```
    n = 0; unit = start
    repeat nunits
        list[n] = unit
        n += 1 ; unit += 1
    endrepeat
end
exec makeList(updateHid,nIn,nUnit-nIn-nOut)
exec makeList(updateHidOut,nIn,nUnit-nIn)

procedure reset                # symmetrize weights
    network reset              # reset weights/biases
    scalar n; n = 0
    repeat nUnit
        weights[n] = weights[][n]    # copy nth column into nth row.
        n += 1
    endrepeat
end
exec reset
```

First, we need to setup the update lists `updateHid` and `updateHidOut` to contain the appropriate indeces. For this, we define a procedure `makeList` that takes three arguments `list`, `start`, and `nunits` and sets the elements of the vector `list` to `units` consecutive numbers starting from `start`. Then we execute this procedure to set the elements of `updateHid` to the indeces for the hidden units, and the elements of `updateHidOut` to the indeces for the hidden and the output units.

The last procedure `reset` simply makes the weight matrix in `weights` symmetric which is assumed in the simulated annealing process. `network reset` is a **PlaNet** command that initialize all the weights and the biases in the network with random numbers.

2.7 Defining Patterns in a Pattern File

Input and target patterns to be given to the network are stored in a file called the pattern specification file, or pattern file for short, and are used by reading the file into the system. Patterns can be defined either with special digits or floating point numbers.

2.7.1 Digit representation of patterns

By default, patterns are represented as strings of coded digits. Each line in a pattern file consists of two strings of contiguous digits separated by white space(s). The first string represents an input pattern and the second represents the corresponding target pattern. Each digit specifies the activation level for a unit: The digit 0 corresponds to the minimum activation level of the network, and the digit `a` (for decimal 10) the maximum activation level. The other digits (1,2, . . ., ,9) correspond to intermediate activation levels. The default maximum value is 1.0, and the minimum is 0.0, so that the digit 1 corresponds to the value 0.1, 2 to 0.2, and so on. Actually, the maximum/minimum activation levels are parameters you can set (more on setting parameters later), and the actual values in the patterns are computed relative to the values of these parameters. Note that whenever you change either of these parameters, the pattern file needs to be read in again because of this dependency.

An example of a pattern file is:

```
0000    a
037a    0
a0a0    0
22aa    0
0a00    0
```

```
0a5a    a
0aa0    a
. . .   .
```

where there are four values for each input pattern and one value for each target pattern (this is the symmetry problem in which the target is on only when the input is symmetric around the center). Optionally, a third column in a pattern file can contain *labels* for the patterns. Each label is an arbitrary character string and will be used when the response of the network to the pattern is displayed on the screen or printed.

2.7.2 Floating-point representation of patterns

It is possible to specify patterns with floating-point numbers if more precision is necessary than the digit representation allows. To do this, set the parameter *floatpattern* to *on* (see the next section for how to set a parameter) before reading in the pattern file. (**PlaNet** may set this automatically if it suspects that you have forgotten to do so.) The first line in a floating-point pattern file should contain two integers indicating the lengths of the input and the target patterns, respectively. An optional third integer (any number) indicates that the file contains a label for each pattern pair. If the input and target lengths are N and M, then the next N strings in the file are interpreted as N floating-point numbers representing the first input pattern, the next M strings the first target pattern, the next string the first label if the third integer is present on the first line, the next N and the next M strings the second input and the second target patterns, and so on until the end of the file. Note that the floating-point numbers are interpreted as they are, not relative to the maximum/minimum activation levels as is the case for the digit representation of patterns.

Chapter 3

PLANET INTERFACE

3.1 Interacting with PlaNet

You interact with **PlaNet** through a command-based interface. There are currently about 50 commands, most of which will be explained in this chapter. This section explains how to use some basic commands to setup a network and patterns, and then to run the network. Section 3.2 “Looking into the Network” shows various ways of graphically examining network states on the screen. Examples are based on a simple 3-layer backpropagation network, but the methods described in this chapter are very general and can be used for any types of networks.

