APPENDIX C

File Formats for Network Weight, Template, Look, and Pattern Files

This appendix describes the formats used in network, weight, template, look, and pattern files. These files are generally referred to as *.net*, *.wts*, *.tem*, *.loo*, and *.pat* files, respectively.

NETWORK FILES

The *.net* file is used to specify the network architecture for the following programs: **iac, cs, pa**, and **bp**. The **ia** program has a fixed network, and the **aa** and **cl** programs have networks of variable size but with a totally predictable architecture, so a *.net* file is not necessary.

The *.net* file consists of several sections, some of which may be optional. These sections are definitions, network, constraints, biases (for **cs**, **pa**, and **bp**), and sigmas (used in *harmony* mode in **cs**). Each section plays a different role in defining the network. The *definitions* section is used to specify the number of units in the network and also specifies other crucial variables. The *network* section is used to specify the pattern of connections among the units. Connections are of various types, each designated by a single letter. These connection types are defined in the *constraints* section, so called because the different types of connections are specified by constraints on the values they can take. The *biases* and *sigmas* sections specify characteristics of the *biases* and *sigmas* (if any) associated with the units in the network. Like connections, *biases* and *sigmas* can be of various types, each designated by a single letter defined in the constraints section. The definitions section of the *.net* file must come first, and is followed by the constraints section, if any (some connection types are predefined). Then comes the network section, followed by the biases and sigmas sections, if required.

The Sections of the Network File

Each section of a *.net* file begins with the name of the section (all lowercase), followed by a colon (e.g., *definitions:*) on a line by itself. The section ends with a line containing *end*. The following paragraphs describe the details of the information required in each section and the format of that information.

Definitions. The definitions section is used to specify basic parameters of the network architecture. For **iac** and **cs**, the only parameter that must be defined is *nunits*. If you wish to read patterns from a file, you will also have to define *ninputs*; this can be done in the definitions section, in the *.str* file, or by hand anytime before the *get/ patterns* command is issued. For **pa** and **bp**, *nunits, ninputs*, and *noutputs* must be defined. Other variables may also be initialized in the definitions, although they can also be set in the *.str* file.

Definitions are given by placing the name of the variable and the value you wish to assign to it on a line by itself. Thus, the definitions section of the *jets.net* file used with the **iac** program looks like this:

```
definitions:
nunits 68
end
```

Constraints. The constraints section of the file is used to define the meanings of the characters that are used in the network part of the file to designate the weight types. Each constraint definition is given on a separate line, consisting of a lowercase letter called the *constraint character*, followed by a list of constraint attributes. Constraint attributes can be:

- *A floating-point number.* The initial value assigned to each weight designated with the constraint character. If no value is given, the initial value will be 0, unless otherwise specified by another attribute. For unmodifiable weights, the weight will remain at this value.

- *Positive.* The weight is constrained to have a nonnegative value. This constraint is imposed after every weight adjustment; weights with this constraint that go below 0 are reset to 0.

- *Negative.* The weight is constrained to have a nonpositive value. This constraint is imposed after every weight adjustment; weights with this constraint that go above 0 are reset to 0.

- *Random.* The weight is initialized to a random value in a range given by the parameter *wrange*. Positive random weights vary

between 0 and *wrange*; negative random weights vary between $-wrange$ and 0; otherwise the weight will vary between $+wrange/2$ and $-wrange/2$.

- *Linked.* The weight is constrained to have the same value as all other weights designated with this constraint character. This constraint is imposed at initialization and during each change to the weights. All the weights that are linked together are adjusted by an amount equal to the sum of the adjustments that would be made to each.

Several constraint letters are predefined. These are:

r Stands for random. The connection is set to a random value between $+wrange/2$ and $-wrange/2$.

p Stands for positive random. The connection is set to a random positive value between $+wrange$ and 0.

n Stands for negative random. The connection is set to a random negative value between $-wrange$ and 0.

