A Users Guide for
The **Xerion** Neural Network Simulator
Version 4.0

Drew van Camp
Department of Computer Science
University of Toronto

October 24, 1994

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.

Contents

0 Introduction **2**

1 Network Structure **2**

2 Running the Simulator **3**

3 A Simple Network — The XOR Network **4**
- 3.1 Building the Network 4
  - 3.1.1 Working with Nets 4
  - 3.1.2 Working with Groups 6
  - 3.1.3 Working with Units 7
  - 3.1.4 Working with Links 7
- 3.2 Creating an Example Set 8
- 3.3 Activating the Network 9
- 3.4 Training the Network 10
- 3.5 Saving and Restoring Weights 11
- 3.6 Retraining the Net 12

4 A More Detailed Network — The Encoder Network **13**
- 4.1 Building the Network 13
- 4.2 Creating an Example Set 14
- 4.3 The uts_info command 15
- 4.4 Training the Network 15
- 4.5 Types of Groups 16
- 4.6 Complexity Cost 18

5 Minimization Methods **20**
- 5.1 Steepest Descent 20
- 5.2 Momentum Descent 21
- 5.3 Conjugate Gradient 21
- 5.4 Other Methods 22

6 Examining Data Structures **22**
- 6.1 Viewing Structures 22
- 6.2 Setting Structure Fields 24
- 6.3 The "uts_bound" Command 24

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 networks, activating them, training them, etc. It also has a graphical interface for displaying network activations on specified examples, and for examining connection weights between units.

The command interface for **Xerion** is written using **Tcl** (Tool Command Language) and **Tk** both of which were developed by John Ousterhout at the University of California at Berkeley. Tcl is a simple, embeddable, scripting language for controlling and extending applications. It provides generic programming facilities, such as variables and loops and procedures that are useful for a variety of applications. **Tk** is an extension to **Tcl** that implements a toolkit for the X Window System. The graphical interface for **Xerion** is written using **Tk**.

This document is an overview of the Back Propagation simulator written with the **Xerion** Neural Network libraries. It contains a tutorial on how to build networks, build data sets for training, and train using this data. As well it describes the different minimization techniques available for training and shows how to examine low level data structures.

1 Network Structure

Networks in **Xerion** are composed of sets of objects or *primitives*. Building a network requires creating these objects and connecting them together in the proper structure as shown in Figure 1.

The first top-level objects are **Nets**. Nets are defined by the units they contain and how they are connected. They are container objects.

Inside nets are **Groups**. 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".

Inside groups are **Units**. A unit is an object that has a total input, an output and a function for mapping the total 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, error derivatives for the weights, and a function for calculating these derivatives. A link can connect two, and only two, units together.

Other top-level objects are **ExampleSets**. Example sets contain the data necessary to train and test a network. The data is stored internally as a list of examples, each example containing an input and a target vector.

The final type of top-level objects are **Minimizers**. They are used to train nets on example sets using any of several optimization techniques.

Xerion has compiled commands for building and directly manipulating these objects. As well, there are many utility commands that perform more complex operations on these primitives. In general, the utilities are interpreted procedures. and can easily be viewed using the **Tcl** commands **info body**, **info args**, etc.

In order to avoid naming conflicts within the simulator, all C object types have the string "**Uts_**" prepended to their names, eg. "**Uts_Net**". As well, all **Tcl** commands added by **Xerion** have the prefix "**uts_**". Commands added specifically by and for the back propagation simulator have the prefix "**bp_**".

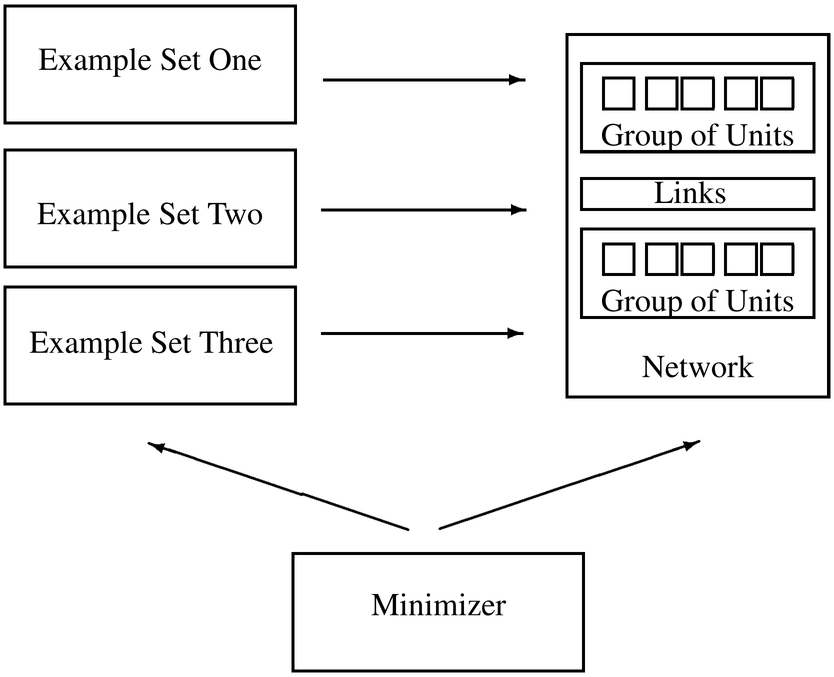

Figure 1: Typically, a simulation run will have a single network, multiple example sets, and a single minimizer. Examples are retrieved from one of the example sets and presented to the network. Minimizers train networks using a particular example set. The network can then be tested or validated using one of the other example sets.

2 Running the Simulator

The only thing that you have to configure in your shell environment before running **Xerion** is your path variable. Just make sure that the directory where the programs **bp_wish** and **bp_sh** are installed is somewhere in the path (this may already be the case for you if **Xerion** was installed in a standard place).

Once your environment is configured, the back propagation simulator can be started from a **UNIX** prompt by typing one of the commands:

```
unix> bp_sh
unix> bp_wish
```

If you run **bp_sh** there is no graphical interface to the simulator. It only takes commands from the prompt. This program is useful for running batch jobs or scripts. If you are running **bp_wish** there is both a command line interface and a graphical interface. This tutorial covers only the command line interface for both programs.[1] **Xerion** has added several commands to the standard **Tcl** interface. These commands are for building networks, for optimizing mathematical functions, as well as for doing tasks that come up frequently when dealing with neural networks. All commands have online help that can be obtained with the **help** command provided by **TclX**. This command allows you to look up help pages on built-in commands using a multilevel tree of subjects. There is help for standard **Tcl** commands as well as for

[1] The graphical interface should be self explanatory once you understand the command line interface.

