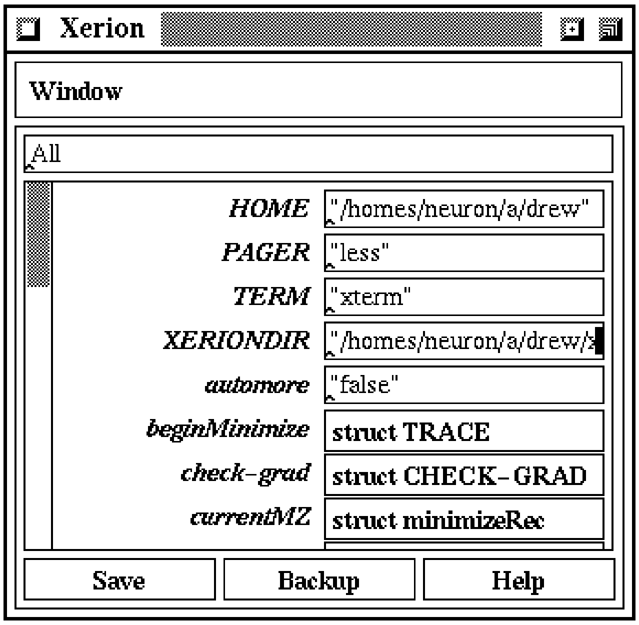

Figure 12: Variables/Objects

To examine, change and traverse the variables, use the right hand field of the variable pairs. Depending what type a variable is, the field will have a different type and action.

- If the variable is a simple one (not a structure, union, or array), then the paired field will be an editable text field. It will contain the current value of the variable which you may change then save.
- If the variable is simple, but is read-only, then the paired field will be a label containing the value. It cannot be changed.
- If the variable is an array, then the field will be a pull-down menu from which you can select which element within the array you wish to examine. Once you select an element, the display will change to show *just* that element of the array.
- If the variable is complex (a structure of union), then the field will be a push-button. Selecting the button will cause the display to show all of the fields in the structure.

The final part of the display (along the bottom) contains three push buttons. These buttons have the following actions:

Save - Save the values of all modified fields in the display.

Backup - If you are viewing a field within a structure, or an item within an array, backup to show the entire structure, or the entire array (the inverse of selecting an array item, or structure field).

Help - Give help about an object/variable.

4.8 Commands for Manipulating Displays

The following commands may be typed at the command line (or stored in a startup file). They each create, pop up, or manipulate the graphical displays in some manner.

buildDisplays - Builds the graphical display.

createGraph - create and pop up a graph for an object.

createMixtureDisplay - create and pop up a mixture display.

destroyDisplays - Destroys the graphical display.

doLayout - read an input file containing layout commands.

popup - pop-up one of the graphical displays.

triggerCallbacks - trigger callbacks associated with a display name.

xshow - show a structure object using a separate window.

Part II
The Xerion Command Interface

The **Xerion** command line interface is used to read and execute commands from a terminal or a file. It is similar in form to a **UNIX** shell but has restricted syntax and some *very* different commands. This part of the document describes how to use the **Xerion** interface.

5 Command Syntax

First, we need to define a few terms. A name is a sequence of letters, digits, or dashs beginning with a letter or dash. A command is a sequence of non-blank words separated by blanks (spaces tabs or new-lines). The first word of the command is the name of the command to be executed, the remaining words are passed as arguments to the command. When a command completes, it returns an exit status and sets a variable called **status** to this value. The return status is 0 upon successful completion. If the command exits abnormally, the variable will be set to a non-zero number indicating the type of error encountered.[4]

Normally, commands are delimited by a new-line. If a command is too long to enter on a single line, the command may be continued on the next line by ending the first line with a backslash (\). A list of commands may be entered on a single line separated by semicolons (;). A list of commands following a semicolon is only executed if the list before the semicolon returns a zero exit status.

5.1 Startup

When the **Xerion** simulator is started up it looks for the files (in order):

```
$XERIONDIR/config/xerionrc
$XERIONDIR/config/<name>rc
$HOME/.xerion
./.xerion
```

(where `<name>` is the name of the simulator, e.g. `bp`). These files can contain any commands that you want. In particular, the default files contain commands that set the `read-path` variable.

Make sure that `$XERIONDIR` (and any other variable within **Xerion**) does not contain the character `@`. This character is used for output redirection (see the help page for `redirection`), and may cause unexpected things to happen.

5.2 Comments

The character # causes itself and all the following characters up to a new-line to be ignored.

5.3 Parameter substitution

The character $ is used to introduce substitutable parameters. The **Xerion** interface implements keyword (named) parameters. Keyword parameters (also known as variables) may be created by using **var** and assigned values by using **set** (see Section 6.2). A command and variable may share the same name.

[4] The valid values for **status** errors are defined in the header file **xerion/itf.h**

Parameters may be one of several types: `String, int, Real, float` or `double`. The `Real` type defaults to either `float` or `double` depending on how **Xerion** was compiled. As well, parameters may be arrays of these types.

Parameters are substituted using the following syntax: `$parameter` causes the value, if any, of the parameter to be substituted; if the parameter name contains characters other than letters, digits, or dashes, or if it is followed by a letter, digit, or dash that is not to be interpreted as part of its name, then the syntax `${parameter}` must be used. Individual elements of an array parameter are accessed by using the syntax `${parameter[n]}`, where `n` is either an integer or the name of an integer variable. For example:

```
xerion-> var Real error[3]
xerion-> var int idx
xerion-> echo "idx = $idx"
idx = 0
xerion-> echo "error[idx] = ${error[idx]}"
error[idx] = 0
```

Parameter substitution may also be performed on network objects and data structures. For example, the value of `currentNet.error` may be substituted using `${currentNet.error}`.
By default, it is an error to substitute the value of a variable that has not been created. However, if the (automatically created) variable `unset-error` is set to `false`, the value of an uncreated variable is an empty string.

