BPTT Simulator
Version 1.0

A. Laakmann
Winter, 1992

Contents

1 Introduction .................................................. 2

2 An Overview of the BPTT Concept ..................... 2

3 The Command Line ......................................... 2
   3.1 How to Interpret the Prompt ...................... 2
   3.2 Using the Readline Interface ..................... 3
      3.2.1 Command Completion ........................ 3
      3.2.2 Introduction to In-Line Editing ............. 3
      3.2.3 History Commands ............................ 4

4 How to Configure A Network ......................... 4
   4.1 Creating the Network ............................... 4
   4.2 Displaying the Network ............................ 5

5 Specifying Training Patterns ......................... 5
   5.1 Using Files as the Pattern Source ............... 5
      5.1.1 The “.data” File ............................ 7
      5.1.2 The “.teach” Files .......................... 7
   5.2 Using C Functions as the Pattern Source ........ 8
   5.3 Displaying your Patterns .......................... 10
      5.3.1 Displaying from Files ....................... 10
      5.3.2 Displaying from Functions .................. 10

6 Training the Network .................................. 11
   6.1 The run-network Command ......................... 11
   6.2 Tracking the Error ................................ 12
   6.3 Teacher Forcing .................................. 13
   6.4 Loading and Saving Weights ..................... 13
   6.5 Helpful Training Hints ......................... 14
      6.5.1 Learning Rates ............................. 14
      6.5.2 Delay and Dwell ............................ 16
7 Testing the Network
  7.1 Producing Output .................................................. 16
      7.1.1 What is a template? ............................................ 17
      7.1.2 Calling Templates .............................................. 19
      7.1.3 Making Graphs with GnuPlot .................................. 19
  7.2 Examining the Weights ............................................ 20
  7.3 Stimulating and Lesioning Units ................................. 20

8 Advanced Features
  8.1 Disconnecting Units and Fixing Weights .......................... 20
  8.2 Changing the Bias .................................................. 21
  8.3 Adding Gaussian Noise ............................................. 21
  8.4 Changing the Random Initial Weights ............................. 21

9 Using Script Files
  9.1 Creating Scripts ................................................... 22
  9.2 Running Scripts .................................................... 22
  9.3 The “.bptinit” File ................................................ 22
  9.4 Automatic Network Scripts ....................................... 22

10 Running BPTT in the background
  10.1 Automatic Backgrounding ......................................... 23
  10.2 Backgrounding from the Shell .................................... 23

11 The BPTT Text Editor ................................................. 24

A Command Descriptions .............................................. 25

B Parameter Listing .................................................... 27
1 Introduction

This manual has been designed to serve two purposes. The first goal is to offer an introduction to the commands and techniques required to successfully use BPTT. The second goal is to offer a quick reference source for users desiring more specific information.

2 An Overview of the BPTT Concept

The principle behind the back-propagation through time learning algorithm is extending a feed-forward network through time. In theory, this can be accomplished by remembering all previous time steps, and allowing those older states to continuously affect the current computation of the most recent time step. Although an infinite number of past time states, or copies, would be required to exactly simulate a recurrent network, it turns out that a more reasonable number allows a fairly accurate approximation.

The concept of copies in BPTT should not be confused with the concept layers in a feed-forward network. A copy refers to the network activation at a given time step. For every time step of the network, a new copy is generated and the oldest is thrown out. Activation then flows from the oldest copy forward, resulting in an activation for the units of the most recent copy. It is important to note that past input lines to each copy are saved. Thus, when activation flows forward, the input values to past copies are again used in computing the new activation values. Once the latest activation vector has been produced, the output units receive an error signal that propagates backwards through all copies. This structure can simulate a recurrent network since, on every time step, a unit both receives information and can change the weights of past time states.

3 The Command Line

This section examines the BPTT prompt and the Readline interface.