Xerion commands. Commands added by **Xerion** are stored under the subject **uts/** (University of Toronto Simulator). For more help on the help mechanism try:

```
bp-> help help
```

Finally, if you wish to quit the simulator, just type the command **exit**.

> Although this tutorial assumes that you are typing all of the commands at the prompt, they can just as easily be saved to a file read in using the **Tcl** command **source**.
>
> ```
> bp-> source file.tcl
> ```

3 A Simple Network — The XOR Network

In this section we will introduce you to the commands used by **Xerion** to build and train networks using the simple example of an XOR network (a network that learns the exclusive-or problem). It will have a single hidden layer of 2 units.

3.1 Building the Network

The most common architecture for back propagation networks is a single hidden layer network with fully connected layers. To simplify building this type of network, **Xerion** provides a utility procedure that constructs the entire network in one step. This procedure is called **uts_simpleNet**. The procedure takes four arguments: the name for the net; the number of input units; the number of hidden units; and the number of output units. It returns the name of the newly created net. To build the XOR network, you can use the command:

```
bp-> uts_simpleNet Xor2 2 2 1
Xor2
```

> The name for the net can contain any character except ".". Names of nets, groups, units, etc. must be unique. If they are not unique an error will be generated. The names may contain spaces; however, if they do contain spaces (or special characters) they must be quoted using "" or **{}**.

3.1.1 Working with Nets

Uts_simpleNet not only creates a net called Xor2, it creates a **Tcl** command with the same name. This command can be used to examine or manipulate the net. In our example **uts_simpleNet** created a command called **Xor2** that can be used for the XOR network. The command **Xor2** takes as its first argument any of several subcommands. These subcommands are all listed in the help page for **uts_net**:

```
bp-> help uts/network/primitives/uts_net
```

For now we only need to deal with a small subset of the subcommands.

One of the most important network subcommands is **configure**. The configure command allows you to view and modify simple fields (options) of the network. The command has three forms. Executed with no arguments, it returns a list describing all of its available options. Each element of the list is itself a list. Each sublist is composed of the option's name, the default value for the option, the current value for the option, and a short description of the option.

```
bp-> Xor2 configure
{-error 0.0 0.0 {total network error}} {-cost 0.0 0.0 {total network complexity
   cost}} {-currentEpoch 0 0 {current training epoch}} {-batchSize 0 0 {number
   of examples to process in one batch}}
```

In this case, configure will accept any of the options: **-error**, **-cost**, **-currentEpoch**, or **-batchSize**.

In the second form of the command, one of the available options is given to the command. In this form, a list describing just that option is returned.

```
bp-> Xor2 configure -error
-error 0.0 0.0 {total network error}
```

The returned list is identical to the corresponding sublist for the value returned if no option is specified.

In the final form of the command, one or more of the options is specified paired with a value to modify the option to have that value. In this case, the command returns an empty string.

```
bp-> Xor2 configure -error 1.0
bp-> Xor2 configure -error
-error 0.0 1.0 {total network error}
```

Another important subcommand is **delete**. This subcommand destroys the network, all of its groups and units, and removes the associated commands.

A Short Digression on Tcl Using the configure command, you can quickly see the options of the net; however, there is often extraneous data being printed. If you only want to see the current value of an option (and not the name, default value and help message), you must combine **configure** with the **Tcl** command **lindex**. **Lindex** returns the item with index i from a list:

```
bp-> lindex {a b c d} 0
a
bp-> lindex {a b c d} 2
c
```

To extract the value from the **configure** list, do:

```
bp-> lindex [Xor2 configure -error] 2
0.0
```

As well as the **lindex** command, the other construct we have used here is square brackets, which are used to invoke command substitution. If an open bracket appears in an argument, then everything from the open bracket up to the matching close bracket is treated as a command and executed recursively by the **Tcl** interpreter. The result of the command is then substituted into the argument in place of the bracketed string.

As well, you can assign the value of the command to a variable using the **set** command.

```
bp-> set a [lindex [Xor2 configure -error] 2]
0.0
```

This command will create a new string variable called **a** and set it to the result of the **lindex** command. You can later substitute the value of the variable using a dollar sign.

```
bp-> set b $a
0.0
```

will result in the variable **b** containing the same value as **a**.

3.1.2 Working with Groups

As stated in Section 1, networks are composed of groups. To determine the names of the groups in a net use the net's **groups** subcommand. The form of this command is:

```
<name> groups ?!?  <mask>
```

<Name> is the name of the net whose groups you want to list (in our case **Xor2**), and **<mask>** is a list of type masks describing the groups you want to list. **<Mask>** can contain any of the strings **INPUT**, **OUTPUT** , **BIAS** or **ALL**.

INPUT units are those that have their output value clamped when an example is presented to a network.

OUTPUT units are given a target value when an example is presented to a network. They contribute error directly to a net when activated.

BIAS units always have an output of 1.

ALL is a shortcut for specifying all units in the network.

> **<Mask>** can also contain any of the group type masks described in section 4.5 (however the above masks are the ones most commonly used in this context).
> There is no special type mask for hidden units, since they are really just units with none of the other types set.

The list returned by the **groups** command will contain all groups in **<name>** that have any of the types in **<mask>**, or all of the groups in **<name>** if **<mask>** is **ALL**. The optional **!** can be used to negate the test. If present, the command returns a list of all the groups in the net *not* containing any of the types in **<mask>**.

To list all of the groups in the XOR network use the command:

```
bp-> Xor2 groups ALL
Xor2.Bias Xor2.Input Xor2.Hidden Xor2.Output
```

To list just input and output groups, try:

```
bp-> Xor2 groups {INPUT OUTPUT}
Xor2.Input Xor2.Output
```

To list all the groups except the bias:

```
bp-> Xor2 groups !  BIAS
Xor2.Input Xor2.Hidden Xor2.Output
```

Now that you know the names of the groups you can manipulate them. As with the net, the command that creates a group also creates a command with the same name as the group. This command is used to manipulate the group. The group command takes subcommands in the same manner as the net command. For a complete description of this command see the online help for **uts_group**.