5.4 Command substitution

The standard output from a command enclosed in parenthesis preceded by a dollar sign (`$()`) or a pair of grave accents (`‘‘`) may be used as part or all of a word; new-lines and surrounding white space are compressed to a single space. You are discouraged from using the second (archaic) form. In this form, the string between the quotes is processed for special quoting characters before the command is executed (see section 5.5 on “Quoting”). Command substitution may be nested.

As an example of command substitution, consider setting a string variable to the structure name of a group:

```
xerion-> var String name
xerion-> set name = $(groupLongName Bias)
xerion-> echo "name = $name"
name = currentNet.group[0]
```

5.5 Quoting

The following characters have special meaning to the interface and cause termination of a word unless quoted:

```
; = :  , { }( ) < > | % @ new-line space tab
```

A character may be quoted (i.e. made to stand for itself) by preceding it with a backslash (\) or inserting it between a pair of quote marks (`''`or `""`). During processing, the interface may quote certain characters to prevent them from taking on special meaning. Backslashes used to quote a single character are removed from the word before the command is executed. The pair `\new-line` is removed from a word before command and parameter substitution.

All characters enclosed between a pair of single quote marks (`''`), except a single quote, are quoted by the interface. Backslash has no special meaning inside a pair of single quotes. A single quote may be quoted inside a pair of double quote marks (for example, `"'"`).

Inside a pair of double quote marks (`""`), parameter substitution occurs and the interface quotes the results to avoid blank interpretation.

5.6 Script files

Xerion implements script files that can be executed much like shell script files; that is, they may be executed by entering their name at the **Xerion** prompt. Alternately, they may be executed by giving the file name as an argument to the `exec` command. Script files are found by searching the `readpath` variable for a file with the given name and whose first line consists of *exactly* the string `#!xerion`. If an error occurs while executing a script, it will be exited immediately.

Before the script is executed, the following variables are created and set:

`$tokc` The number of tokens (positional parameters) from the command line used to execute the script.

`$tokv` A String array containing `$tokc` elements. These elements are the tokens from the command line used to execute the script.

`$*` The complete command line used to execute the script.

A sample script file is shown and discussed in Appendix C.

6 Built-in Commands

The following commands are built into the **Xerion** interface. They are not specifically for building networks, but are of more general use. For complete (and up-to-date) descriptions of them, use the **Xerion** online help facilities.

6.1 Conditionals

Commands in this section may be used to conditionally or iteratively execute other commands.

`for name in word ...  ; do ; list ; done`

Each time a `for` command is executed, `name` (which must be a simple variable or object) is set to the next word taken from the `in word` list. The commands in `list` are then executed. Execution ends when there are no more words in the `in word` list.

`if command ; then ; list ; [elif ; command ; then ; list ;] ...  [else ; list ; ] fi`

The command following `if` is executed and, if it completes successfully, the list following the first `then` is executed. Otherwise, the command following `elif` is executed and, if it succeeds, the list following the next `then` is executed. Failing that, the `else` list (if present) is executed. A test command may be negated by preceded it with the `!` character.

`while command ; do ; list ; done`
A while command repeatedly executes the while `command` and, if the exit status of the command is zero, executes the commands in `list`; otherwise the loop terminates. If no commands in `list` are executed, then the `while` command returns a zero exit status; `until` may be used in place of `while` to negate the loop termination test. As well, the test may be negated by preceding it with the `!` character.

6.2 Parameter and Variable Manipulation

Commands in this section may be used to set, show, or otherwise manipulate variables, parameters, or network objects.

`default name value`
Specifies the default value for a structure field.

`echo [-n] [arg ...]`
Echo prints its arguments to the standard output, followed by a new-line. If the -n option is given, the trailing new-line is omitted.

`export name ...`
The `String` variables given are put in the environment (and are passed to subprograms). If the variable does not exist in the simulator, but does already exist in the environment, then it is created in the simulator and its value is copied from the environment. Each time the value is changed in the simulator, the copy of it in the environment is also changed.

`format name [format]`
Sets the default print format for a parameter, variable or structure field.

`fpPrecision <Real-prec> <float-prec> <double-prec>`
Sets the precision for printing floating point values.

`print [name [format]] ...`
Prints out the values of the given parameter or variables in the desired format. Formats are a subset of those used by `printf`.

`reset name`
Resets the fields of structure `name` to their defaults as specified by the `default` command.

`set [name [= value]] ...`
Sets the value of parameter, or variable name. If `value` is omitted, then `name` is set to `true`. If no arguments are given, the values of all parameters are printed.

`show [name] ...`
Show prints the values , with labels, of parameters, simple objects, structures, or array objects. It can be used for displaying objects which are too complex for `print`.

`unset name ...`
Sets the value of parameters to “false”.

`var type name [<digit>]`
Creates a variable or vector of the given type. Type should be one of: `String, int, Real, float, double`.

`visible +/- name`

This command sets the value of the visibility flag for fields of structures. If the visibility flag is off for a particular field in a structure type, then when object of that structure type is shown using `show`, that field will not appear.

6.3 Traces

The **Xerion** interface contains objects called **traces** which can be used to execute commands automatically.

Traces are compiled into the **Xerion** simulators. New traces can only be added by recompiling the simulator.

Traces can be thought of as trigger-points. At some point while **Xerion** is running a trace will be triggered. To see what traces are built into the simulator you are using, use the command `showTrace`. The triggering of a trace is not in and of itself useful. What is useful, is that commands can be added to a trace (see `addTrace`). When the trace is triggered, it will execute, in order, all of the commands that have been added to it. It is very common to add `print`, and `show` commands to a trace.