. (The single character "."). This is the only character that is not an actual letter that can occur in the network specification itself. It specifies that the connection should be initialized to 0 and be treated as unmodifiable—that is, not subject to change through learning.

Note that positive, negative, and link constraints are only enforced in the **bp** program. In **pa** they are not used.

The constraints section of the *jets.net* file set up for use with **iac** looks like the following:

```
constraints:
u 1.0
d 1.0
v -1.0
h -1.0
end
```

Once this has been interpreted, it means that connections specified in the network section with the letters *u* or *d* will be assigned a weight of 1.0, and connections specified with letters *v* or *h* will be assigned a weight of -1.0. Though all the weights labeled *h* or *v* are assigned the same value according to this specification, it would be a trivial matter to dissociate the two by

giving *v*, say, a value of −2.0; this is in fact what we did in the *jets.net* file that is set up for use with the **cs** program.

Network. The network section of the file specifies which of the defined constraint characters applies to each of the connections in the network. This part of the file can come in either of two formats. The more elementary format consists of a full matrix of *nunits* rows of characters, each *nunits* long and containing no tabs or spaces. This is the format used in the *jets.net* file. In this format, the entry in a particular row-column location specifies the connection to the unit whose index is the row number from the unit whose index is the column number. Thus in the *jets.net* file, the connection in row 0 column 1 specifies a connection to the Jets unit (unit 0) from the Sharks unit (unit 1). This connection is marked as a *v*, which has been defined to have the value −1.0. This connection will therefore be assigned the value −1.0.

Note that weight values are assigned according to the network specification file when the *.net* file is processed. Random weight values are reassigned in programs that learn (**pa, bp, aa,** and **cl**) when a *reset* or *newstart* command is executed. Note that the weight values assigned in the network specification file can be overridden, either by setting individual weights using the *set/ weights* command or by reading in a file of weights using the *get/ weights* command.

The network specification for the *jets.net* file is rather large, so we will use a simpler example instead: the *cube.net* file used with the **cs** program. This consists of a set of 16 rows of 16 dots:

```
network:
................
................
................
................
................
................
................
................
................
................
................
................
................
................
................
end
```

The actual values of these weights are set by reading in the *cube.wts* file using the *get/ weights* command.

Letters in the network specification can be uppercase or lowercase. If the letter is lowercase, the corresponding weight is modifiable; if it is uppercase, it is fixed. Of course, no weights are modifiable in programs that do not learn; thus for **iac** and **cs**, the characters *v* and *V* are synonymous.

A more complicated format for the network specification file is also available. In this format, connections are specified in *blocks*. A block specification is simply a specification of a portion of the full conceptual matrix of *nunits* by *nunits* connections. The block specification specifies which subpart of the matrix the specification applies to, as well as the characteristics of the connections in the block. The subpart is specified by indicating which units receive the connections specified in the block and which units send these connections to these receivers.

A block specification begins on a line consisting of a % character. The block-specification line also contains four integers. These integers represent:

1. The index of the first receiving unit in the block.
2. The number of receiving units in the block.
3. The index of the first sending unit in the block.
4. The number of sending units in the block.

For example, in a pattern associator network, there will generally be some number of input units, each projecting to some number of output units. The following specification can be used to set up such a network, with 12 input units and 8 output units:

```
network:
% 12 8 0 12
rrrrrrrrrrrr
rrrrrrrrrrrr
rrrrrrrrrrrr
rrrrrrrrrrrr
rrrrrrrrrrrr
rrrrrrrrrrrr
rrrrrrrrrrrr
rrrrrrrrrrrr
end
```

The %-line specifies a block of connections coming into unit 12 and the next 7 units (for a total of 8 receiving units) from units 0 and the next 11 units (for a total of 12 sending units). The next eight rows, one for each receiving unit specified on the %-line, each consists of 12 *r*'s, one for each sending unit specified in the %-line. The *r*'s indicate that these weights should be initially random and modifiable.