As an introduction to the command, see how many units are in each group using the **size** subcommand:

```
bp-> Xor2.Input size
2
bp-> Xor2.Hidden size
2
bp-> Xor2.Output size
1
```

To see the name of a specific unit in a group, use the **unit** subcommand. This command requires that you give the index for the unit that you want. The indices of units start at 0 and go to one less than the number of units in the group.

```
bp-> Xor2.Input unit 0
Xor2.Input.0
bp-> Xor2.Input unit 1
Xor2.Input.1
bp-> Xor2.Input unit 2
Error:  Command error:  index greater than number of units.
```

Groups do not support the **configure** subcommand (because they have no easily configurable parameters).

3.1.3 Working with Units

In the previous section we saw how to obtain the names of the units in a group. As with groups (and nets), each unit has a command associated with it, and the name of the command is the same as that of the unit. For example, in our network there are commands **Xor2.Input.0**, **Xor2.Hidden.0**, etc. For a complete description of these commands see the online help for **uts_unit**.

As with networks, the most important subcommand for units is **configure**. Use the following commands to examine and change the state of the single output unit in the XOR network.

```
bp-> Xor2.Output.0 configure
{-inputDeriv 0.0 0.0 {derivative of total input}} {-output 0.0 0.0 {unit
   output}} {-outputDeriv 0.0 0.0 {derivative of unit output}} {-target 0.0
   0.0 {unit target (only has meaning for OUTPUT units)}} {-totalInput 0.0
   0.0 {total input to unit}}
bp-> Xor2.Output.0 configure -output
-output 0.0 0.0 {unit output}
bp-> Xor2.Output.0 configure -output 1.0
bp-> Xor2.Output.0 configure -output
-output 0.0 1.0 {unit output}
```

3.1.4 Working with Links

Links connect units together. They are unidirectional; that is, they go from a pre-unit to a post-unit. Order is important. The connection from unit a to unit b is *not* the same as the connection from unit b to unit a.

Links are specified using the two units they connect: the pre-unit and the post-unit. For example, the link connecting the first input unit to the second hidden unit of our network would be specified by the two strings **Xor2.Input.0** and **Xor2.Hidden.1**.

To find out what units are connected to a specific unit you can use the unit's subcommands **numPreUnits** and **preUnit**:

```
bp-> Xor2.Output.0 numPreUnits
3
bp-> Xor2.Output.0 preUnit 0
Xor2.Bias.0
```

There are similar commands to determine the units that a given unit is connected to: **numPostUnits** and **postUnit**. Once you have determined whether a connection exists, you can have a look at the link itself. To do this, use the command **uts_link**.

Each link does not have an individual command associated with it. There is only one command that acts on all links: **uts_link**. You must pass this command the names of the two units the link connects. The reason there is not an individual command associated with each link is that there can be a *huge* number of links in even a medium sized network. The overhead of creating a command for each one would be prohibitive.

Have a look at the fields in the link using the **configure** subcommand, and try changing some of their values.

```
bp-> uts_link configure Xor2.Input.0 Xor2.Hidden.1
{-weight 0.0 -0.147821 {connection weight}} {-deriv 0.0 0.0 {derivative of
   connection weight}} {-deltaWeight 0.0 0.0 {last change in weight}}
bp-> uts_link configure Xor2.Input.0 Xor2.Hidden.1 -weight
-weight 0.0 -0.147821 {connection weight}
bp-> uts_link configure Xor2.Input.0 Xor2.Hidden.1 -weight 0.3
bp-> uts_link configure Xor2.Input.0 Xor2.Hidden.1 -weight
-weight 0.0 0.3 {connection weight}
```

3.2 Creating an Example Set

Now that we have created a network, we must create and store some patterns to train it on. Patterns in **Xerion** are stored in objects called **ExampleSets**. You can have as many example sets in the simulator as you want. When you train a network, you must specify which set of examples you want to train it on. Similarly, when testing, you must specify the set to test it on.

To create a structure that will hold a set of examples use the command **uts_exampleSet**. This command only requires you to specify the name for the new set:

```
bp-> uts_exampleSet Training
Training
```

As with networks, the command returns the name of the example set and creates a command with the same name. See the help page for a complete list of the options available to the command.

When you create an example set it initially contains no examples. To see how many examples are in it at any time use the command:

```
bp-> Training size
0
```

There are many different ways of adding examples into the example set (since example files tend to all have different formats), most of which are procedures. For our case the easiest to use is **uts_addExample**. This command creates a new example and adds it to the end of the list of examples stored in the example set.

The command takes four arguments: the name of the example set to add to; a "tag" name to give the new example (this is normally an empty string); a list of input values for the example; and a list of target values for the example. The lists of values are composed of floating point numbers. The command returns the index of the new example in the set.

Our example set has four examples:

```
bp-> uts_addExample Training {} {0 0} {0}
0
bp-> uts_addExample Training {} {0 1} {1}
1
bp-> uts_addExample Training {} {1 0} {1}
2
bp-> uts_addExample Training {} {1 1} {0}
3
bp-> Training size
4
```

3.3 Activating the Network

Now that we have a network and a set of examples, we can see how the network performs on them. To do this we have to follow a two step process. First we have to present the example to the network. Presenting an example entails clamping the outputs of all units in groups of type **INPUT** with the input vector from the example, and setting the targets of all units in groups of type **OUTPUT** with the target vector from the example. Once the example has been presented, we then activate the net, mapping the inputs to output values for all groups and calculating the error on the example.

The commands that do this are: **uts_presentExample**, and **uts_activateNet**. **Uts_presentExample** takes three arguments: the name of the network to present the example to; the name of the example set to retrieve the example from, and the index of the desired example. **Uts_activateNet** takes only one argument: the name of the net. Both commands return an empty string.

```
bp-> uts_presentExample Xor2 Training 0
bp-> uts_activateNet Xor2
```

Now that the network has been activated you would probably like to see how well it performed in generating an output similar to its target. We can use the unit **configure** command from section 3.1.3 to compare the single output unit's target and actual output:

```
bp-> Xor2.Output.0 configure -target
-target 0.0 0.0 {unit target (only has meaning for OUTPUT units)}
bp-> Xor2.Output.0 configure -output
-output 0.0 0.511589 {unit output}
```

This shows that the output unit has a target value of 0.0 and an actual output of 0.511589 (to six significant digits). Also of interest is the network error:

```
bp-> Xor2 configure -error
-error 0.0 0.130862 {total network error}
```

The value stored in the network error is the sum-square error (one half the sum of the squared difference of the output and the target) Each time the network is activated the error is incremented by the sum-square error on the current example. If you want to reset the error, use:

```
bp-> Xor2 configure -error 0.0
```

You may want to repeat presenting examples, activating the network, and examining the network state for each of the examples in the example set.