For example, most simulators contain the trace `endMinimize`. This trace is triggered when the `minimize` command completes. To print the network error automatically whenever `minimize` finishes, use the following command:

```
xerion-> addTrace endMinimize "show currentNet.error"
```

The commands that can be used to add, delete, and otherwise manipulate traces are:

`addTrace name [+/-active] [-freq f] [+/-N] command ...`

Add a command to the trace `name`. If the command is already on the trace, it is not added, but is turned on (`+active`) or off (`-active`) as appropriate, and moved to the end of the trace. The command will be executed every `f` times the trace is called (default 1). The `+/-N` argument will insert the trace after/before the `N`th command already on the trace.

`deleteTrace name [N/all]`

Deletes permanently the `N`th or all commands from a trace.

`doTrace [+all] name [N]`

Executes the `N`th or all commands associated with a trace.

`showTrace [-verbose] [name] [N]`

Prints out details of the named traces or all traces. The -verbose format is suitable for saving to a file, editing, and then reading back in. This is a convenient way to change traces.

`traceOff name [N ...]`

If no trace number is supplied the whole trace is turned off. If a trace number is supplied, just that command in the trace is turned off. A range of values may be supplied for the trace numbers, e.g. “1-4”.

`traceOn name [N ...]`

If no trace number is supplied the whole trace is turned on. If a trace number is supplied, just that command in the trace is turned on. A range of values may be supplied for the trace numbers, e.g. “1-4”.

6.4 Miscellaneous Commands

`alias [name [def]]`

Assign `def` to the alias `name`. `def` is a single (quoted) string that may contain escaped variable substitutions. `name` is not allowed to be alias or unalias. If `def` is omitted, the alias `name` is displayed along with its current definition. If both `name` and `def` are omitted, all aliases are displayed.

> Due to bugs in command processing, aliases are disabled by default. To enable them set the variable `alias-enabled` to `true`.

`args [arg ...]`

Prints its arguments, with labels, one per line.

`background [infile] [outfile]`

Disconnects stdin and stout from the terminal and connects them to files. This allows the process to be stopped and backgrounded so the user can logout.

`calc [arg ...]`

Evaluates a floating point arithmetic expression and prints the result on stdout.

`cd name`

Changes the current working to `name`. The read command will be affected by this if the `readpath` variable contains any relative path names (e.g. `.`).

`close name`

Close a stream which has been opened by `open`. The stream is kept, but closed and remains inactive. It can be reopened with the command `reopen`.

`deleteStream [-q] name`

Deletes a stream permanently. If the -q option is given, do not complain if the stream does not exist.

`exec`

The `exec` command sources and executes a xerion shell script, passing in its arguments as positional parameters. Xerion shell scripts must contain the exact string `#!xerion` as the first line of the file. The script is searched for in the directories listed in the variable "readpath" (see `read` below). When a script is executed the following parameters are automatically created and set:

`$tokc` The number of tokens (positional parameters) from the command line used to execute the script.

`$tokv` A String array containing `$tokc` elements. These elements are the tokens from the command line used to execute the script.

`$*` The complete command line used to execute the script.

If an error occurs while executing a script, it will be exited immediately.

`exit`

If the simulator is reading a file, it will stop reading it. If the simulator is at the top level (i.e. prompting), it will quit.

`hello`

Prints out the string "Hello!".

`help [[-k] name]`
: The most useful command in **Xerion** is `help`. All commands have a help message which explains how the command is to be used, and what it does. Try typing `help help` inside the **Xerion** interface for more information.

`ls [name ...]`
: List the specified directory or file. If no directory is specified the current working directory is listed.

`nice [value]`
: `Nice` increments the nice value of the current process. It is not possible to decrement the nice value. If no arguments are given, the nice value is not changed, just printed out.

`open name <redirection>`
: Open a stream permanently, and associate it with a name. To reuse a stream name, the stream must be deleted with "deleteStream", otherwise it hangs around waiting to be reopened. `<redirection>` is one of the forms described in Section 7.3.

`prompt name string`
: The prompt `string` is printed to standard output and one line is read from standard input. The line read is then stored in the variable `name`.

`pwd`
: Print the current working directory.

`quit`
: Quits the simulator.

`read name ...`
: Reads a file containing **Xerion** commands. The file is searched for in the directories listed in the variable "readpath" (unless the filename begins with a '/'). The current directory will not be searched unless "." is in `readpath`. The directories in `readpath` should be separated by a comma, colon, or space.

`redirection`
: Prints documentation about streams and I/O redirection.

`reopen name`
: Reopen a currently inactive stream. The stream type is changed from its previous type.

`same`
: Compares two strings and completes successfully if the two are identical, and fails if they are different. It is most useful in an "if", or "while" test.

`segv`
: Causes a segmentation violation. This command is useful for checking signal handling.

`sh [-echo] command`
: The shell command is executed as typed, except that any escaped characters are converted to the character they stand for, e.g. '\n' is converted to the new-line character .

  Quote characters are retained in the line sent to the shell, except if the interface sees only one token in the shell command, in which case the quotes are omitted. This allows the quoting of shell commands in order to get the shell and not the interface to interpret redirection and other meta-characters.

`showStream [name ...]`
    Prints out details of the named stream, or if no stream name is supplied, of all streams.

`unalias [name]`
    Discard the alias with name `name`. All aliases are deleted by `unalias *`.

> Due to bugs in command processing, aliases are disabled by default. To enable them set the variable `alias-enabled` to `true`.

`whatis [name]`
    Prints out details of command, variable, type, field, or structure **name**.

7 Miscellaneous

7.1 Automatically Created Parameters

The following parameters are automatically created by the **Xerion** interface.

HOME - The default home directory of the user (imported from the invoking shell).

PAGER - The default pager used to show command output (imported from the invoking shell).

TERM - The terminal type the shell is running in (imported from the invoking shell).