Note that if all of the weights in a block are to be of the same type, you can specify this by putting the letter specifying the type immediately

following the %, *with no intervening space*. Thus the preceding example could be given more succinctly as follows:

```
network:
%r 12 8 0 12
end
```

Note that a network can consist of either a single, complete matrix of connections or of one or more block specifications. However, there is an important restriction:

A particular receiving unit *can only be specified in a single block*.

Finally, note that when block specifications are used, weights are only allocated for the connections actually specified in the blocks, and files of weights are assumed to contain values only for the weights that have been allocated. A detailed specification of the format of such files is given below.

Biases. The biases section of the file (applicable only to **pa, cs,** and **bp**), like the network section, can be given in either of two formats. In the simpler case, it consists of a row of *nunits* characters indicating the characteristics of the bias terms for all of the units in the network. Alternatively, biases may be specified in blocks, analogous to those used in the network specification. Block specifications consist of a line beginning with a %, followed by two integers that indicate the first unit in the block and the number of units in the block. The following line then gives a row of characters indicating the specification for each bias in the block; or if each bias is to be specified with the same character, the character may be given directly after the %. For example, random biases for the eight output units in the pattern associator network mentioned earlier could be specified in either of three ways:

```
biases:
...........rrrrrrrr
end

biases:
% 12 8
rrrrrrrr
end

biases:
%r 12 8
end
```

Note that when biases are in use, the programs allocate biases for all units in the network, even when only a subset of the biases are specified using block notation.

Sigmas. The sigmas section of the file is analogous to the biases section, though it is applicable to *harmony* mode in the **cs** program only. The values associated with the specification characters are taken to specify the value of the parameter *sigma* for units in a harmony network.

An Example Network File

Here we give a complete example of a *.net* file, taken from the *xor.net* file used with the **bp** program. It specifies a network with two input units, one output unit, and two hidden units, with initially random, modifiable connections from each input unit to each hidden unit and from each hidden unit to the output unit. It also specifies random, modifiable bias terms for the hidden units (units 2 and 3) and the output unit (unit 4).

```
definitions:
nunits 5
ninputs 2
noutputs 1
end
network:
%r 2 2 0 2
%r 4 1 2 2
end
biases:
%r 2 3
end
```

The first line in the network section specifies that the hidden units (units 2 and 3) receive connections from the input units (units 0 and 1) and that these connections are modifiable, with initially random connection strengths. The next line specifies that the output unit (unit 4) receives connections from the hidden units (2 and 3), which are also modifiable, with initially random values. The biases section indicates that the biases of units 2 through 4 are also modifiable and initially random.

WEIGHTS FILES

The *.wts* files can be read into or written out from all of the programs other than **ia**, which has no matrix of weights as such. The format of these files is governed by the network specification file, according to the following conventions.

The *.wts* file consists of a list of the weights in the network, followed by a list of the bias terms, if any. The list of weights can be thought of as consisting of a series of rows, one for each unit with incoming connections from other units. Rows are ordered by unit number, from first to last. Each row consists of one floating-point number for each connection to the receiving unit it applies to. These numbers specify the values of these connections, ordered by unit number, from first to last. Row elements must be separated from each other by any number of spaces, tabs, or newline characters. Thus, the order of entries is crucial, but spaces, tabs, and newlines[1] can be used freely for readability without affecting the way the file is interpreted. In fact, the *save/ weights* command places only a single weight on each line of the file it produces, even though the weights are conceptually grouped into rows. This is done to avoid the possibility of exceeding line-length limitations imposed by the file system on your computer.

In the simplest case, the *.net* file will specify a full matrix of *nunits* by *nunits* connections. In this case the list of weights in the *.wts* file will consist of *nunits* rows, one for each unit, and each row will contain *nunits* floating-point numbers, one for the connection of each unit to the row unit.