As mentioned in Section 1.1, the **PlaNet** system has a few different versions of an essentially same program: **XNet** for the X-window environment; **StarNet** for any UNIX environment without graphics; and **SunNet** for the SunView (suntools) environment (**SunNet** is not officially supported). In this chapter **PlaNet** refers to one of these programs that is actually being used.

3.1.1 Shell interface for PlaNet

Newer versions (after Version 5.5) of **PlaNet** is designed to receive commands from a shell environment. That is, **PlaNet** itself is run in the background and all commands are typed into a shell, such as C-shell (csh) or K-shell (ksh). This allows you to stay in the shell environment while interacting with **PlaNet** and use all the facilities the shell provides, such as alias, history mechanism, controls like *if, foreach* (csh), *while, function* (ksh) etc., and shell variables. You can even write shell scripts to control complex experiments, as shown in Section 3.3.1. Currently, facilities are provided for using either C-shell or K-shell as the interface.

> [*Comments: Things to be aware about shells*] Note that characters such as ‘>’, ‘*’, ‘|’, ‘!’, ‘&’, and ‘$’, have special meanings for the shells, so you need to quote any arguments containing these characters.
>
> You might encounter some conflicts between **PlaNet** commands and regular shell commands. For example, `set` used to be a **PlaNet** command but it is also a shell command. Since `set` tends to be used often in aliases that are set up automatically, making `set` a **PlaNet** command was likely to cause unexpected conflicts. To avoid this problem, the **PlaNet** command `set` has been changed to `nset`. Similar problem may arise with `echo`, `print` and some other **PlaNet** commands. If you want to use them as shell commands, you should either precede them with a backslash as in **\print ...** or use the full pathnames as in **/bin/echo ...** etc.. Alternatively, you can edit the file “netcsh” (“netkorn” if you use ksh) to change the name of a **PlaNet** command, for example, `necho` instead of `echo` (see the definition of `nset` in this file).
>
> **PlaNet** usually keeps track of where you are in the file system. If it loses track of where you are (e.g., it cannot find a file in the directory you are in), type `netcd`. [1]

[1] It is no longer necessary to use `chd` instead of `cd` when changing directories.

3.1.2 Starting up

The easiest way to start a **PlaNet** session is to use a shell program **RunNet**. **RunNet** decides which program to run depending on your environment (machine architecture and window system), establishs a communication between **PlaNet** and the shell, and set up your shell so that you can type in **PlaNet** commands into it. To use it, just type:

```
source RunNet
```

if your shell is C-shell, or

```
. RunNet.ksh
```

if your shell is K-shell.

If your window system is X, start it up from any window running a shell, making sure you have the DISPLAY environment set correctly. If your window system is SunView (suntools), do this in a Sun Graphics Tool Window which can be opened by selecting "gfxtool" in the Root Menu.[2]

> [*Comments: Customize* **PlaNet** *with an initialization file*] If there is a file named *.netinit* in the same directory you start **PlaNet** in, **PlaNet** will interpret and execute the commands in the file before starting. This can be used to customize default values for some parameters, etc.

In X, you may be asked to place the graphics window opened up automatically on the screen. This window will be called the **PlaNet display area** and will be used to display various network states. You can change the window size but the automatic scaling of display objects has not been implemented in **XNet**. **XNet** understands some of the standard user preferences such as the color of the background (flag **-bg color**), the foreground (flag **-fg color**), the window geometry (flag **-geom geometry**), and the font (flag **-font fontname**), which can also be specified in the **.Xdefaults** file in your home directory.

In SunView, the graphic (lower) portion of the window will be cleared and a large rectangular box will appear. This serves as the **PlaNet display area**. If you change the size of the graphics window, this box will adjust its size (and the size of all objects displayed in it) proportionally to fit the new window size.

When you are done with a **PlaNet** session, you can terminate **PlaNet** by typing `quit`. To start up another session in the same shell, just run **XNet**, **StarNet** or **SunNet** in the background.