XERIONDIR - The top level directory for the **Xerion** files (imported from the invoking shell).

alias-enabled - If true, perform alias expansion. By default, this is `false` (due to bugs in the command parser).

automore - If set to `true`, pass all command output through the `$PAGER`.

echo - If `true`, echo commands before executing them.

help-automore - If set to `true`, pass output from `help` through `$PAGER`.

noclobber - If set to `true`, return an error when redirection will overwrite an existing file.

plus-minus-options - If this is `false`, then commands will not accept options preceded by `+` (default: `false`).

printstatus - If set to `true`, print exit status of each command list after it completes.

prompt - Primary prompt string (default "`xerion->`").

read-echo - If both `echo` and `read-echo` are true, echo commands from read-files and scripts before executing them.

readpath - A list of space or colon separated directories to search for read-files and scripts. This variable is used by all commands which take input from files.

redirection - If `true` allow output redirection.

status - The exit status of the last completed command list.

unset-error - If true, it is an error to try to substitute the value of a variable that has not been created or set. If false, the substitution results in the null string. The default value is true, for backwards compatibility.

7.2 Blank Interpretation

After parameter substitution, explicit null arguments `""` or `''` are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.

7.3 Output Redirection

A command's output may be redirected using a special notation interpreted by the interface. The following may appear anywhere in a simple command or may follow a command and are not passed on as arguments to the invoked command. Note that parameter and command substitution occurs before word or digit is used. To disable command redirection, set the parameter `redirection` to `false`.

>word - Use file `word` as standard output for the command. If the file does not exist it is created; otherwise, it is truncated to zero length. If `noclobber` is set to `true`, and the file exists, the command will be aborted and the file unchanged.

>>word - Use file `word` as standard output. If the file exists output is appended to it (by first seeking to the end-of-file; otherwise, the file is created. This command is unaffected by the value of `noclobber`.

|word - Use the output of the **Xerion** command as standard input to the **UNIX** command `word`.

% digit word - Redirect standard output to the socket number `digit` on host `word`.

@ word - Use the stream `word` opened with the `open` command as standard output.

Part III
Programming in Xerion

8 Writing New Simulators

To demonstrate writing new simulators with **Xerion** we will use the example of a simple Back Propagation simulator (the prototype of the **bp** program). The source code for the real module is in the directory $XERIONDIR/src/sim/bp. Only important parts of the files will be illustrated here. For the full implementation and support procedures, see these source files.

Note that all library procedures and object types described in this section have online man pages that can be viewed using **sman**.

8.1 Creation and Destruction Hooks

The networks in **Xerion** are built using C objects (structures). These objects are `Units`, `Links`, `Groups`, `ExampleSets`, and `Nets`. The header files containing the definitions of these object types can be found in $XERIONDIR/include/xerion/*.h. The header files also contain references to the library functions and procedures which act on the objects.

Each of these object types has a creation function and destruction procedure. For the **Net** object these are:

```
Net createNet (String name, Mask mask, int timeSlices) ;
void destroyNet (Net net) ;
```

Calling `createNet` allocates the memory for the network, initializes it, and returns a pointer to the newly created object. `destroyNet` destroys all children (e.g. Units, Links) of the net and then frees the memory. Similar functions exist for the other objects. *ALWAYS* use the create function to get new objects and the destroy function to free them.

To let you modify the objects when they are created and initialize them as you want, the simulator provides creation and destruction "hooks". These are procedures that will be called by the creation and destruction routines as the last (first) thing they do. That is, after the create function has allocated memory and initialized the object it will call the create hook. Then it will return the object. The destroy procedure will immediately call the destroy hook, then it will go about destroying the object.

Each object type has procedures for setting these hooks. For the `Net` they are:

```
void setCreateNetHook (NetProc initHook) ;
void setDestroyNetHook (NetProc deinitHook) ;
```

The hook procedures should be defined as:

```
void initHook (Net net) ;
void deinitHook (Net net) ;
```

Using the hooks, you can initialize values in the objects, allocate or free memory, etc. The Network hooks are described in Section 8.3 and shown in Figure 15.

8.2 The Main Function

The main procedure for **Xerion** is very simple (see Figure 13). It starts with simulator dependent initialization. Usually this consists of setting the create and destroy hooks for some of the objects. Next it calls the procedure `IStandardInit` which initializes **Xerion**. Finally it calls `ICommandLoop`. This loop reads input from the command line and processes input from the graphical user interface. In practice, this procedure never returns.

Make sure you do any private initialization before calling `IStandardInit`. It parses the command line, and may attempt to read files, or build networks that depend on your initialization.

```
/***********************************************************************
 *      Name:           main
 *      Description:    the main function, used for the xerion simulator
 *      Parameters:
 *              int     argc    - the number of input args
 *              char    **argv  - array of argument strings from command
 *                                line
 *      Return Value:
 *              int     main    - 0
 ***********************************************************************/
int main (argc, argv)
  int   argc ;
  char  **argv ;
{
  /* Insert any private initialization routines here */
  setCreateNetHook(initNet) ;
  setDestroyNetHook(deinitNet) ;
  setCreateGroupHook(initGroup) ;

  /* Perform initialization of the simulator */
  IStandardInit(&argc, argv);

  /* Enter loop that reads commands and handles graphics */
  ICommandLoop(stdin, stdout, NULL);

  return 0 ;
}
/***********************************************************************/
```

Figure 13: Main procedure for Xerion

8.3 Extension Records

The network objects in **Xerion** are quite general; however, every simulator will need to add some extensions or new fields to them. To allow new fields to be added without recompiling the libraries each object has an **Extension Record**. The extension record is simply a pointer to an unknown type of structure. By defining these structures in the simulator header file, and allocating them in the creation hook, new fields can be dynamically added to the objects.