More complex cases arise when weights have been specified using block specifications. In this case, rows of weights are only given for units that actually receive connections from other units, and each row will only have entries for the connections specified in the block specification involved. Thus the *xor.wts* file, which is used in the XOR example in Chapter 5, contains only three rows of weights, one for each of the two hidden units and one for the output unit. Each row contains only two entries since each of these units receives a connection from only two other units.

Following the list of weights comes the list of biases, if any. Note that if a biases section is present in the *.net* file, a full list of *nunits* biases is expected in the *.wts* file, even if only a subset of the bias terms were actually specified in the *.net* file. Each bias is a floating-point number, separated from others by spaces, tabs, or newlines.

An Example Weights File

As an example of a *.wts* file, we give here the file *xor.wts*. It was created using the *save/ weights* command and is read into the **bp** program using the *get/ weights* command, after the *xor.net* file has been used to set up the network. Unspecified biases are simply stored as 0.00000. Here is the file:

---

[1] The *newline* character is the character in a file that separates lines from each other.

```
0.432171
0.448781
-0.038413
0.036489
0.272080
0.081714
0.000000
0.000000
-0.276589
-0.402498
0.279299
```

Note that the weights file contains no special indications about where the rows of weights end or even where the weights end and the biases begin. This is because the number of conceptual rows of weights, the number of weights per row, and the number of biases is determined strictly by the specifications contained in the *.net* file.

TEMPLATE FILES

The template file is used to specify the appearance of the display screen and the way in which various display objects, called *templates*, will appear on the screen. The file consists of an optional *layout*, followed by a series of *template specifications*. The layout is used to set up the background on which the various templates will be displayed and to specify where the templates will occur in the background. If the layout is omitted, template locations can be specified directly. We will first describe the format for the layout, then describe the template specifications.

The Layout

If there is a layout, it must occur at the very beginning of the template file. The first line of the file must be

```
define: layout
```

The first line can also contain two integers indicating the number of rows and columns to use on the screen. By default, the program assumes it can use 23 rows (lines) and 79 columns. This is one row and column less than the typical screen size, to make it easy to review screen dumps simply by

using the *type* command in MS-DOS. Twenty-three lines of 79 characters will fit on the screen without scrolling off the top. To change this default, give the desired number of rows and columns after the word *layout*. For example, to increase the size of the display to 47 by 131, the first line would be

```
define: layout 47 131
```

Subsequent lines are taken as containing literal characters to plot directly to the screen and $'s, which specify where templates are to be inserted. No tabs are allowed in the layout; you must use spaces as separators, although it is perfectly acceptable for lines to be blank or less than 79 characters long. The layout thus gives an exact image of what the screen is to look like. The end of the layout is indicated by a line containing the single word *end*. The layout is printed to the screen starting on line 5, and may be up to $nr - 5$ lines long where *nr* is the number of rows. Five rows are reserved for the command line at the top of the screen and the help area just below it.

An Example Layout From a Template File

The layout for the cube example used with the **cs** program (in the file *cube.tem*) is specified as shown in Figure 1. The layout is read from top to bottom and, within each row, from left to right; the locations of the $'s are stored in the order encountered. Successive $'s in the layout are numbered starting from 0. Below we explain how the $'s are paired up with templates.

The Template Specifications

Each template specification consists of a number of entries, or *template specifiers*, separated by white space (spaces, tabs, or newlines). To illustrate, we will describe the template specification from the *cube.tem* file that is responsible for putting the value of the variable *cycleno* in the correct place on the screen. The specification is

```
cycleno variable 1 $ 2 cycleno 3 1
```

The entries correspond to the following specifiers. The specifiers are listed in the order encountered, with their values from the *cube.tem* example given in parentheses.