3.4 Training the Network

As you've seen, the XOR network doesn't initially perform very well on the patterns. This is because the weights are initialized to random values. To train the weights we need to create a "minimizer". The minimizer knows how to optimize functions, and in our case how to minimize the network error. To create the minimizer use the command **Bp_NetMinimizer**[2]. The command requires that you give a name for the minimizer. It creates an object, as well as a command with the same name, that can be used to train the network:

```
bp-> Bp_NetMinimizer mz
mz
```

The first thing we have to do is tell the minimizer what net we want to train and what example set we want to train on. To do this we use the **configure** subcommand.

```
bp-> mz configure -net Xor2 -exampleSet Training
```

By default, the minimizer will use a fairly sophisticated conjugate gradient method for optimizing the weights. As well, there are *many* more options available to the minimizer's **configure** command. This is explained in a little more detail in Section 5.

The minimizer is now ready to train the net. The only thing we have to tell it is how many iterations (weight updates) we want it to run for:

```
bp-> mz run -iterations 25
```

The output of this command is shown in Figure 2 (The actual numbers in your output may be different). The number of iterations was chosen arbitrarily. You can try any number you want or even repeat the command to train for twenty more iterations. Each line of the output corresponds to one weight update. The columns in the output are: the current iteration (how many times the weights have been changed); the number of times the network has been forward propagated through all the examples[3]; the number of times the network has been backward propagated through all the examples[4]; the current sum-square error, and the length of the gradient vector.

Once the algorithm terminates. It prints a short report describing the reason it stopped and the final state of the network. The terms in the report are self explanatory except for **d** which is the final slope of the function, i.e. the dot product of the gradient and the search direction. You can obtain a more concise version of this at any time using the minimizer **status** command.

```
bp-> mz status
{iter 19} {nFE 160} {nGE 46} {f 0.333358} {|g| 4.83018e-05} {d -6.94654e-06}
   {code 1} {codeString {algorithm has converged.}}
```

[2] Yes, this command is capitalized. Its name may be changed in future releases of **Xerion**.
[3] This term is called nFE because it is the number of times the function was evaluated.
[4] The term nGE means the number of times the gradient was evaluated.

```
bp-> mz run -iterations 25
iter:    1  nFE:    6  nGE:    2  f: 0.5          |g|: 0.00371685
iter:    2  nFE:   11  nGE:    3  f: 0.499985     |g|: 0.00184774
iter:    3  nFE:   16  nGE:    4  f: 0.499981     |g|: 0.000342592
iter:    4  nFE:   46  nGE:   14  f: 0.488898     |g|: 0.0315635
iter:    5  nFE:   53  nGE:   15  f: 0.488883     |g|: 0.0322995
iter:    6  nFE:   68  nGE:   20  f: 0.486445     |g|: 0.0177396
iter:    7  nFE:   74  nGE:   22  f: 0.481876     |g|: 0.0501779
iter:    8  nFE:   83  nGE:   25  f: 0.433226     |g|: 0.0702656
iter:    9  nFE:   86  nGE:   26  f: 0.398858     |g|: 0.128371
iter:   10  nFE:   89  nGE:   27  f: 0.355283     |g|: 0.0632105
iter:   11  nFE:   95  nGE:   29  f: 0.338093     |g|: 0.00697103
iter:   12  nFE:  100  nGE:   30  f: 0.336639     |g|: 0.0229863
iter:   13  nFE:  105  nGE:   31  f: 0.334467     |g|: 0.00124624
iter:   14  nFE:  112  nGE:   32  f: 0.334435     |g|: 0.00308472
iter:   15  nFE:  124  nGE:   36  f: 0.333537     |g|: 0.00038147
iter:   16  nFE:  131  nGE:   37  f: 0.333529     |g|: 0.00121881
iter:   17  nFE:  143  nGE:   41  f: 0.333388     |g|: 0.000111121
iter:   18  nFE:  148  nGE:   42  f: 0.333385     |g|: 0.00099017
iter:   19  nFE:  160  nGE:   46  f: 0.333358     |g|: 4.83018e-05
iteration 19: algorithm has converged. (code 1).
      f: 0.333358                        nFE:   160
    |g|: 4.83018e-05   d: -6.94654e-06   nGE:    46
```

Figure 2: Network minimizer training results.

Now that the network has been trained, you can try presenting the examples to it and compare the network output to the target. It may, at this point, be helpful to write a simple procedure that will do this for you. The procedure shown in Figure 3 is such a one. The new **Tcl** commands used here are: **proc**, **loop**, and **puts**. There is online help for each one of these.

To create the procedure, edit a file called **compare.tcl** (or any other name you want), and enter the procedure as it appears in the figure into the file. Then source the file:

```
bp-> source compare.tcl
```

You can now execute **compare** just like any other command:

```
bp-> compare
target:  0.0, output:  0.00226229
target:  1.0, output:  0.666593
target:  1.0, output:  0.666593
target:  0.0, output:  0.666626
```

3.5 Saving and Restoring Weights

If you're not happy with the network's performance on the examples, you may want to try training it again but with different initial random weights. Before retraining the net you can save the weights. To do this use the command **uts_saveWeights**. This command requires the name of the network whose weights you wish to save, and the name of a file to save the weights to.

```
proc compare {} {
  loop i 0 4 {
    uts_presentExample Xor2 Training $i
    uts_activateNet Xor2
    set output [lindex [Xor2.Output.0 configure -output] 2]
    set target [lindex [Xor2.Output.0 configure -target] 2]
    puts "target: $target, output: $output"
  }
}
```

Figure 3: Procedure for comparing target and actual output values.

```
bp-> uts_saveWeights Xor2 xor2.wt
```

If you have a look at the file generated by the command, you will see that it contains **Tcl** commands that configure the weights of all the links in the net. To reload the weights, you need only source the file:

```
bp-> source xor2.wt
```

You can also save the weights in a binary format that is much more compact and has full precision. To do this, use the command:

```
bp-> uts_dumpWeights Xor2 xor2.wt
```

To reload the weights, use **uts_loadWeights**. This command requires (besides the name of the file), the name of the net to load the weights into. The structure of the net you load the weights into must be *exactly* the same as that of the net that dumped the weights.

```
bp-> uts_loadWeights Xor2 xor2.wt
```

3.6 Retraining the Net

Now that the weights are saved, you can retrain the net from scratch. First, you must re-randomize the weights. The **Xerion** command for this is **uts_randomizeNet**:

```
bp-> uts_randomizeNet Xor2
```

This command will re-randomize all the weights in the network to numbers on the range -0.3 to $+0.3$. You can specify an alternative range on the command line if you want:

```
bp-> uts_randomizeNet Xor2 -1.0 2.0
```

Once the weights are randomized, you must tell the minimizer to discard any saved information it has about the network state:

```
bp-> mz reset
```

Now you can retrain the network using the command **mz run**. Repeat this as many times as you like, comparing the different results you get with different initial weights.