3.1 How to Interpret the Prompt

The BPTT prompt displays information regarding the current status of the network, patterns, and weights. When BPTT is first run, the following prompt will appear:

```
bptt [file-NWP] $```

The “file” string specifies that files are being used as the pattern source. The alternate state for this flag is “func”, which means internal C functions, written by the user, will generate the input and output patterns used by the network.

The “N” flag indicates that no network has been configured. Likewise, the “W” flag indicates that no weights have been created.

The last flag, “P”, is present when no patterns have been loaded or internal C functions have not been compiled with BPTT. This position can also contain an “I” or “T”, indicating that input or teacher patterns have not been loaded.
3.2 Using the Readline Interface

The Readline Interface is an advanced interface created by the generous folks at GNU (the makers of Emacs). This interface allows command completion, in-line editing, and a history package enabling quick recall of previous commands. Only the most important commands will be described in this manual, and readers desiring more information should examine the Readline manual.

3.2.1 Command Completion

BPTT will automatically complete command names, filenames, and parameter names with the [TAB] key. If the [TAB] key is pressed during the first word on the line, BPTT will attempt to complete with the most appropriate command name. If the first word on the line is set or print, BPTT will complete with the closest parameter name. In all other cases, BPTT will attempt to complete with the best match filename. A list of possible completions can be displayed at any time by pressing the [TAB] key twice.

3.2.2 Introduction to In-Line Editing

The following conventions will be used in this manual. The text C-k is read as “Control-K”, and this describes the character produced when the Control Key is depressed with the K key. The text M-K is read as “Meta-K” and describes the character produced when the meta key and K key are depressed simultaneously.

The following are a list of editing commands available at the BPTT prompt. Many of the keystrokes are similar to some emacs editing commands.

C-B Move back one character
C-F Move forward one character
DEL Delete the character to the left of the cursor
C-D Delete the character underneath the cursor
C_- (underscore) Undo the last thing you did

Movement Commands

C-A Move to the start of the line
C-E Move to the end of the line
M-F Move forward a word
M-B Move backward a word
C-L Clear the screen, reprinting the current line at the top

Killing Commands

Killing text means to delete the text from the line and save it away for later use. The text can be reproduced by yanking it at back into the line.
C-K  Kill text from the current position to end of line
M-D  Kill from the cursor to the end of the word
C-W  Kill from the cursor to the previous whitespace
C-Y  Yank the most recently killed text back into the line
M-Y  Rotate the kill-ring, and yank the new top. You can only do this if the prior command is a C-Y or M-Y

3.2.3 History Commands

The following commands allow you to quickly retrieve previously entered lines. The old line replaces your current line, and in-line editing of the new line is possible.

C-P  Retrieve the previous line (can be used repeatedly)
C-N  Retrieve the next line (can be used repeatedly)
C-R  Interactive reverse search (try it! very nice!)

4 How to Configure A Network

4.1 Creating the Network

Before a network can be run, it must be configured. Networks are specified in BPTT through the use of various parameters. The following parameters define the actual architecture of a network:

units
The total number of units in the network. This number includes output and hidden units, but not input lines

copies
The previous time steps to remember (i.e. the stack size)

inputs
Total number of input lines (including bias) into the network. ex. if two input lines are desired, this parameter should be set to three in order to include the bias

outputs
Maximum number of units that will receive training. As a matter of convention, output units are always the low numbered recurrent units. For example, a network with 3 units and 1 output would have hidden units numbered 1 and 2, and an output unit numbered 0.

The set command is used to change these and other parameters. Its form is as follows:

set (parameter) (value)
bptt [file-WNP] $ set units 3
bptt [file-WNP] $ set copies 5
bptt [file-WNP] $ set inputs 3
bptt [file-WNP] $ set outputs 1
bptt [file-WNP] $ initialize-weights
bptt [files]

Figure 1: Commands to Configure an XOR network. Notice the H disappears when the network is defined and the W disappears when the weights are initialized. The only prompt flag remaining is the P, which means no patterns have been loaded.

Once a network is configured, the weights can be initialized with the initialize-weights command. Figure 1 contains a list of commands that can be used to generate a network capable of solving the XOR problem.

Other important configuration parameters not included in the above list are delay, dwell, bias, and eta. While these parameters do not define the physical size of the network, proper values for these parameters are required for successful training.

### 4.2 Displaying the Network

Network parameters can be displayed with the display-configuration command. Additionally, a text picture can be seen with the display-network command. Figure 2 illustrates the use of these commands.

### 5 Specifying Training Patterns

To train a network, input and teacher patterns must be supplied. BPTT allows two methods of pattern specification. The first, and simplest, is to use files that list the patterns. The second, and more involved, involves linking C functions that generate patterns. Unless you know how to program in C and enjoy a little bit of hacking, it is recommended that files be used to supply the patterns.

#### 5.1 Using Files as the Pattern Source

When using the file interface, BPTT requires two files. The first file must include a “.data” extension, and this file specifies the input patterns. The other file should have “.teach” extension, and this file contains both the teacher and tmask patterns. Tmask patterns indicate which, if any, output units are to receive training during a given time step. The maximum number of pattern pairs that can be loaded is 3500. This limit is set to conserve memory usage. Note that there should be exactly the same number of patterns in the “.data” and “.teach” file. Once the files have been created, they can be loaded with the load-patterns command.
bptt [file-P] $ display-configuration

Units = 3    Copies = 5    Inputs = 3    Outputs = 1
Delay = 2    Dwell = 1    Bias = 1.000
Fixed weights: none
Number of Patterns: 4

bptt [file-P] $ display-network

0 1 2 0 1 2

B I I ---> O R R
B I I ---> O R R
B I I ---> O R R
B I I ---> O R R
B I I ---> O R R

Figure 2: Commands for displaying network parameters. In the output of the display-network command, (B) stands for the bias, (I) stands for input units, (O) stands for output units, and (R) stands for other recurrent units (hidden units). Output units, as a matter of convention, are always the low recurrent units. Thus, in the XOR network configured above, the output unit would be unit number 0, and units 1 and 2 would be the “hidden” units.
5.1.1 The “.data” File

This file defines the input patterns which are presented to BPTT. The first line must be an integer specifying the number of input patterns to follow. The remainder of the file consists of the actual input vectors, and these may be composed of either integers or floating point numbers. Each pattern should contain one less number of values than the number of inputs into the network (the bias is automatically included). Note that these input patterns are presented to the network in sequential order unless the -R flag is used while training (see the section on “Training the Network”, pg 10). Figure 3 illustrates example “.data” files.

5.1.2 The “.teach” Files

This file has a similar, but more complex, structure than the “.data” file. The first line is an integer again specifying the number of patterns to follow. Like the “.data” file, the remainder of the lines supply the actual vectors. The additional complexity is the optional tmask array which can be included for each pattern.

The tmask pattern is used to indicate which units, if any, should receive training during the current time step. For most applications, training is desired across all output units on every time step, and this can be accomplished by either supplying an asterisk after each teacher pattern or turning on the train-all parameter (note that if the train-all parameter is set, no tmask symbols should be present in the “.teach” file). The train-all parameter is initially set when BPTT is started, so most users can simply ignore the tmask array entirely.

Sometimes, applications require that individual units receive no training, entire input patterns should be presented without any training, or that a delay should exist between corresponding input and output patterns. Some of these effects can be accomplished with the delay and dwell parameters (see the corresponding section, pg 16, for more information). If these parameters cannot produce the desired effects, the tmask array should be altered.

The tmask pattern is a vector composed entirely of ones and zeros that, if specified, must be precisely the same size as the output vector. A pipe symbol (|) indicates the

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>0.0</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Figure 3: Example “.data” files. The file on the left could be used with the XOR problem. Notice that it is impossible to tell the number of inputs by looking at the file. This value must specified with the inputs parameter which can be set while in the program.
beginning of the tmask pattern. Where a one is present in the tmask array, training will occur, and where a zero is present, no training will take place.

In addition to the asterisks and pipe symbol, a caret (*) can be used to specify that no training should occur for the entire pattern. Figure 4 has example “teach” files that illustrate the uses of the asterisk, pipe, and caret symbol.

Altering the tmask array is for advanced applications only, and most users should be able to accomplish their desired effect by just using the asterisk and/or manipulating parameters.

5.2 Using C Functions as the Pattern Source

To do this, copy all of the files and directories from your local bptt directory into your working directory. On icogsci1, the following command will accomplish this:

```
    cp -rp /home/other/za7/src/BPTT/src/*.
```

The file, “user_function.c” should be modified to include your pattern generating functions. Room is allowed for four separate pattern generators, and they can be selected
while running BPTT by modifying the `psource` parameter.

Bring up the “user_function.c” file into your text editor. Change the second line from the top to read

```c
int FUNCTIONS = TRUE;
```

This indicates to BPTT that functions will be compiled with it. You will notice that there are four function headers, and each of these correspond to a different pattern generator. An example header is as follows:

```c
void pattern_generator_1 (double input [], double output [], int tmask [], int start) {
}
```

When generating functions, your C code should modify the input, output, and tmask arrays according to the designs of your pattern generator. For most applications, the tmask array should just be set to all ones. Refer to the section on generating teacher files for more information regarding tmask arrays. Additionally, the “start” flag will be set to 1 when the function is first called. This allows the function to initialize any static variables that might be used during the generation.

Global variables can be referenced from your C function that can be modified at runtime in BPTT. There are ten variables, labeled USER0 to USER9, which are of type double. The `set` command in BPTT can be used to change these variables. Additionally, the global variables `NET_UNITS`, `NET_INPUTS`, and `NET_OUTPUTS` can be referenced to get size information about the network.

Here is an example function that trains a network to oscillate with a sine wave. Notice that there is no input into the network except the bias. Also notice that the tmask array is initially set to one (by checking the “start” variable). The user parameter USER0 can be changed at runtime to indicate to the function how many degrees (in radians) to move the counter on each timestep.

```c
void pattern_generator_1 (double input [], double output [], int tmask [], int start) {
    static i = 0;