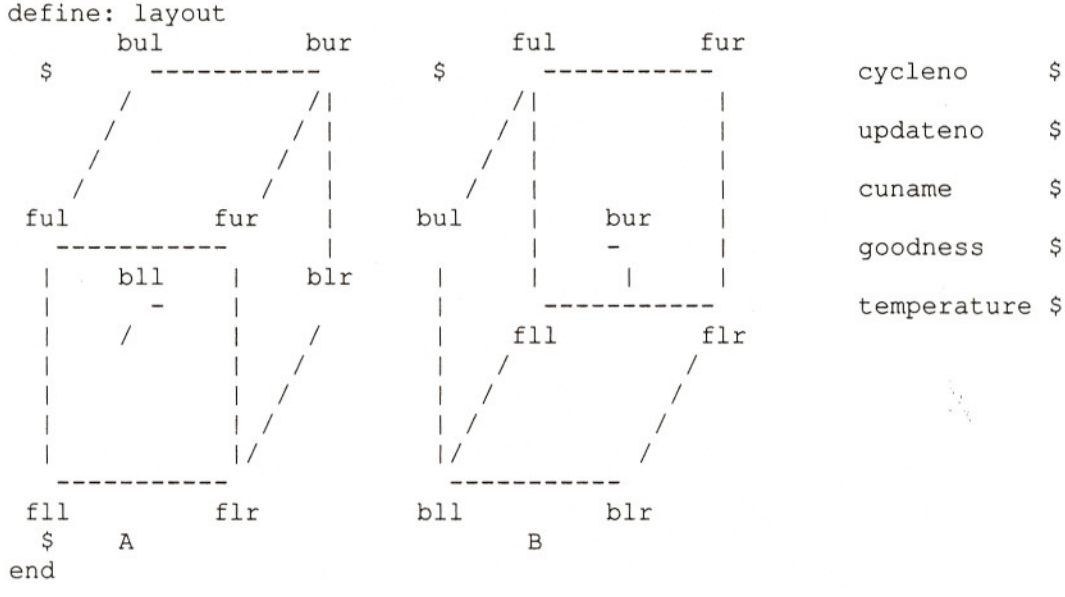
define: layout
bul
bur
$
ful
fur
bll
blr
fll
flr
$
A
end
ful
fur
$
bul
bur
fll
flr
bll
blr
B
cycleno $
updateno $
cuname $
goodness $
temperature $

FIGURE 1.

TNAME (*cycleno*)

The name for the template, or display object, itself. This name can be used as an argument to the *display* command, and will cause the template to be displayed on the screen. Thus to display this template you would enter

display cycleno

Note that a number of templates can have the same name. If such a name is given to the *display* command, all of the templates with that name will be displayed together.

TYPE (*variable*)

The template type of the template. There are a number of different template types, which will be described in the next section. The *variable* template type is used for most single-valued variables such as *cycleno*.

DLEVEL (1)

The display level of the template. The display level is an integer that determines whether the template will be displayed when the screen is updated by the program. There is a global *dlevel* variable and a *dlevel* value associated with each template. Templates are updated if their specific *dlevel* is less than or equal to the global *dlevel*. However, templates whose specific *dlevel* is 0 are treated specially. Generally, these are templates that display information that is not expected to change in the course of processing. These templates are only displayed if the screen has been cleared since the last time the screen was updated.

POS ($ 2)

This compound entry specifies the row and column location of the upper left-hand corner of the screen region in which the template is to be displayed. The "$ 2" notation is used to specify which $ in the layout should be taken as the one indicating the upper left corner of this template. Note that $'s are numbered starting with 0, in the order they are encountered in processing the layout, as specified above. An alternative notation is "$ *n*," where the *n* stands for *next* (note that *n* is used literally here). This notation is taken to mean the next $ after the one specified in the previous template specification. If no templates have already been specified, *n* is taken to mean 0. As a third alternative, the compound entry can be two integers. These are taken to indicate directly the row and column number of the upper left corner of the template. Note that the row number must be 5 or greater and must be less than the number of rows.