The definitions of the extension records for the Back Propagation module (found in **unit.h**) are shown in Figure 14. The `/* BIND */` comment is used to tell the command interface that the variable type may be manipulated from the command line through `show` and set. `/* netParam:   */` tells the interface that the field should be shown in the **Learning Methods** display in the **Misc. Parameters** list. For a complete explanation of the `/* BIND */` and `/* netParam: */` comments see the sman page for **create-bindings**.

Previous version of **Xerion** required all the extension records to be defined in the simulator header. This is no longer required. You need only define the extension records you plan to use. It is the user's responsibility to allocate and free the memory for the extension records. This may be done in the creation and destruction hooks (see Figure 15).

The create hook is usually used to allocate memory, initialize fields and to set the methods used by the objects.

```
/*********************************************************************/
typedef struct NetExtensionRec {        /* BIND */
  Real          zeroErrorRadius ;       /* netParam: */
} NetExtensionRec ;

typedef struct GroupExtensionRec {      /* BIND */
  CombInProc    combIn ;                /* calculate totalInput */
  CombInProc    combBack ;              /* fan derivatives back */

  TransferProc  transfer ;              /* calculate output */
  TransferProc  transferBack ;          /* backprop error   */

  ErrorProc     contributeError ;       /* calculate output */
  ErrorProc     contributeErrorBack ;   /* calculate output */

  Real          sum ;                   /* miscellaneous sum */
  Variable      errorStdDev ;           /* nosave: */

  /* stuff used by competing experts */
  Mixture       mixture ;
  Group         gateGroup ;             /* nosave: */
} GroupExtensionRec ;

typedef struct UnitExtensionRec {       /* BIND */
  Variable      gain ;
  Group         expert ;
  Gaussian      gaussian ;
} UnitExtensionRec ;
/*********************************************************************/
```

Figure 14: Extension Records for Backprop Simulator

```
/*********************************************************************
 *      Name:           initNet
 *      Description:    sets the error calculation procedures for
 *                      a network as well as allocating the memory
 *                      for the extension record
 *      Parameters:
 *              Net     net - the net to act on
 *      Return Value:   NONE
 *********************************************************************/
static void     initNet (net)
  Net   net ;
{
  net->calculateErrorDerivProc = calculateNetErrorDeriv ;
  net->calculateErrorProc      = calculateNetError ;

  net->extension = (NetExtension)calloc(1, sizeof(NetExtensionRec)) ;

  MzeroErrorRadius(net) = 0.2 ;
}
/********************************************************************/
static void     deinitNet (net)
  Net   net ;
{
  if (net->extension != NULL)
    free(net->extension) ;
}
/********************************************************************/
```

Figure 15: Create and Destroy Hooks for the Net

8.4 Methods

Some of the objects in the simulator have methods attached to them. The most important of these are the error calculation methods for the `Net` object and the unit update methods for `Groups`. Macros are provided for calling these methods. The naming convention used for these macros is to begin each with a `M` followed by the method name with each word starting with caps. (e.g. `MupdateNetActivities(net)`). The object the method acts on is passed as a parameter.

The `Net` method fields are `calculateErrorDerivProc` and `calculateErrorProc`. The error derivative procedure is used during training. It is responsible for processing examples through the net and updating the network error and the derivative fields in the links. The error procedure is only responsible for updating the error. These two procedures are shown in Figures 16 and 17.

Methods must also be supplied to activate units, and to update the derivatives of the links. These methods are stored on the `Group` objects. The names of these fields are `unitActivityUpdateProc` and `unitGradientUpdateProc`. The activity update procedure should transfer the unit's total input to an output. If this output affects the network error it should also update that. In the **bp** module this procedure is called `forward` (see Figure 18).

The gradient update procedure should update the derivative fields in all incoming links to the unit. In the prototype for **bp** this procedure is called `backward` (see Figure 18).

```
/*********************************************************************
 *      Name:           calculateNetError
 *      Description:    error calculation procedure for backprop net
 *                      It processes 'MbatchSize(net)' examples
 *      Parameters:
 *              Net             net - the net to use
 *              ExampleSet      exampleSet - the examples to use
 *      Return Value:
 *              NONE
 *********************************************************************/
static void     calculateNetError(net, exampleSet)
  Net           net ;
  ExampleSet    exampleSet ;
{
  int           numExamples ;

  net->error = 0.0 ;
  for (numExamples = 0 ; numExamples < MbatchSize(net) ; ++numExamples) {
    MgetNext(exampleSet) ;
    MupdateNetActivities(net) ;
  }
  if (numExamples <= 0)
    IErrorAbort("calculateNetError: no examples processed") ;

  /* update the cost after everything else is done */
  MevaluateCost(net) ;
}
/*********************************************************************/
```

Figure 16: Error Update Method for the Net

```
/**********************************************************************
 *      Name:           calculateNetErrorDeriv
 *      Description:    gradient calculation procedure for backprop net
 *                      It processes 'MbatchSize(net)' examples
 *      Parameters:
 *              Net             net - the net to use
 *              ExampleSet      exampleSet - the examples to use
 *      Return Value:
 *              NONE
 **********************************************************************/
static void     calculateNetErrorDeriv(net, exampleSet)
  Net           net ;
  ExampleSet    exampleSet ;
{
  int           numExamples ;

  /* zero the net error and all derivative fields in the links */
  net->error = 0.0 ;
  for (idx = 0 ; idx < net->numLinks ; ++idx)
    net->links[idx]->deriv = 0.0 ;

  /* For each example   - zero the derivative fields in the units
   *                    - do a forward pass updating the activities
   *                    - do a backward pass updating the derivatives
   */
  for (numExamples = 0 ; numExamples < MbatchSize(net) ; ++numExamples) {
    MgetNext(exampleSet) ;

    netForAllUnits(net, ALL, zeroUnit, NULL) ;
    MupdateNetActivities(net) ;
    MupdateNetGradients(net) ;
  }

  if (numExamples <= 0)
    IErrorAbort("calculateNetErrorDeriv: no examples processed") ;

  /* update the cost after everything else is done */
  MevaluateCostAndDerivs(net) ;
}
/**********************************************************************/
```