    if (start)
    {
        i = 0;
        tmask[0] = 1;
        input[0] = 1;
    }
    else
    {
        output[0] = (1 + sin (i * (3.1415927 / 180))) / 2;
        i = i + USER0;
    }
}
```

Once the functions have been created. Type “make” in your working directory to recompile bptt. An executable called “bptt” will be placed in the current directory.
FILE source: (inputs) | (teachers) | (tmasks)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.000</td>
<td>1</td>
</tr>
<tr>
<td>1.000</td>
<td>0.000</td>
<td>1</td>
</tr>
<tr>
<td>1.000</td>
<td>1.000</td>
<td>0</td>
</tr>
<tr>
<td>0.000</td>
<td>1.000</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5: Example output from the display-patterns command. Notice that the input, teacher, and tmask array are all displayed.

5.3 Displaying your Patterns

Once pattern sets have been loaded or internal functions compiled, BPTT can be used to display your pattern sets. In either case, the display-patterns command is used to examine the patterns. The format for the command is slightly different depending upon which file source is being used. Example output from this command can be seen in Figure 5.

5.3.1 Displaying from Files

When patterns are loaded from files, each pattern set is assigned a unique number. The first pattern set (input, output, and tmask) is assigned the number zero, and each successive pattern set is assigned the next integer.

When using the display-patterns command with files, if no arguments are supplied, all the patterns will be listed. If only certain patterns are desired, a pattern range can be specified as follows:

    display-network 10-20

5.3.2 Displaying from Functions

If functions have been compiled with BPTT, and these are being used as the pattern source, the display-network command can be used to examine the output of these functions. This can be helpful in debugging your generator functions.

When functions are being used, the display-network command must be supplied an integer argument. This value indicates the total number of calls to your generator function. For instance, if you wish to examine 100 time steps of output from your generator functions, the following command would suffice:

    display-network 100
6 Training the Network

Training a network involves presenting input patterns, generating network activations, comparing these activations with teacher patterns, and changing various connection strengths between units. Because of this, certain prerequisites must be met before training can begin. Before BPTT will train a network, the follow conditions must be established: patterns must be present, a network must be configured, and weights must be initialized. Once these conditions have been met, training will be allowed.

6.1 The run-network Command

The run-network command is used to train the network. This command accepts a variety of required and optional flags that allow specific customization of the training process.

The following flags can be supplied with run-network. All flags are optional except the -s flag. Arguments to flags in {curly braces} are optional, while those in (parenthesis) are mandatory. Certain flags are linked to global parameters, and these parameters can be set with the set command. If a flag has a corresponding global parameter, then the parameter name is indicated with *asterisks*. For example, if the command "set random on" is entered at the BPTT prompt, the -R flag will always be set when the run-network command is called. Likewise, if the "set update 1000" command is entered, the -u flag will always be assumed, and the value in update will be supplied with it.

-s (num)
  Specifies the number of sweeps
  (ex. run-network -s 1000)

-R *random *
  When used in file mode, it indicates that patterns should be presented in a random, and not sequential, order. For example, the XOR patterns need to be presented in a random order to prevent the network from learning to produce a cyclic patterns of ones and zeros.

-E {filename} num
  Prints pattern error averaged over (num) [default 100] sweeps. If {filename} is supplied, the values are sent to the indicated file instead of to the screen.
  (ex. run-network -s 1000 -E hello 10)
  See page 12 for more information.

-u (num)
  *update *
  Prints a message every (num) sweeps indicating the number of sweeps completed so far. If no message is desired, a value of 0 should be supplied.

-lesion (num) ...
  Specifies which units should be lesioned. Lesioned units will be clamped with the value of the lesion parameter. lesion parameter defaults to 0.0
  (ex. run-network -s 1000 -lesion 1 2 3).
-stimulate (num) . . .
   Similar to lesion, but units are clamped with the value of the stimute parameter. Stimulate parameter defaults to 1.0

-o (filename) {num} . . .
   Used to generate network output. See page 16 for more information.
   (ex. run-network -s 1000 -o hello.tem goodbye.tem 2)

-M (num)
   Used to terminate training before the maximum number of sweeps is reached. If the average RMS error goes below the supplied (num), training will be terminated. The average RMS error is the same as the one outputted by the -E flag. If the -E flag is not used, a default of 100 sweeps is used.
   (ex. run-network -s 10000 -M 0.05)
   (ex. run-network -s 10000 -E 250 -M 0.05)

-t
   Used to test the network, it signals that no learning should occur.
   See pg 16 for more information.

-f {num} . . .
   Use teacher forcing (clamping output units with teacher values after training). When used, all output units will be forced except those unit numbers which appear after the -f flag.
   See pg 13 for more information

-B (filename)
   This flag signals BPTT to background itself. Control will be passed back to the shell, but BPTT will continue running in the background. When training is finished, BPTT will save the weights to the specified filename and terminate.

-n (num)
   Add gaussian noise distributed in the range of plus or minus the supplied num to the activation of all units in the network.

6.2 Tracking the Error

As mentioned above, the -E flag can be used to track the error while training is occurring. This error information can be directed to the screen or to a file. When sent to the screen, the error information can be used to monitor training. If sent to a file, the file can be used by most plotting programs to produce error graphs (Figure 7 illustrates an example error graph). See the section on GnuPlot, pg 19, for specific information about plotting error graphs. Error is computed using this function:

\[ error = \sqrt{\frac{\sum_{i=0}^{n-1} (o_i - l_i)^2}{n}} \]
with $n$ being the number of outputs, $o_0...n-1$ being the output activation array, and $t_0...n-1$ being the teacher array. Figure 7 contains an example transcript depicting the use of error tracking.

6.3 Teacher Forcing

Teacher forcing is helpful when training oscillatory problems. If teacher forcing is used, output activations are clamped to the corresponding teacher values immediately after each time step. The -f flag is used to indicate teacher forcing. Here is an example:

```
run-network -s 2000 -f
```

Output unit numbers can be supplied after the -f flag, and those units will not be forced. This option allows teacher forcing of selective output units. Here is an example:

```
run-network -s 2000 -f 0 1 2
```

In this example, all output units except 0, 1, and 2 will be teacher forced.

6.4 Loading and Saving Weights

Network weights can be saved and reloaded by means of the `save-weights` and `load-weights` commands.
bptt [file] $ run-network -s 1000 -E error 20
bptt [file] $ run-network -s 100 -E

0.45456
0.40344
0.35233
0.30434
0.15454

Figure 7: The first example will send the RMS error averaged over 20 sweeps to a file called "error". The second example will print the RMS error averaged over 100 sweeps (the default if no number is specified) to the screen.

The save-weights command produces a weight file that contains the connection strengths of all units within the network. A filename must be specified. Note that a "wts" extension will be concatenated to the end of the filename. Here is an example:

bptt [file] $ save-weights xor

In addition to the weight values, comments (surrounded by parenthesis) will be included to allow easier reading of the weight file. These comments are ignored by the load-weights command. If for some reason the comments are not wanted, the save-weights command can be called with the -nc flag.

Weights can be reloaded by means of the load-weights command. This command will reconfigure the current network to match the architecture specified at the beginning of the "wts" file. Figure 8 contains the first portion of an example weight file.

6.5 Helpful Training Hints

6.5.1 Learning Rates

Proper selection of a learning rate is critical to correctly train a recurrent network. If a learning rate is too large, a network can "bounce" into an oscillation that will prohibit further training. Absolute values for learning rates are not known, and only through experimentation can you find the optimum learning rate for your problem. The learning rate is linked to the parameter eta, and can be changed with the set command. For example,

set eta 4.0
(UNITS COPIES INPUTS OUTPUTS)
5 5 3 1
(C:0 R:0 from R0-5)
-0.688658
-0.846051
-2.055128
-7.585862
7.740448
(C:0 R:0 from IO-3)
-3.051260
-0.096400
-0.090721
(C:0 R:1 from R0-5)
-0.220689
-0.555760
-0.738941
-0.466256
-0.416252
(C:0 R:1 from IO-3)
-1.059118
-1.568864
-1.314695

Figure 8: Example “.wts” file. Notice that the network architecture is specified at the beginning of the file and parameters such as delay, dwell, and eta are not included in the weight file.
Delay = 1, Dwell = 1
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

Delay = 2, Dwell = 1
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

Delay = 2, Dwell = 2
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

Delay = 4, Dwell = 2
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

Figure 9: Various effects of the delay and dwell parameters. In these examples, letters indicate the input and teacher patterns. The top row in each case depicts successive teacher patterns, while the bottom row depicts each input pattern. Each column is one network time step, and time moves from the left to the right. Where a “-” is present, no training will occur.

6.5.2 Delay and Dwell

Like the learning rate, improper values for the delay and dwell parameters can prohibit a network from learning a problem. Delay indicates the number of time steps between the presentation of an input pattern and the training on its corresponding teacher pattern. Dwell indicates how many network ticks an input pattern will remain over the input units before training occurs. Figure 9 illustrates the effects of various delay and dwell settings.

7 Testing the Network

Generally, while testing a network learning should be suppressed. This can be accomplished by either specifying the -t flag with the run-network command, or by running the test-network command. The structure of test-network is identical to that of run-network, the only difference being the assumed -t flag with test-network. The best way to test and analyze a network is to examine the output. The following sections describe how templates can be created and used in order to produce output.

7.1 Producing Output

BPTT supports a template driven output system that allows users to customize the structure of their output. Network output can be displayed to the screen, sent to a file, or plotted with various graphics package.
7.1.1 What is a template?

A template is a file created in your favorite text editor that allows you to specify output formats. Templates allow you to print out characters, units, and error values. In addition, the output can be directed to either the screen or files. Here is an example template:

```
"hidden-units" (r3-4)
"outputs" (r0-2)
here are units : (i0-2)
```

Each line in a template file reflects one line of output. If the line begins with a quoted name, then that name is assumed to be a filename, and the following line is sent to the specified file. Any characters not in parenthesis are translated literally, and reproduced as is on the screen or file. For instance, the `here are units` of the third line would be reproduced every time the template is called.

Template commands (those characters in parenthesis) are used to print unit values, output error values, and display teacher values. In the above example, the (r3-4) means to output the values of recurrent units 3 through 4. The (i0-2) means to print the values of input units 0 through 2.

If the template displayed above were in a file called “temp”, and the following command was entered:

```
test-network -s 3 -o temp
```

This would get printed to the screen (assuming these values are the input values).

```
here are units: 0.434 0.343 0.657
here are units: 0.434 0.343 0.657
here are units: 0.434 0.343 0.657
```

This would get sent to a file called “hidden-units” (once again, fake values are used).

```
0.776 0.989
0.343 0.122
0.656 0.323
```

And finally, this would get sent to a file called “outputs”.