3.1.3 Getting help with `help` and setting parameters with `nset`

Now you can type in **PlaNet** commands to the shell. Type `help`. **PlaNet** will print all the command names. The syntax and an explanation about the use of a command can be obtained by typing `help <command name>`. Try `help help`. (Text to be typed by the user will be printed in this `typewriter` font throughout this chapter.)

The command `nset`, with no arguments, prints all the parameters that can be set by the user with a short description and the current value of each (try this now — too see all of them you may need to do `nset | more`.) A parameter can be set by `nset <parameter name> <parameter value>`. You can give multiple pairs of parameter names/values as the arguments to `nset`. To print the value(s) of some parameter(s), use the command `whatis` with the parameter name(s) as the argument(s). Some useful parameters are **eta (learning rate)** and **alpha (momentum)** used for backpropagation learning. Try, for example:

```
nset eta 0.2 alpha 0.4
whatis eta alpha
```

The **PlaNet Reference** has description of all these commands and parameters.

[2] If **RunNet** fails, see the **README** file for how to start up a **PlaNet** session.

[*Comments:* `nset` *is versatile!*] The command `nset` can actually be used to set almost any objects in **PlaNet**. Besides the **PlaNet** parameters, `nset` can set any variable (including scalar, vector, or matrix variables) defined in the network file to values of any other variable or to a constant. In fact, you can set a variable to any expression (properly evaluated), such as described in the first chapter by typing `nset <vector> <expression>`. Type `help nset` or see the command `nset` in section I-B of the **PlaNet Reference** .

3.1.4 Setting up a network and patterns with `network` and `pattern`

Typically, the first thing you do in a **PlaNet** session is to set up a network and training patterns by reading in a network file and a pattern file whose formats were described in Chapter 2. The command **network** reads in a network file and the command **pattern** reads in a pattern file. Both take a file name as an argument. Let us use here the 4-2-4 backpropagation network (which we saw in Section 2.3.1) defined in the file **example/net/n.4.2.4** and the patterns for an auto-encoding problem defined in the file **example/pat/p.encode4**. They can be read in by the following.

```
cd example
network n.4.2.4
pattern p.encode4
```

[*Comments: What is an auto-encoding problem?*] In an auto-encoding problem, the desired output is the same as the input: when the network is trained on a set of patterns, it needs to develop internal representation (or features) in the hidden units that can *encode* all the patterns and then to decode them out into the output units.

[*Comments: Compressed files*] To save space you may wish to use the "compress" or "compact" program to reduce size of a large pattern file. **PlaNet** will automatically "uncompress" a file with `.Z` suffix and "uncompact" a file with `.C` suffix. This applies to most types of files that **PlaNet** reads.

To see if everything in the network file is now in **PlaNet**, type `names layer connection procedure`. This will print the names of and the number of elements in all the layers and connections as well as the procedure names. You can also see a listing of the actions in a procedure by `listproc <procedure name>`. Try `listproc learn`, for example.

Choosing network definitions on-line

As mentioned in Section 2.2.1, the file **n.4.2.4** contains only the definition of the network size as string definitions such as **define Nin 4**, etc.. The actual network definition is contained in another file **n.3layer** in which these defined strings are used as parameters specifying the network size. This file is loaded in when we read **n.4.2.4**. We can achieve the same effect by specifying the necessary string definitions on a command line. Thus, `network Nin=8 Nhid=3 Nout=8 n.3layer` would set up a same 3-layer network with 8 input, 3 hidden, and 8 output units. Note that the parameters should be specified before the network file name.

Adding network definitions

If you need to load additional definitions of variables or procedures written in separate files, say *file1* and *file2*, use `network + file1 file2` to read them in. The plus flag + means that these are additional definitions. Without the + the current network will be deleted before reading the files.

You can add network definition on-line without reading from a file by typing:

```
network -c <network command>
```

where `<network command>` can be any command that can be used in a network file such as `layer`, `define`, or `procedure`. To define a new procedure on-line, type

```
network -c procedure <procedure-name>
```

(this can be followed by argument declaration — see Section 2.5.4) and then type in the procedure definition body which should be ended with `end`.