4 A More Detailed Network — The Encoder Network

In this section we will go into greater detail with the **Xerion** commands by building a more complicated network than the XOR network. This network is an auto-encoder; a network that is trained to generate as its output, its input. To do this it must encode the input pattern in an internal state.

The auto-encoder we will build has eight input units, three hidden units, and eight output units. The example set we will present it consists of eight vectors of eight binary values where only one of the values is on (equal to one). The network will have to learn to encode the eight patterns in the activation levels of the three hidden units.

4.1 Building the Network

To build the auto-encoder we will not use the procedure **uts_simpleNet** (even though it would work for this case). Instead we will build the network using lower level commands. First we will create an empty net. Next we will add groups to the net, then we will connect the groups together and initialize the weights.

To create an empty net use the command **uts_basicNet**. This command will create a net with only one group: a bias group with one unit. As well, it will do some initialization that is more detailed than we need to go into at this point.

```
bp-> uts_basicNet Encoder
Encoder
```

As with the **uts_simpleNet** command, you must specify a unique name for the net. The command creates the net and associated network command. Once the net is created we need to add the groups to it.

The order in which you create the groups is important. When the net is activated, the groups activate in that order. If you want to change the order you can use the net's **groupOrder** subcommand.

Groups are created using the **uts_group** command. The command determines which net the group is to be added to from the name you give the group. If you want to add a group to the net with name **Encoder**, you must give the group a name that starts with the prefix "**Encoder.**", for example **Encoder.Input**. You must also specify the initial number of units to put in the group. You can change this number at a later time using the **uts_unit** command. Finally, you must specify the "type" of the group (which may be an empty string). Types are the same ones introduced in Section 3.1.1. The groups we need for the Encoder network are:

```
bp-> uts_group Encoder.Input 8 {INPUT}
Encoder.Input
bp-> uts_group Encoder.Hidden 3 {}
Encoder.Hidden
bp-> uts_group Encoder.Output 8 {OUTPUT}
Encoder.Output
```

The net now contains all the units it needs; however, none of the units are connected to one another. Fully connecting groups together is fairly straight forward. Simply use the command **uts_connectGroups** passing in the names of the two groups you wish to connect.

```
bp-> uts_connectGroups Encoder.Input Encoder.Hidden
bp-> uts_connectGroups Encoder.Hidden Encoder.Output
```

```
{1 0 0 0 0 0 0 0} {1 0 0 0 0 0 0 0}
{0 1 0 0 0 0 0 0} {0 1 0 0 0 0 0 0}
{0 0 1 0 0 0 0 0} {0 0 1 0 0 0 0 0}
{0 0 0 1 0 0 0 0} {0 0 0 1 0 0 0 0}
{0 0 0 0 1 0 0 0} {0 0 0 0 1 0 0 0}
{0 0 0 0 0 1 0 0} {0 0 0 0 0 1 0 0}
{0 0 0 0 0 0 1 0} {0 0 0 0 0 0 1 0}
{0 0 0 0 0 0 0 1} {0 0 0 0 0 0 0 1}
```

Figure 4: Data file for the encoder network.

Don't forget that there is an automatically created bias group in the net called **<name>.Bias**. You must connect this unit to all groups in the network, except for the input group.

```
bp-> uts_connectGroups Encoder.Bias Encoder.Hidden
bp-> uts_connectGroups Encoder.Bias Encoder.Output
```

If you want to connect single units together, you can use the **uts_link** command. When connections are made their weights are set to zero. The most common way to initialize the weights is to randomize the entire net.

```
bp-> uts_randomizeNet Encoder
```

The encoder network is now completely built.

4.2 Creating an Example Set

Creating the example set for this problem is the same as creating the example set for the XOR network (except we're going to use a different name for the set):

```
bp-> uts_exampleSet EncoderData
EncoderData
```

For the XOR data we entered each example separately. In this case we are going to put all the data in a file, then load the file into the example set in one step. The basic format of the example files is one line corresponding to one example. Each line should contain two lists. The first list contains the input patterns for the example (floating point numbers), the second list contains the target patterns (also floating point numbers).[5] Figure 4 shows the example file for the encoder network.

To load the data file, use the command **uts_loadExamples**. The command requires you to specify the name of the example set to load the data into, as well as the name of the file containing the raw data.

```
bp-> uts_loadExamples EncoderData encoder.ex
```

If you now check the size of the example set, you will see that it is eight.

```
bp-> EncoderData size
8
```

[5] The format of the file is actually somewhat more flexible than this. For a complete description of the format see the help page for **uts_loadExamples**.

4.3 The uts_info command

Xerion allows you to have multiple nets, multiple example sets, and multiple minimizers all loaded into the simulator at once. This can sometimes become confusing. To help you keep track of all the simulator objects **Xerion** supplies a command called **uts_info** for listing the currently existing objects. If you want to list all the nets in the simulator, use the command:

```
bp-> uts_info nets
Encoder Xor2
```

If you want to list all the groups, use:

```
bp-> uts_info groups
Xor2.Bias Xor2.Hidden Xor2.Input Encoder.Bias Xor2.Output Encoder.Hidden
   Encoder.Input Encoder.Output
```

Similar commands exist for example sets, units, links, etc.