VNAME (*cycleno*, second occurrence)

The name of the variable as it is known in the program. Note that this is not the same as the name of the template, mentioned above;

two templates with different names can display the same variable. In general, any of the variables that can be accessed by the *set* and *exam* commands can also be accessed in templates.

SPACES (3)
: The number of spaces the program is to use in displaying this template.

SCALE (1)
: A floating-point number that specifies a scale factor that is multiplied with the value of the number to be displayed. This is particularly useful for floating-point variables that are to be displayed in small fields without decimal points.

Other specifiers are more specific to particular template types. These are defined in discussing each template type below.

The Template Types

Template specifications always begin with the TNAME, followed by the TYPE. The following list describes the different types that are available, together with a list of the specifiers that must come after the TYPE. Note that for each template, *all* of the specifiers must be present in the order indicated. Remember also that the POS specifier is a compound specifier, consisting of two entries separated by white space.

label (DLEVEL POS ORIENT SPACES)
: The TNAME of this template is taken as a literal label to be displayed either horizontally or vertically, as determined by the value of ORIENT (*h* or *v*). The label is truncated if it is longer than SPACES characters.

variable (DLEVEL POS VNAME SPACES SCALE)
: The single-valued variable VNAME is displayed. The variable may be an integer, a floating-point number, or a character string. Integers and floating-point numbers are multiplied by SCALE before the value is displayed. In either case, the value is truncated to an integer. Thus a variable whose internal value is 2.67 will display as a 2 if SCALE is 1, as 26 if SCALE is 10, and as 267 if SCALE is 3. Negative numbers are generally displayed in reverse video. Numeric values that would take up less than the available number of spaces are right-justified; this is typically appropriate for numbers. Values that would take up too much space are displayed according to the conventions described in Chapter 2. For character string variables (e.g., pattern names), if the value of the SPACES field is negative then the string is left-justified. Thus "−5" means that five spaces are allocated and that the string always begins at the

left end of the region. SCALE is not used for character strings, but a value must be specified anyway.

floatvar (DLEVEL POS VNAME SPACES SCALE)
The value of the scalar floating-point variable VNAME is displayed to four decimal places, in standard floating-point notation. The value is multiplied by SCALE before displaying and is right-justified within the SPACES indicated. If the value is too big, it is truncated on the right so that it fits the available SPACES.

vector (DLEVEL POS VNAME ORIENT SPACES SCALE START NUMBER)
NUMBER elements of the vector VNAME are displayed, starting with element START. The vector may consist of integers, floating-point numbers, or character strings. For example, a list of unit names can be displayed as a vector template. If ORIENT is *h*, the elements of the vector are displayed across the same line. If ORIENT is *v*, the elements are displayed starting in the same column, on successive lines. In both cases multiple characters making up the same element are displayed horizontally. For numeric variables, each element is multiplied by SCALE and then truncated to an integer, as described above for *variable* templates. For vectors of character strings, SCALE (which should have an integer value in this case) is interpreted as a specifier indicating the number of blanks to place at the beginning of the string. This allows the user to put blanks between the successive elements of a string vector when displayed horizontally.

label_array (DLEVEL POS VNAME ORIENT SPACES START NUMBER)
This template is similar to the preceding one, but it is used only for vectors of character strings or labels, and it allows strings to be displayed horizontally or vertically. The value of the ORIENT specifier applies to the orientation of the individual strings. If ORIENT is *h*, each individual string is shown horizontally, and successive strings are shown on successive lines. If ORIENT is *v*, each individual string is shown vertically, and successive strings are shown in successive columns.