Figure 17: Error and Derivative Update Method for the Net

```
/**********************************************************************
 *      Name:          forward / backward
 *      Description:   update procedures for units
 *      Parameters:
 *              Unit    unit - the unit of concern
 *      Return Value:
 *              NONE
 **********************************************************************/
static Proc     forward(unit)
  Unit  unit ;
{
  /* sum the inputs and pass the result through a sigmoid */
  dotProduct(unit) ;
  unit->output = sigmoid(unit->totalInput) ;

  /* update the network error for output units */
  if (unit->group->type & OUTPUT)
    unit->net->error += square(unitError(unit)) ;
}
/**********************************************************************/
static Proc     backward(unit)
  Unit  unit ;
{
  /* add the deriv from the error if the unit is an output one */
  if (unit->group->type & OUTPUT)
    unit->inputDeriv += 2.0*sigmoidDeriv(unit->output)*unitError(unit) ;

  /* add the deriv from any backpropagated errors */
  unit->inputDeriv += sigmoidDeriv(unit->output)*(unit->outputDeriv) ;

  /* propagate the derivs back through the incoming links */
  backDotProduct(unit) ;
}
/**********************************************************************/
```

```
The unit methods have become significantly more complex in later
version of bp.  The ones shown in this figure implement simple
dot-product combining functions, and sigmoid activations, with
sum-square error.
```

Figure 18: Unit Methods

9 Writing New Commands

This section describes how to extend the interface by programming your own commands. Commands are automatically extracted from object modules and libraries used by **Xerion**. Any function whose name starts with the string `command_` is assumed to define a command. For example, the command **set** has a corresponding function whose name is `command_set`. All commands in **Xerion** are created this way.

To write your own command, simply create a function that follows this naming convention and is defined as in Figure 19. The function will be passed two arguments, `tokc` and `tokv`. These are analogous to the arguments passed to a C main routine: `argc` and `argv` (the number of command line arguments, and the corresponding vector of arguments, where `argv[0]` contains the command name).

> Unlike a main routine in C the command function should return zero on *failure* and non-zero on success. If it does not, and it is included in a read file, then the read may abort.

The first part of the function must contain the help section as shown in Figure 19. This is used by the **help** command to give a short synopsis of the command or a long help message (try `sman help` for more information). After the help section you can put the body of the function.

To simplify input and output, two global file descriptors exist. They are `din` and `dout`. All reading should be done from `din` and all normal writing should be done to `dout`. `stderr` can still be used for error reporting.

To aid in error recovery, a routine called `IErrorAbort` is available. When called, it prints an error message to `stderr` and aborts the command. It never returns. Its arguments are a format string and associated arguments (similar to `printf`) that are to be printed as a description of the error.

> The global variable `currentNet` (type `Net`) points to the current network used in the simulator. User commands should never directly modify its value, but they may modify the net that it points to.

A template for writing commands is shown in Figure 19 on page 42 and can be found in the file $XERIONDIR/templates/command.c.

```
#include <xerion/useful.h>
#include <xerion/simulator.h>
#include <xerion/commands.h>

/*********************************************************************
 * Name: command_template
 * Description: Template for user commands. Any function with
 * the name command_<name> will be extracted
 * automatically from the object module by
 * extract-commands.
 * int tokc    - the number of command line tokens
 * char *tokv[] - the vector of tokens
 * Return Value:
 * int command_template - 0 on failure, 1 on success
 *********************************************************************/
int command_template(tokc, tokv)
  int tokc ;
  char **tokv ;
{

  IUsage("template options");
  if (GiveHelp(tokc)) {
    ISynopsis("Template for user command");
    IHelp
      (IHelpArgs,
       "Summary of how the command works.",
       "...",
       "EXAMPLES",
       "SEE ALSO",
       NULL) ;
    return 1 ;
  }

  /* Parse any command line options */

  /***
   * Do whatever you want in here.
   * Note that all writing should be to external FILE * 'dout'.
   */
  fprintf(dout,"Command 'template' at your service!\n");

  /***
   * In case of error call
   *   IErrorAbort("%s: error is with %d", tokv[0], 88);
   * This call returns you to the command prompt and prints the
   * error message.
   */

  return 1 ;
}
/*********************************************************************/
```

Figure 19: Template for writing a command

Appendices

A Commands for Working with Networks

Networks are built and trained using a subset of the commands built into **Xerion**. Although the commands for building a network can be typed in at the prompt, it is easier to save them in a file, and read the file in using the `read` command. `Read` searches the directories in the `readpath` variable to find files (see Sections 6 and 7).

A.1 Commands for creating Networks

The commands available for building networks are:

addExamples - add examples to the specified example set of a net.

addGroup - create a group of units in the current network.

addMixtureCost - add a mixture of Gaussians cost model to the net.

addNet - create a network and add it to the list of existing nets in the simulator.

addSumSquareCost - set the network cost model to use sum of squares.

addUnit - add a unit to an existing group in the current network.

connectBlocks - connect blocks of units in one group to units in another.

connectGroups - connect two groups of units to each other.

connectUnits - connect two units to each other.

constrainGroup - constrain incoming links of one group of units with those of another.

constrainLink - constrain one link to be a linear multiple of another.

constrainUnit - constrain incoming links of one unit with those of another.

deleteCost - delete the network cost model.

deleteExamples - delete examples from the specified example set.

deleteGroups - delete groups of units from the current network.

deleteNets - delete networks from the simulator.

deleteUnits - delete units from the current network.