```
0.232 0.656 0.120
0.989 0.153 0.645
0.123 0.876 0.765
```

Below is a listing of available template commands:

(r) specifies recurrent units. Remember, output units are low values recurrent units, so this command should be used to display output values as well.

ex. (r1-10) or (r3)
(i) specifies input units
    ex. (i0) or (i3-4)

(t) indicates actual teacher values for specified units
    ex. (t1-3) or (t6)

(e) the error (difference squared) for units
    ex. (e1-3) or (e13)

(o) this command will offset any values printed during the remainder of the line by the
    supplied amount
    ex. (o-1.5) or (o1.5)

(s) similar to offset, but scales the values (note: scaling always takes precedence over
    offsetting)
    ex. (s-1.3) or (s2)

Within the (r), (e), (i), or (t) commands, the following symbols can be used. These
symbols will get replaced with the indicated value.

I : total number of inputs minus 1
U : total number of units minus 1
T or O : total number of outputs minus 1

Note that all the values have been decremented by one. This is because unit numbers
begin at 0, which means a 20 unit network has units numbered from 0 to 19. These
symbols have been included to allow creation of generic templates that work for networks
of any size.

The following is a template that displays useful information and illustrates the function
of these special symbols:

    (i1-I) ! (r0-T) ! (t0-T) ! (e0-T)

This template would output all of the input lines (not including the bias), the output
units (since output units are the low recurrent units), the teacher values, and finally the
error values. If there were 3 inputs lines and 1 output unit, this would be an example
printout:

    1.000 0.500 ! 0.100 ! 0.500 ! 0.160
7.1.2 Calling Templates

The “run-network” section on page 10 mentioned the -o flag. This flag can be used with either the run-network command or the test-network command to call templates. The structure of this flag was mentioned previously, but it will be repeated here:

-0 (filename) {num} ...

Here is an example:

test-network -s 100 -o hello 10 goodbye aloha 3

This example would test the network for 100 sweeps. Every tenth sweep the template called “hello” would be run, every sweep the template “goodbye” would be run, and every third sweep the “aloha” template would be called. From this example, it should be clear that the (filename) specifies which template to call, and the optional number (default 1) specifies the frequency with which to call it. Any number of templates may be specified, but at least one must be present with the -o flag.

7.1.3 Making Graphs with GnuPlot

Templates can be used to generate files suitable for plotting with many graphics packages. Popular graphs to plot are hidden unit activations, output activations, and error curves.

Although many graphic packages exist, Gnuplot will be described here. There are a few reasons why GnuPlot is the graphics package covered in this manual. GnuPlot is a program that is fairly simple to use, has many features, and is readily available. A word of warning though, a comprehensive account of GnuPlot is beyond the scope of this manual, and only the barebones necessities will be presented here.

The simplest file that GnuPlot will plot is a simple list of y-values. With a file like this, GnuPlot will generate a graph of the y-values verses an increasing x counter beginning at 0. This file structure is easily obtained with BPTT templates.

As an example, let us assume that we have a network with 1 output unit and 2 hidden units. If we wanted to plot the activations for all three units on the same graph, the following template would generate files with the required structure:

"out0"(r0)
"hid1"(o1.5)(r1)
"hid2"(o3.0)(r2)

Use of this template with a network command would generate the three files. Each file would contain the values of the specified unit over all the sweeps, and the hidden unit values would each be offset by a certain amount. This offset is to insure that GnuPlot does not plot the graphs right on top of one another, and instead stacks them atop one another. The specified offsets would put the output unit at the bottom of the graph, with each hidden unit stacked above it.

Once these files have been generated, BPTT could be terminated with the quit command or suspended with C-Z (to resume BPTT, type fg at the shell prompt). Gnuplot can be invoked from the shell by typing gnuplot. The following command in GnuPlot will display the graphs:
plot "out0" with lines, "hid1" with lines, "hid2" with lines

GnuPlot has an extensive help package. This can be started by typing help at the
GnuPlot prompt. It is suggested that all those interested in making graphs should read
this help file.

7.2 Examining the Weights

The command, graph-weights, can be used to generate a postscript file illustrating
the relative connection strengths of units. The postscript file can be sent to any laser
printer to viewed with a postscript viewer. Immediately after the command call, a filename
should be included. A ".ps" will be concatenated to the end of the filename. Graph-
weights accepts a variety of optional arguments that allow customization of the final
image. Here is a list of the available options for graph-weights.

\begin{verbatim}
graph-weights (filename) { -t (string) } make a title
{ -c } include a caption
{ -k } include a key
{ -rf (val) } font size (in) of text
{ -tf (val) } font size of title
{ -m (val) } modulus for ref.# printing
\end{verbatim}

Here are some example graph-weights calls:

\begin{verbatim}
graph-weights temp -t Network Weights -c -rf 0.2 -tf 0.4
graph-weights temp2 -k -m 2
\end{verbatim}

7.3 Stimulating and Lesioning Units

Some insight into the function of hidden units can be extracted by stimulating and
lesioning units within the network. The run-network and test-network commands
support flags that allow such a surgery on the network. The -lesion flag clamps the value
of the lesion parameter onto the specified units. The -stimulate flags clamps the value
of the stimulate parameter onto the specified units. Example calls are below:

\begin{verbatim}
bptt [file] $ run-network -s 1000 -lesion 1 2 3 4 5
bptt [file] $ test-network -s 1000 -stimulate 2
\end{verbatim}

8 Advanced Features

The following is a description of some advanced features that can help with training
and testing of networks.

8.1 Disconnecting Units and FixingWeights

Sometimes a partially recurrent network is desired. Connections between units can be
removed or fixed with the fix-weights command. The following are some examples:
bptt [file] $ fix-weights (rec)(i0)(-2.5)
bptt [file] $ fix-weights (r0-5)(inp)(0)
bptt [file] $ fix-weights (out)(inp)(0)

The first example would fix the bias weights of all recurrent units (including output units) to -2.5. The second example would disconnect recurrent units 0 through 5 from all input units (fixing a weight to zero is identical to removing the connection). The last example would disconnect all output units from all input lines.

The display-configuration command will display which weights, if any, are fixed.

8.2 Changing the Bias

Sometimes a value other than 1.0 is desired for the bias. This value is linked to the bias parameter and can be easily changed with the set command. Here is an example:

```
set bias 0.5
```

8.3 Adding Gaussian Noise

Noise can be added to activation of all units in the network by using the -n flag with the run-network or test-network command. An argument should be supplied after the flag which indicates the range of the noise. The values added will be gaussian distributed in the range of plus or minus the supplied value.

8.4 Changing the Random Initial Weights

The parameter wrange can be used to specify the range for the initial random weights. This number is used to indicate the plus/minus range for the random initial values. Networks with large numbers of units need smaller values for wrange to minimize the value of net for each unit. The default value for wrange is 1.0, which creates initial random weights between -1.0 and 1.0. This value can be changed with the set command as follows:

```
set wrange 0.5
```

9 Using Script Files

Script files are lists of commands to be executed by BPTT. By creating a script file, the user can call the script file and execute a sequence of commands without having to type them in. Perhaps the most useful application of a script file is for specifying network architecture. For instance, a script file containing commands describing an XOR net could be created, and then any time an XOR architecture was required, the script could be invoked.
set units 3
set outputs 1
set copies 5
set inputs 3
initialize-weights
set delay 2
set eta 4.0

Figure 10: Script file suitable for XOR network

9.1 Creating Scripts

Script files must be created in a text editor. BPTT supplies a line editor within the program, and this package is described later in the manual. One advantage of using the BPTT editor is the command completion. The BPTT editor will complete command names with the [TAB] key (like at the prompt) allowing script files to be generated with ease. Figure 10 contains an example script file suitable for creating an XOR network.

9.2 Running Scripts

Script files can be started with the execute-script. The output during a script file will be suppressed unless the -out flag is used during the command call. Two examples calls are below:

execute-script nothing.scr
execute-script xor.script -out

9.3 The “.bpttinit” File

If a file with this name exists in the home directory, BPTT will execute it like a script file when the program is started up. This allows users to define environments that are automatically established when BPTT is begun.

9.4 Automatic Network Scripts

BPTT includes a command to automatically generate a script describing the current network environment. This function, make-network-script, creates a script file listing all of the current parameters and fixed weights. Figure 11 contains an example script file created by this command. Here is an example call:

make-network-script sample.script
set units 5
set copies 5
set inputs 3
set outputs 2
set delay 2
set dwell 1
set bias 1.0
:
initialize-weights
fix-weights (rec)(io)(-2.5)

Figure 11: Script file generated by **make-network-script**

## 10 Running BPTT in the background

Big networks can take a LONG time to train. Sometimes it is helpful to start a network training, log out, and come back when the network is done. BPTT can be instructed to run in the background using two different methods. With the first method, BPTT can background itself. With the second method, BPTT can be backguarded from the shell.

### 10.1 Automatic Backgrounding

When training or testing a network, BPTT can be automatically put into the background from the BPTT prompt. The `-B` option for the **run-network** command will signal that BPTT should background itself. When back guardsed, training or testing of the network will continue, but control will be passed back to the shell. Once training is completed, the weights will be saved to a specified filename. The filename must be included after the `-B` option. After saving the weights, BPTT will terminate itself. Here is an example call:

```bash
run-network -s 100000 -f -R -B hello
```

This example would train the network for 100000 sweeps, save the weights to a filename called “hello.wts”, and then quit.

### 10.2 Backgrounding from the Shell

BPTT, like any process, can be put into the background directly from the shell. This is accomplished by including the “&” command along with the BPTT call. The following shell command would run BPTT in the background using “temp.script” as a command script file.