look (DLEVEL POS VNAME SPACES SCALE SEPARATION LOOKFILE)
This template type can be used with either vector or matrix variables. It consults the LOOKFILE (which must be present in the directory the program is being run in) for a specification of a rectangular grid of locations in which to display elements of the vector or matrix. Each element is multiplied by SCALE and then truncated to an integer, as described above for *variable* templates. SPACES character positions are allowed for each element; adjacent elements are SEPARATION spaces apart. Thus if SPACES is 2, SEPARATION is 4, and SCALE is 10, three adjacent elements with the values 1, 0.5, and 7 would appear as

```
10    5   70
```

label_look (DLEVEL POS VNAME ORIENT SPACES SEPARATION LOOKFILE)
This template type is similar to the previous one, but is used with arrays of strings or labels, such as unit names and pattern names. ORIENT is *h* or *v*, indicating whether individual strings should be displayed horizontally or vertically.

matrix (DLEVEL POS VNAME ORIENT SPACES SCALE ST_ROW N_ROWS ST_COL N_COLS)
A section of the matrix VNAME is displayed. The section begins in row ST_ROW and column ST_COL, and is N_ROWS by N_COLS in size. Note that when the matrix is a weight matrix, the row indexes correspond to the units receiving the connections and the column indexes correspond to the sending units. Thus to show the weights for connections to units 4 through 7 from units 0 through 3, ST_ROW would be 4, N_ROWS would be 4, ST_COL would be 0, and N_COLS would be 4. Also note that arrays of vectors are essentially matrices. Thus, for example, *ipattern* is a matrix variable, consisting of *npatterns* rows, each *ninputs* long. In this case, then, the row indexes correspond to patterns and the column indexes to elements of the patterns. The ORIENT field specifies the orientation of the rows of the matrix. Thus if ORIENT is *h*, the rows are displayed horizontally and the columns are displayed vertically. If ORIENT is *v*, the rows are displayed vertically and the columns displayed horizontally. This is equivalent to transposing the matrix.

An Example List of Template Specifications

As a full example, the template list from the file *cube.tem* is shown here:

```
cube1       look     1 $ 0 activation  1 10 1 cube1.loo
cube2       look     1 $ 1 activation  1 10 1 cube2.loo
cycleno     variable 1 $ 2 cycleno     7  1
updateno    variable 1 $ 3 updateno    7  1
uname       variable 1 $ 4 cuname     -7  1
goodness    floatvar 1 $ 5 goodness    7  1
temperature floatvar 1 $ 6 temperature 7  1
weight      matrix   5 $ 0 weight  h   4 10 0 16 0 16
weight      vector   5 $ 2 uname   v   6  1 0 16
weight      vector   5 $ 7 uname   h   4  1 0 16
```

The POS specification consists of a $ followed by a number. Each template is, therefore, displayed at the location of the $ whose index is given by the

second part of the POS specification. Most of the different template types are illustrated here, as well as several useful tricks. For example, there are three templates called *weight*. When the user enters *display/ weight*, all three are displayed simultaneously. In this way, it is possible to put together "macrotemplates" from several ordinary templates.[2] Another trick is the use of the SCALE specifier with vectors of character strings. This occurs in the last entry, which specifies a horizontal array of unit names. The SPACES field indicates that 4 characters are allocated to each unit name, but the SCALE field indicates that each unit name is to be left-padded by a blank. Thus the four spaces are occupied by a blank, followed by the first three characters of the unit name.

LOOK FILES

The look files specify where in a rectangular array the elements of a vector or matrix are to be displayed. The format of the look file is as follows: The first line contains two integers, indicating the number of lines of entries and the number of entries per line. Below this, there are as many lines as specified in the first argument, each containing as many entries as indicated by the second argument.

When the object being displayed is a vector variable, the entries may be integers, in which case they are taken to be the index specifying the element of the vector that is to be printed at the corresponding location in the display. Alternatively, entries may simply be single dots ("."), which are taken to indicate that nothing is to be displayed at this location.