Uts_info allows you to narrow down the objects listed using glob pattern matching. If you specify a pattern on the command line, it will return a list consisting of all the objects of the type requested whose names match the pattern.[6] For example, to list all the groups whose names end in the string **.Input**, use:

```
bp-> uts_info groups *.Input
Xor2.Input Encoder.Input
```

4.4 Training the Network

To train the network we can reuse the minimizer we created in Section 3.4. To do this we must tell it that we wish to train a different network using a different example set. As before, we use the configure command for this.

```
bp-> mz configure -net Encoder -exampleSet EncoderData
```

To train the network we can, once again, use the same commands as in Section 3.4. We only need to change the number of iterations. We can also use one of the minimizer's other configure options to make life a bit easier: **-repetitionCount**. This is an integer value that controls how frequently reports are printed. A report is printed after every **repetitionCount** weight updates. For this network, a report interval of 10 is appropriate

```
bp-> mz configure -repetitionCount 10
```

We're now ready to train:

```
bp-> mz reset
bp-> mz run -iterations 100
```

To help you evaluate the network's performance, you might want to modify the **compare** procedure from Figure 3 so that it prints the outputs and targets of all eight output units in **Encoder**. This is left as an exercise.

[6] Pattern matching is done using the same format used by the **Tcl string match** command.

4.5 Types of Groups

When you create a group you specify a type mask as an argument to the command. This type mask controls certain behaviour aspects of the group. For example units in **BIAS** groups always have an output of 1.0. So far the only types of groups we have discussed are: **INPUT**, **OUTPUT**, and **BIAS** (see Section 3.1.1). These types are used by most of the simulators built with the **Xerion** libraries. However, the back propagation simulator adds several new types of groups. These new types are used to control the activation behaviour of the units.

A group's type is composed of a set of masks. Several types can be listed together for one group, for example a group could have the type **{DISTANCE DPROD}**

The activation behaviour falls into three categories: input combining function; transfer function; and error measures.

The input combining function of a unit takes the outputs of all the unit's pre-units and combines them together using the weights of all its incoming links. This becomes the unit's total input. The back propagation simulator contains three types of input combining functions:

DISTANCE units calculate their total input as the square of 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.

All units in a network except those in **INPUT** and **BIAS** groups must have an input combining function. If one isn't specified, it defaults to **DPROD**.

Transfer functions control how a unit maps 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 transfer functions implemented in the back propagation simulator:

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 the identity transfer function. Their output is the same as their input.

$$y_i = x_i$$

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

$$y_i = \tanh(x_i) = \frac{e_i^x - e^{-x_i}}{e_i^x + e^{-x_i}}$$

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 $(0, \infty)$.

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

As with the input combining function, all units in a network except those in **INPUT** and **BIAS** groups must have a transfer function. If one isn't specified, it defaults to **LOGISTIC**.

In the same class as the other transfer functions, but somewhat different, is the type **NORMALIZED**. **NORMALIZED** groups take the outputs of all units in the group, and normalize them by dividing each output by the sum of all the outputs in the group. **NORMALIZED** can be used in conjunction with any of the above transfer functions; however, it really only makes sense for functions that produce strictly non-negative outputs.

OUTPUT units have an error measure function. This function determines what contribution the unit has to the total network error. This determines what 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:

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

where σ is is the adaptive variance of the errors for the group.

CROSSENTROPY units interpret their outputs as a probabilistic decision. If the output group is not normalized, each unit's output is viewed as a separate probabilistic decision between discrete alternatives (1 and 0). The appropriate error measure for this case is the cross-entropy between the actual and desired probability distributions of the unit's state:

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

If the group is NORMALIZED so that all the outputs sum to 1, the cross entropy error measure is:

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

NOERROR units contribute no error to the network.

Only **OUTPUT** groups can have an error measure specified. If one is not specified, it defaults to **SUMSQUARE**.

In the back propagation simulator you must use the command **bp_groupType** to see what type a unit has. This command requires you to specify the name of the group. You can also specify a type mask on the command line. If you do, the type of the group and its associated behaviour will be changed to it before returning.

Do not use the standard group subcommand **type** to change group types in the back propagation simulator. Since the simulator adds many new types, it also adds many consistency checks on their values. You *must* use **bp_groupType** to change a group's type.

To see what types each of the layers in the Encoder network have, try:

```
bp-> bp_groupType Encoder.Bias
BIAS
bp-> bp_groupType Encoder.Input
INPUT
bp-> bp_groupType Encoder.Hidden
UNKNOWN DPROD LOGISTIC
bp-> bp_groupType Encoder.Output
OUTPUT DPROD LOGISTIC SUMSQUARE
```

Sometimes the type **UNKNOWN** will turn up. It is safe to delete it from the list, or leave it in. It has no effect on the group's behaviour.

For the encoder network you may have noticed that we only want one of the output units to ever fire at a time. This is the ideal situation in which to use a soft-max group. Soft-max groups are created by setting the activation function of the group to **EXPONENTIAL** and normalizing it. As well, soft-max groups are almost exclusively used with a cross-entropy error measure. To turn the output group into a soft-max one, use the command:

```
bp-> bp_groupType Encoder.Output {OUTPUT EXPONENTIAL NORMALIZED CROSSENTROPY}
OUTPUT DPROD NORMALIZED EXPONENTIAL CROSSENTROPY
```

Now try re-randomizing the network and training again. Compare the results with those using the standard **LOGISTIC** transfer function and **SUMSQUARE** error measure.

4.6 Complexity Cost

Up till now we have only been concerned with minimizing the error of the network on the training data. However, if we optimize just the network error, there is the potential of over-fitting the data. This is analogous to over-fitting a polynomial to a set of points. It is possible, given enough degrees of freedom, to exactly pass through all the data points, however the resulting curve would not necessarily fit the underlying model. We want the model produced by the net to generalize well to the overall problem, not just the training data.

To do this, we must minimize not only the network error, but the network complexity. There are many ways of measuring the complexity of a network. One of the most common is concerned with the sizes of the weights in the network. The larger the weights are, the more complex the network is. To reduce the size of the weights, you can add a new term to the function being optimized: one half the sum of the squares of all the weights. Optimizing with this new function is often referred to as introducing "weight decay".

Networks contain objects called **costModels**. Each one of these cost models can be used to add a complexity cost into the function being optimized for the net. When a costModel is evaluated its result is accumulated into the network field **cost**. The sum of the error and the cost is the function the minimizer attempts to minimize. To list the cost models in a net use the command:

```
bp-> Encoder costModels
Encoder.costModel
```

The command **uts_basicNet** automatically creates a single cost model in the net; however, it must be explicitly activated to be used. The command **uts_simpleNet** (which calls **uts_basicNet**), does activate it.

Cost models are named in a manner identical to groups: the name of the net, followed by a ".", followed by a unique name. Each cost model is associated with a set of links. The cost computed by the model is based on these links, and these links alone. Normally, a network will contain a single cost model, and this cost model will compute its cost based on all the links in the net. To see what links are in the Encoder cost model, use the command:

```
bp-> Encoder.costModel links
```

Initially, the cost model has no links in it. To add links to the model use the cost model subcommand **addLink**.