disconnectGroups - disconnect two groups of units from each other.

disconnectUnits - disconnect two units from each other.

freezeGroup - freeze the weights of a group's incoming links.

freezeLink - freeze the weight of a link.

freezeUnit - freeze the weights of a unit's incoming links.

loadWeights - load a weight file into the current network.

moveExample - move an example from one set to another.

moveExamples - move all examples from one set to another.

orderGroups - set the order of processing for the groups in the current net.

randomize - randomize all the weights of the links in the current net.

resetMixtureCost - reset the mixture of Gaussians cost model in the net.

saveWeights - Save all weights in the current network to a file.

setLink - set the weight of a link.

unconstrainGroup - remove constraints from incoming links to units in a group.

unconstrainLink - remove any constraints on a link.

unconstrainUnit - remove constraints from incoming links to a unit.

unfreezeGroup - unfreeze the weights of a group's incoming links.

unfreezeLink - unfreeze the weight of a link.

unfreezeUnit - unfreeze the weights of a unit's incoming links.

useNet - set the current network, or list all the networks in the simulator.

A.2 Commands for Training Networks

The following commands can be used for training a network.

cg - train a network using conjugate gradient descent

checkGrad - check partial derivatives for accuracy.

deltaBarDelta - train a network on the examples using deltaBarDelta.

initMinimize - initialize the minimize record that is used for optimizing the weights.

minimize - train a network on the examples using one of the minimization methods.

momentum - train a network on the examples using momentum descent.

quickProp - train a network on the examples using quickProp descent.

steepest - train a network on the examples using steepest descent.

train - train a network on the examples using momentum descent.

A.3 Miscellaneous Commands

The following commands do not actually help build a network, but they do act on networks. As you become more familiar with the simulator, they may come in handy.

doExample - process a specific example through the net.

doExamples - process an example set through the net.

doLayout - read an input file containing layout commands (used to build the graphical displays).

groupLongName - return the long name of a group.

groupTypes - print out the types for all groups.

oldLoadWeights - load an old style weight file into the current network.

permuteExamples - permute a set of examples.

rangeTerminate - stop minimizing if all outputs are within range of their targets.

seed - seed the random number generator (used in randomizing weights).

showCost - print the cost of a network.

showMixture - show the mixture of Gaussians cost model.

unitLongName - return the long name of a unit.

B Different Types of Units

In **Xerion**, unit behaviour is controlled by type masks. All units in a group have the same type, and thus the same behaviour. Different **Xerion** modules implement different types of units; the module with by far the largest variety of types is the bp module.

The unit type masks in the bp module control behaviour in 5 categories: Fixed Behaviour, Input Combining Functions, Transfer Functions, Error Measures, and Miscellaneous Types. A single unit can have one (or more) types from each of these categories.

Types controlling fixed behaviour must be specified when a group of units is created. They cannot be changed after this point. Types that fall into this category include:

INPUT units have their output value clamped when an example is presented to a network. They have no transfer function or input combining function.

OUTPUT units are given a target value when an example is presented to a network. They *must* have an error measure function. Units not having type OUTPUT *must not* have an error measure function.

BIAS units always have an output of 1. As with INPUT units, they have no transfer function or input combining function.

GATE units are used in mixture of experts networks to gate the outputs of the indivdual expert groups (see the online help for the `gate` command).

GATED units are output units associated with a mixture of experts network. They are only used as a place to store target values.

The input combining function of a unit takes its pre-unit outputs and combines them together using its incoming weights. This becomes the unit's total input. The bp module contains three types of input combining functions:

DISTANCE units calculate their total input as the euclidean distance between the vector of incoming weights, and the associated vector of pre-unit outputs.

DPROD units calculate their total input as the dot-product of the vectors of incoming weights and pre-unit outputs.

PI units have all incoming weights constrained to be equal. Their total input is the product of this weight and all pre-unit outputs.

Transfer functions control how a unit changes its total input to an output value. If we let x_i be the total input to unit i, and y_i be its output given this input we can write the formulæ for the six transfer functions implemented in the bp module:

LOGISTIC units use a sigmoidal transfer function. Output is on the range $(0, 1)$.

$$y_i = \frac{1}{1.0 + e^{-x_i}}$$

LINEAR units use no transfer function. Their ouput is the same as their input.

$$y_i = x_i$$

TANH units use a hyperbolic tan activation function. Output is on the range $(-1, 1)$.

$$y_i = \tanh(x_i)$$

EXPONENTIAL units use an exponential transfer function. Output is on the range $(0, \infty)$.

$$y_i = e^{x_i}$$

NEGEXPONENTIAL units use a negative exponential transfer function. Output is on the range $(-\infty, 0)$.

$$y_i = e^{-x_i}$$

SOFTMAX units have outputs on the range $(0, 1)$. The activation function is a normalized exponential function, thus the outputs of all units in a SOFTMAX group sum to 1.

$$y_i = \frac{e^{x_i}}{1.0 + \sum_j e^{x_j}}$$

OUTPUT units have an error measure function. This function determines what contribution the unit has to the total network error. This is the value that is back-propagated through the network. Using the notation y_i as the output of unit i and d_i as its target output:

SUMSQUARE units contribute one half the square of the difference between their target and output.

$$E_i = \frac{1}{2}(d_i - y_i)^2$$

ADAPTIVESUMSQUARE units use the same model for the error as SUMSQUARE (i.e. they assume that all errors come from a gaussian distribution with mean 0), but they add a variance measure that also changes during training. This leads to the more complex function:

$$E_i = \ln\left(\sqrt{2\pi}\sigma\right) + \frac{1}{2}(d_i - y_i)^2$$

where σ is is the adaptive variance of the errors.