```bash
% bptt < temp.script > /dev/null &
```
This command instructs the shell to call bptt and accept input from the file “temp.script”. The “/dev/null” directs all output to an empty device, thus eliminating any output from the program.

When creating a script file for background running, the file must terminate with a quit command in order to end BPTT. Also, if any weights are to be saved, a save-weights command should be included immediately before the quit command.

Running networks in the background can seriously slow down a machine for subsequent users. It is therefore suggested that backgrounded networks be “nice’d”. The nice command at the shell will instruct the operating system accordingly. Here is an example:

% nice bptt < temp.script > /dev/null &

11 The BPTT Text Editor

BPTT includes a simple line editor for quick generation of templates and script files. The editor is invoked with the editor command, and a filename should be specified. A new file is created if the file does not exist, otherwise the old file is read in.

The following commands are supported in the text editor:

list  list the current file
add  add lines to the end of the file
insert (num) insert lines in specified position
delete (range) delete the indicate lines
modify (num) change indicated line number. Press C-P to “yank” in old line for editing.
quit  abort editor and do not save file
exit  save file and exit editor

The primary attraction of using the BPTT editor is automatic command completion. Command and parameter names will be completed with the [TAB] key (like at the prompt) while in the BPTT editor.
A Command Descriptions

The following is a list of available BPTT commands. Arguments in (parenthesis) are required, while those in {curly braces} are optional. The commands run-network and test-network have too many flags to list, and readers desiring more information on these commands should see the corresponding section in the manual.

display-configuration
Displays critical parameters for network architecture and training. (pg. 5)

display-network
Prints a text picture of network. (B) = bias, (I) = inputs, (O) = outputs, and (R) = hidden units. (pg. 5)

display-parameters
Prints a list of all BPTT parameters and their current values

display-patterns (range) or (num)
When in “file” mode, this function displays the input, output, and teacher patterns. A range of pattern numbers may be included (ex. display-patterns 10-20). If in “func” mode, a number must be included which indicates the number of times to call the current pattern generator. The results of the pattern generator (inputs, outputs, and tmask) will be displayed.

display-source
Displays most recently loaded filename for patterns and weights

editor
Invokes the BPTT text editor. (pg. 24)

execute-script (filename) {-out}
Calls a specified script file. A -out flag can be included to indicate output should not be suppressed. (pg. 21)

fix-weights (to units)(from units)(value)...
Fixes weights between listed units to indicated value. (pg. 20)
ex.
fix-weights (out)(inp)(0)
fix-weights (rec)(i0)(-2.5)
fix-weights (r2-5)(i1-2)(2.5)

help (command name)
 Displays on-line help for various commands

initialize-weights {-S (num)}
Creates weight matrix with random connection strengths. Weight matrix corresponds to architecture specified by the network parameters. The -S flag can be used to specify a fixed random number seed.

load-inputs (filename)
Loads input patterns. (pg. 5)
load-patterns (filename)
   Loads both input and teacher files. (pg. 5)

load-teachers (filename)
   Loads teacher file. (pg. 5)

load-weights (filename)
   Loads specified weight file. Network architecture is automatically configured to
   match specifications inside weight file. (pg. 13)

make-network-script (filename)
   Creates a script file defining current network architecture and parameters. (pg. 22)

print (parameter name)
   Displays current contents of indicated parameter.

quit
   Terminates BPTT

run-network -s (num) . . .
   Runs the network. (pg. 10)

save-weights (filename) {-nc}
   Saves the current network architecture and weights. Comments are included in the
   weight file unless the -nc flag is included. (pg. 13)

set (parameter name) (parameter value)
   Sets the specified parameter to the indicated value.

test-network
   Identical the run-network command, but the -t flag is automatically supplied.
   (pg. 16)
B Parameter Listing

This is a listing of available parameters in BPTT. After each parameter name is a symbol indicating what type of value it can take. The symbols should be interpreted as follows:

**boolean** parameter can be on or off

**real** any floating point number

**integer** any integer number

Certain parameters have maximum and minimum values. If a parameter is set out of its range, an error message will occur indicating the maximum and minimum values for the parameter. After each type, a value is shown indicating the parameter’s default.

**bias** real 1.0
value of input line

**copies** integer 0
number of previous time steps to remember

**delay** integer 1
number of time steps between presentation of an input pattern and training on its corresponding teacher pattern. (pg. 16)

**dwell** integer 1
number of time steps that an input pattern will remain over the input units. (pg. 16)

**eta** real 1.0
the learning rate

**inputs** integer 1
number of inputs (including bias) into network

**lesion** real 0.0
value clamped onto lesioned units. (pg. 20)

**outputs** integer 0
maximum number of units that will receive training. Output units are always the low numbered recurrent units. (pg. 4)

**psource** integer 0
when zero, this indicates that files will be used to specify patterns. If non-zero, then the corresponding internal function will be used to generate the patterns.

**random** boolean off
indicates that patterns (from files) should be presented in a random, and not sequential, order.

**stimulate** real 1.0
value clamped onto stimulated units.
**train-all** boolean *on*
indicates that training should occur for all teacher patterns

**units** integer 0
total number of hidden and output units in network

**update** integer 1000
indicates how often to print a message indicating how many sweeps have been completed thus far. If set to zero, no message will be displayed.

**wrange** real 1.0
indicates the range (plus or minus value) for initializing the random weights. For instance, the default value of 1.0 will generate random weight values between -1.0 and 1.0.

The following parameters can be used with user generated pattern functions. These parameters can be set normally, and then used within the pattern generator functions. For instance, the *user0* parameter can be called from internal functions by referencing the global variable **USER0**

- user0
- user1
- user2
- user3
- user4
- user5
- user6
- user7
- user8
- user9