We want to add all the links in the net into the cost model. To avoid having to explicitly enumerate all the links, we can use the procedure **uts_foreachLink**. This command is of the form:

```
uts_foreachLink <preName> <postName> <net> <body>
```

The command iterates over all links in the network with name **<net>**, setting the variables **<preName>** and **<postName>** to the names of the pre-unit and post-unit for the link. It then evaluates **<body>**. This command lets us add all the links to the cost model simply:

```
bp-> uts_foreachLink pre post Encoder {Encoder.costModel addLink $pre $post}
```

If you now try listing the links in the cost model, you will get a list of all the links in the net.

There is one other important factor to the complexity cost. You almost always want to scale the complexity cost by some value when calculating the function to optimize. That is, the function being optimized is the network error plus lambda times the complexity cost. The lambda value is often chosen by trial and error. Lambda is one of the configure parameters for the cost model. Its name is **scale** and it has a default value of zero.

```
bp-> Encoder.costModel configure
{-scale 0.0 0.0 {scale factor for cost}} {-costCommand {} {uts_sumSquareCost
   -derivs no} {command to evaluate complexity cost}} {-derivsCommand {}
   {} {command to evaluate derivatives of complexity cost}} {-costAndDerivsCommand
   {} {uts_sumSquareCost -derivs yes} {command to evaluate complexity cost
   and its derivatives}}
```

A suitable value for scale is typically something of the order 0.001.

```
bp-> Encoder.costModel configure -scale 0.001
```

If you now try to train the network you will notice that the overall error of the network is not as low as before. You will notice, however, that the weights are much smaller. You may also find that the network makes smaller errors on test data that it has not been trained on. Try printing out the weights of the network using the **uts_foreachLink** command.

```
bp-> uts_foreachLink pre post Encoder {
> puts "$pre $post [lindex [uts_link configure $pre $post -weight] 2]"
> }
```

You may have to fiddle with the **-scale** option several times to find a value that allows the net to learn the problem without the weights getting too large.

5 Minimization Methods

Up to this point we have been concerned only with how changes to the network structure or activation functions affect the network's trained performance. We haven't discussed how our training method affects the final performance. In this section we will discuss the different methods available for training networks using the **Xerion** minimizers.

All the methods in the back propagation simulator are based on the fact that you can calculate the derivatives of the network error with respect to each of the weights by back propagating the derivative of the error with respect to the network output through the net. To examine the derivatives of a particular link you can use the **uts_link configure** command with the option **-deriv**.

```
bp-> uts_link configure Encoder.Input.0 Encoder.Hidden.1 -deriv
-deriv 0.0 4.62861e-06 {derivative of connection weight}
```

The vector of all the derivatives in the network is the gradient vector for the network error. Once the gradient has been calculated, you can then make a change in the weights using it. Two decisions have to be made before you can change the weights: what direction do you want to change the weights in, and how big a change do you want to make.

5.1 Steepest Descent

The simplest way to change the weights is to take a fixed step in the negative direction of the gradient. This is called **steepest descent** optimization. Since the gradient is the direction of greatest increase for the function at a point, the negative of it is the direction of greatest decrease. If we take a small step in that direction the function should get lower. We have to take a small step because the gradient is only valid at a *point*. At a different point the gradient could be pointing in a totally different direction. A large step could put us at a point where the function is greater than at the current point.

To tell the minimizer you want to change the weights in the direction of the negative gradient you use the **configure** subcommand.

```
bp-> mz configure -direction steepest
```

Similarly, to tell it that you want to take a fixed step in that direction (i.e. a step whose length is the magnitude of the gradient times a fixed "learning rate"), use the command:

```
bp-> mz configure -step fixed
```

Next you have to set the size of the step you want to take: epsilon. This must be determined by trial and error, but a good starting value for this net is around 0.1.

```
bp-> mz configure -epsilon 0.1
```

This training method can take many iterations. You may want to configure the repetition count to be large before training or you'll end up with screens and screens of reports.

```
bp-> mz configure -repetitionCount 100
```

As well, now might be a good time to mention that you can interrupt minimization at any time by typing **<Ctrl>-C**. This will stop the minimization and leave the net with the most recently calculated weights.

Now you can re-randomize the net and train it.

```
bp-> uts_randomizeNet Encoder
bp-> mz reset
bp-> mz run -iterations 1000
```

As stated above, when we make a fixed step in the direction of the gradient we have to be careful that the step is not too large or we might increase the function value. Rather than take a fixed step and just hope that the function at the new point is better, we can do a line search in the direction of the gradient looking for the best point along it. The **Xerion** minimizers implement three types of line searches. All behave slightly differently, but the details of their differences are too complex to go into at this point. The names of the line searches are: **lsRay**, developed by Ray Watrous; **lsRudi**, developed by Rudi Mathon; and **lsTap**, developed by Tony Plate. If we use one of these instead of a fixed step, our optimizer can converge much faster.

```
bp-> mz configure -step lsRay
```

If you now re-randomize the net and train it again, you should see a significant improvement in the number of iterations required to train it.

5.2 Momentum Descent

Although the gradient can be used to calculate the direction of steepest descent at a point, it may not give a true picture of the best direction to optimize in. For example, if the error surface has a long narrow ravine, the gradient on the side points much more towards the centre of the ravine than towards the end of it. Using steepest descent to find the minimum will result in taking many small steps back and forth across the ravine, each one taking you slightly closer to the end. If you want to reach the minimum more directly, you should really go along the ravine, not across it.

An improved direction for optimization in this case can be calculated using a decaying average of the directions previously calculated or **momentum**. This will help reduce sloshing in the ravines of the error surface, and is often quite effective at speeding the rate of convergence. Line searches will not work with this direction method because momentum can produce directions that slope upwards.

When using momentum, you must set the rate of decay to use, **alpha**. **Alpha** must lie between 0 and 1 and has a default value of 0.9:

```
bp-> mz configure -direction momentum -step fixed
bp-> mz configure -alpha
-alpha 0.9 0.9 {rate of decay for momentum descent}
```

As before you, can re-randomize the net and train it again.