CROSSENTROPY units interpret their outputs as a probabilistic decision. The appropriate error measure for this case is the cross-entropy between the actual and desired probability distributions of the unit's state:

$$E_i = d_i \ln\left(\frac{1}{y_i}\right) + (1 - d_i)\ln\left(\frac{1}{1 - y_i}\right)$$

The final category of unit types is a catch all. At the time of writing, only two types fall into this category:

IPI units are **I**mmediately **P**ost **I**nput. Since derivatives do not have to be backpropagated into INPUT units, a processing speedup can be obtained by flagging those units that only have incoming connections from INPUT units.

RBF units have a combination of the two types DISTANCE and NEGEXPONENTIAL. This is simply a convenience type, it introduces no new behaviour.

C A Sample Script File

The **Xerion** script shown in Figure 20 illustrates the use of many of the commands and concepts introduced in Part II of the Users Guide. The script takes the name of a unit, and prints the names of all its incoming links, followed by their respective weights. It can be executed by typing `showLinks`, in which case it will prompt for the name of the unit:

```
xerion-> showLinks
Enter the name of the unit:  (Bias) Output.7
name = "Bias -> Output.7"
weight = 0.292484
name = "Hidden.0 -> Output.7"
weight = -0.282742
name = "Hidden.1 -> Output.7"
weight = -0.0930817
name = "Hidden.2 -> Output.7"
weight = 0.144974
xerion->
```

You can also execute the script by typing `showLinks Output.7` directly, in which case it will not prompt for the name of the unit. In this section, we will walk through the script explaining what each command does.

The first thing to notice is line 0[5]:

[5] All lines from the shell script are preceded by their line number *for reference purposes only*. The numbers do not really occur in the file.

```
0 #!xerion
1
2 unset unset-error
3 if same "$unit" "" ; then
4     var String unit
5     set unit = "Bias"
6 fi
7 set unset-error
8
9 var String usage
10 set usage = "usage: ${tokv[0]} [unit]"
11
12 if same $tokc 1 ; then
13     prompt unit "Enter name of unit: ($unit) "
14 elif same $tokc 2 ; then
15     set unit = ${tokv[1]}
16 else
17     echo "$usage"
18     exit
19 fi
20
21 if unitLongName $unit > /dev/null ; then
22     show $(unitLongName $unit).incomingLink | egrep '(weight =)|(name =)'
23 else
24     ${tokv[0]}
25 fi
26
27 exit
```

Figure 20: A script file for showing a unit's incoming links.

```
0 #!xerion
```

All **Xerion** scripts *must* have this as their first line. If they do not, then **Xerion** will not recognize them as a script.

On line 2 we use the `unset` command to ensure that the variable `unset-error` is false. This allows us to substitute variables that have not been created. Lines 3 through 6 are used to create and initialize the variable that holds the name of the unit. The value stored in this variable will be saved between invocations of the script.

```
3 if same "$unit" "" ; then
4     var String unit
5     set unit = "Bias"
6 fi
```

First we check if the variable is the same as an empty string. This will be true if the variable is empty, or more importantly, if it has not been created yet. If the string is empty, we create it (xerion allows you to recreate a variable, so long as the type is not changed). We then set it to a default value (`Bias`).

Once this is done, we reset `unset-error` to `true` on line 7.

Next, on lines 9 and 10, we create a String variable that will contain a message describing how to use the script. It will be printed in case of error. The string uses `tokv[0]` (the first argument when the script was called, which is the name of the script file) as part of the message.

```
 9 var String usage
10 set usage = "usage: ${tokv[0]} [unit]"
```

Lines 12 through 19 are used to determine what unit the script will act on. The name of the unit is stored in the previously created variable `unit` and saved between invocations of the script.

```
12 if same $tokc 1 ; then
13     prompt unit "Enter name of unit: ($unit) "
14 elif same $tokc 2 ; then
15     set unit = ${tokv[1]}
16 else
17     echo "$usage"
18     exit
19 fi
```

If `tokc` is `1`, then the script was called with no arguments; we must prompt for the name of the unit and store it in `unit` (line 13). If the user simply hits return at the prompt, then `unit` will be unchanged (and its previous value will be used as the name). If `tokc` is 2, then the second argument (`tokv[1]`) was the name of the unit; store it in `unit` (line 15). Any other number of arguments is an error; print the usage string and exit the shell script (lines 17 and 18).

The next section of code does the actual work. First it sees if the name given by the user is a valid name. This is done on line 21. The command `unitLongName` is used as the `if` test. If the name given by the user is valid, the command will complete successfully, and the command on line 22 will be done. If not, the command after the else will be done. The output of the `unitLongName` command is redirected onto `/dev/null`, since we are only interrested in whether or not the command is successful, not what it prints.

```
21 if unitLongName $unit > /dev/null ; then
22     show $(unitLongName $unit).incomingLink | egrep '(weight =)|(name =)'
23 else
24     ${tokv[0]}
25 fi
```

The code on line 22 does the actual printing of the link values. To do this it uses command substitution, nested parameter substitution, and redirection. First the parameter substitution is done to change the string `$(unitLongName $unit)` to `$(unitLongName Output.7)`. Then, command substitution is done to change it to the output of that command: `currentNet.group[3].unit[7]`. Once that is done the line has become:

```
show currentNet.group[3].unit[7].incomingLink | egrep '(weight =)|(name =)'
```

The `show` command is executed and its output is redirected to the unix command `egrep`. Egrep takes the output and filters out all lines except those containing the string `weight =` or `name =`. This produces the output shown at the start of this section.

The only remaining part of the script is line 24. This line is only executed if the name of the unit was invalid (i.e. the unit doesn't exist). As mentioned previously, `tokv[0]` contains the name of the shell script. Therefore, after parameter substitution line 24 contains the string `showLinks`, which will recursively call the shell script again, prompting for a new unit name. Once that script exits, this script will continue to line 27 and exit as well, returning control to the user.