An Example Look File

The file *cube1.loo* is shown here as an example. This look file is used by the first template that was specified in the *cube.tem* file to indicate where, with respect to this template's upper left-hand corner, the 0th through 7th elements of the activation vector are to be displayed. Entries in the look

---

[2] A variant of this involves naming a group of templates with names that begin the same, but end differently. This way if the user enters the initial part that is shared, all of the variants are displayed, but if the user enters enough of the string to specify the template uniquely only the unique template that matches the full string entered is displayed. For example, the environment, including the set of *pnames* and *ipatterns* (and possibly *tpatterns*) is often specified by templates called *env.pname*, *env.ipat*, and *env.tpat*.

file itself must be separated by white space (spaces, tabs, or newlines), but these do not affect the actual appearance of the display. The spacing of the display itself is controlled in part by the SEPARATION variable associated with the template; this variable indicates the number of successive spaces between look elements along the same line.

```
15 19
. . . . . 0 . . . . . . . . . . 1
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
2 . . . . . . . . . . 3 . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . 4 . . . . . . . . . . 5
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
6 . . . . . . . . . . 7 . . . . . .
```

An Example Look File for a Matrix Variable

When using look files with matrices (such as weight matrices), each non-dot entry specifies the row and column in the underlying internal matrix of the element to be displayed at that location. Row and column numbers are separated by a comma. As an example, the following look file is used to specify a layout for the weights coming into each of two hidden units from each of sixteen input units. This is the file *16wei.loo*, used with the **cl** program. The file lays the weights out in two square arrays, one for the connections coming into one of the units and one for the connections coming into the other. The column of dots down the middle separates the two arrays.

```
4 9
 16,0  16,1  16,2  16,3 .  17,0  17,1  17,2  17,3
 16,4  16,5  16,6  16,7 .  17,4  17,5  17,6  17,7
 16,8  16,9 16,10 16,11 .  17,8  17,9 17,10 17,11
16,12 16,13 16,14 16,15 . 17,12 17,13 17,14 17,15
```

PATTERN FILES

Pattern files are of two kinds. One kind contains a list of input patterns; the other contains a list of input-target pattern pairs. The first kind is used with **iac**, **cs**, **aa**, and **cl**. The second kind is used with **pa** and **bp**. We describe the input pattern type first.

Files for Lists of Input Patterns

Files containing lists of input patterns consist of a sequence of patterns. Each pattern consists of a name followed by *ninputs* entries specifying the values of the elements of the pattern. The entries are separated by white space (spaces, tabs, or newlines).

Each name is a string of characters beginning with a letter; pattern names should not begin with any of the digits. By convention we use the letter *p* as the first letter of the name of any pattern which for mnemonic reasons would begin with a digit.

The entries specifying the values of elements of the pattern are treated as floating-point numbers. The following special single-character entries are recognized:

. (dot) is assigned the value 0.0.
+ (plus) is assigned the value +1.0.
− (minus) is assigned the value −1.0.

Files for Lists of Input-Target Pairs

These files contain a sequence of pattern pairs. Each pair consists of a name, *ninputs* entries specifying the elements of the input pattern, and *noutputs* entries specifying the elements of the output pattern. Entries are separated by white space as in input pattern files. There is no special separation between input and target patterns. Both input and target entries are treated as floating-point numbers, with the same special characters recognized as in input pattern files.

In **bp**, negative entries have special meanings that are different for input pattern elements and target pattern elements. A negative input pattern element is interpreted as an instruction to set the activation of the corresponding input unit to *mu* times its previous activation plus the activation of the unit whose index follows the minus sign. A negative target pattern element is interpreted as an instruction to ignore the output generated by the corresponding output unit. The error for this unit is set to 0.

Finally, the reader should note that in **bp**, the parameter *tmax* is used to modify target elements specified with entries of 1 or 0. If the entry is 1, the target is set to *tmax*, and if the entry is 0, the target is set to $1-tmax$. By default *tmax* is set to 1.0, but sometimes it is useful to use a value like 0.9. In this case, if the entry is 1, the target is set to 0.9, and if the entry is 0, the target is set to 0.1.