5.3 Conjugate Gradient

You may have noticed that the methods discussed up to this point all converge much more slowly than the default method that was used when you first created the net. The default method used by the minimizer is a much more powerful one, based on conjugate gradients. The conjugate gradient direction method chooses directions that are conjugate to previous directions. This means that minimization along the new direction does not undo the minimization already achieved along the previous directions. A line search should be used with conjugate gradient methods because directions change radically, and consequently so do appropriate step sizes.

Three conjugate gradient methods are implemented in **Xerion**: **cg** was written to follow the description in "Numerical Recipes in C" ; **cgRudi** is a modification of the standard method. The original code for this method was written by Rudi Mathon; **cgRestart** chooses directions that are conjugate to previous directions, and does a form of restarting. The original code for this method was also written by Rudi Mathon.

There are no parameters that need to be configured for the conjugate gradient methods. Try training with the standard conjugate gradient method and Rudi's line search:

```
bp-> mz configure -direction cg -step lsRudi
```

The network should converge *significantly* faster than with the previously discussed methods.

5.4 Other Methods

Besides the methods discussed above, the **Xerion** minimizers implement two training methods developed explicitly for neural networks: **deltaBarDelta**, and **quickProp**.

The delta-bar-delta direction method is an attempt to improve on straight momentum by taking larger steps (gain) for those components of the gradient whose signs are unchanging. This method was introduced by R. Jacobs in "Increased Rates of Convergence Through Learning Rate Adaptation", COINS-TR-87-117, 1987, University of Massachusetts.

As with straight momentum, line searches will not work with this direction method because momentum can produce directions that slope upwards.

Quickprop is a method developed by S. Fahlman in an attempt to get higher rates of convergence than are possible with gradient descent. In this method the current and previous component values and derivatives are modeled as independent quadratics and a jump is made to the projected minimum of each quadratic. This method is discussed in S. Fahlman "An Empirical Study of Learning Speed in Back-Propagation Networks", 1988, CMU-CS-88-162, Carnegie Mellon University.

Normally this method is used with a fixed size step, with a step size (epsilon) of one. It is possible that line searches could work with this direction method.

6 Examining Data Structures

Xerion has utilities that allow you to examine some of the C data structures in the simulator directly from the command line. This gives you much of the power normally available only inside a debugger. You can look at the values stored in structure fields, modify their visibility, and even change the values stored in the fields.

Structures that can be examined in this manner are said to be "bound" to the interface. Clearly, the most important bound structures are the ones used to build the networks and example sets. In order to view the structures we must know the name of the object used in the C code. To find out this name, most of the network commands (nets, example sets, groups, units, etc) support the subcommand **boundName**. This command returns the name of the C data structure used to build the object. For example, to find out the bound name for the Encoder network, you would use the command:

```
bp-> Encoder boundName
uts_net[1]
```

The string returned by this command is the name that would be passed to the commands discussed in the following sections.

6.1 Viewing Structures

To view the data structures, the most useful command is **uts_show**. This command takes the bound name of an object, and produces a description of the C data structure used to build the object, formatted in a way that is easy for humans to read. For example, to see the structure used to build the Encoder network use the command:

```
bp-> uts_show {uts_net[1]}
uts-net[1] = {
         numCostModels = 1
             numGroups = 4
                  name = "Encoder"
                 error = 0.0
                  cost = 0.0
          currentEpoch = 0
             batchSize = 0
             costModel = 0x10062e98 (Uts-CostModelRec * [numCostModels])
                 group = 0x10054e98 (Uts-GroupRec * [numGroups])
          parameterSet = 0x10069128 (Uts-ParameterSetRec *)
             extension = 0x10050118 (Uts-NetExtensionRec *)
}
```

Notice that the bound name of the network is quoted using braces before being passed to **uts_show**. This is necessary because of the "[]" in the name. If it is not quoted **Tcl** will try to evaluate the string between the brackets as a command, and return an error.

The **uts_show** command allows you to traverse all levels of the data structure. You can look at individual fields, or follow pointers into other data structures.

```
bp-> uts_show {uts_net[1].error}
uts-net[1].error = 0.0
bp-> uts_show {uts_net[1].group[2]}
uts-net[1].group[2] = {
              numUnits = 3
                  name = "Encoder.Hidden"
                  unit = 0x10064678 (Uts-UnitRec * [numUnits])
             extension = 0x100649b0 (Uts-GroupExtensionRec *)
}
```

When accessing fields of a structure you should use the separator "**.**", not "**->**", even if the arrow construct seems correct. The decision to always use "**.**" was made to ease typing from the command line.

If you look at the fields listed in the network objects you will notice one called **extension**. This is a structure defined differently in each simulator. It is used to hold data that is not part of the standard library, but is required by the simulator. In the back propagation simulator the network extension field looks like:

```
bp-> uts_show {uts_net[1].extension}
uts-net[1].extension = {
       zeroErrorRadius = 0.0
}
```

ZeroErrorRadius is used when calculating the error for an output unit. The error for a unit is considered to be zero if it is within **zeroErrorRadius** of its target. For example, if the target for a unit is 1.0, and **zeroErrorRadius** is 0.1, any output between 0.9 and 1.1 would produce no error (and no associated derivatives).

6.2 Setting Structure Fields

If you want to change the value of a field in a structure, you must use a lower level command: **uts_bound**. This is a much more powerful command than **uts_show**.[7] This command allows you to see what the value of an individual field is, and to change that value. To see the error of a network (the field used to store the value set by **Encoder configure -error**), use the command:

```
bp-> uts_bound set {uts_net[1].error}
0.0
```

The command is similar to the **Tcl** command **set**, except that it acts on bound structure fields. If a value is specified on the command line, then the field is set to that value before returning:

```
bp-> uts_bound set {uts_net[1].error} 1.0
1.0
```

Using **uts_bound** to change the error is unnecessary since the network allows you to change it using **configure**; however, fields in the extension record can only be changed using **uts_bound**. For example, the only way to change the **zeroErrorRadius** for the Encoder network is:

```
bp-> uts_bound set {uts_net[1].extension.zeroErrorRadius} 0.1
0.1
```

6.3 The "uts_bound" Command

The **uts_bound** command can be used to do more than just set the values of simple fields. It can:

- list all of the object types known to **uts_bound**.
- determine whether an object type is a structure or simple type.
- list the fields in a structure.
- return the name, type, and value of an object.
- set the value of an object.
- set default values for a structure field.
- set an object back to its default value.
- make a field or object read-only.
- make a field or object visible or invisible.

For details on how to perform these operations see the online help page for **uts_bound**.

[7] The **uts_show** command is, in fact, just a procedure wrapped around this command.