3.1.5 Running and training the network with `present` and `cycle`

Once a network file and a pattern file have been read into **PlaNet** the network can be "run." Running this network involves two steps: (a) presenting a pair of input/target patterns to the network (i.e., putting them into the input/target buffers) and (b) executing a procedure defined in the network file. The command `present` does these steps for a single input/target pair. The command `cycle` cycles through all pairs of input/target patterns and executes a procedure for each pair, and repeats this process for a given number of times. `Cycle` is usually used to train the network and `present` to test the network's performance. For example, try:

```
present 0
```

It presents the input/target pattern 0 and executes a procedure. The default procedure is the one previously used, or *activate* initially. To specify a procedure other than *activate*, type **present 0** `<procedure>`. Try `present 0 test`, for example. (**PlaNet** will complain that the procedure *test* is not defined.) Try `present 1, present +, present -, present 1-3, present all`, and so forth. **PlaNet** will print the error for each pattern presented. The error for each pattern is computed by averaging the square errors across the output units, where the output units are defined as those units that are compared against the target pattern.

Your network has not been trained, so the errors printed are probably large. So, let us start training the network. Type

```
cycle 100
```

PlaNet will cycle through all pairs of patterns 100 times, executing the procedure *learn* after presentation of each pair (the default procedure for the command **cycle** is initially *learn*, and the previous procedure thereafter). After each cycle, **PlaNet** prints a line that looks like:

```
epoch=57 error= 0.085257
```

indicating how many learning epochs have been run, and the overall error averaged across all the patterns. (This error is thus the average single unit squared error averaged across all the output units and across all the patterns.) The parameter **print** specifies how often this information should be printed. Type `nset print 10`, and try `cycle 100` again. **PlaNet** will now print the error information at every 10 cycles. If *print* is set to zero, nothing will be printed. The default value for *print* is 1.

By default the patterns are cycled through in the order they are given in the pattern file. However, if the parameter **randomize** is set to *on* (default=*off*) the order of pattern presentation is randomized — the patterns are randomly permuted before starting each cycle.

> [*Comments: Interrupting a command*] If, for any reason, you wish to interrupt a command that can take a long time to complete such as **cycle**, hit your usual "interrupt" key on the keyboard. (This is often *Ctrl-C*.) The command will stop execution after a while. An alternative, and often quicker way to interrupt a command is provided in **XNet**: just click a mouse button in a small square at the upper-left corner of the display area. In either case, you will be asked to choose, among several options, at what level to interrupt. [3] Type `help interrupt` or see the section on "interrupt" in **PlaNet Reference** for more details.

[3] The interrupt method has been changed from earlier version in which there were only one or two options. The current method gives between one and six options depending on what you are doing at the moment.

[*Comments: Debugging a procedure*] Normally, the state of the network can be examined only between procedure executions. However, it is sometimes useful to examine network states in a middle of a procedure. This can be done, whether debugging or not, by executing the procedure in *debug-mode.* In this mode, you are allowed to execute the actions one by one, and also to execute any **PlaNet** commands between actions. Debug-mode can be entered either by setting the parameter **debug** to **on** before executing a procedure or by choosing the (a)ction option after interruptting the procedure. In addition to normal **PlaNet** commands, some debug commands are available in this mode. Type `help debug` or see the section on **debug** in section I-C of the **PlaNet Reference** for details.

3.1.6 Plotting learning curves with `graph` and `glist`

While running the network with **cycle**, it is useful to graph the learning curve, that is, error as a function of learning epochs. Reinitialize the network (clear all units and randomize the weights) and initialize a graph by typing:

```
network reset
graph new
```

The `graph new` command will prompt you to specify some necessary information for graph initialization. Just hit `return` after each prompt to get the default graph (X ranges from 0 to 300 [the current epoch number and 300 epochs plus that], and Y ranges from 0 to 1.0 [error]). After a graph is initialized, the average square error is plotted in the graph at every N learning cycles, N being specified by the parameter **graph** (default = 1). Type `cycle 300` and watch how the error decreases. (Set *graph* to 0 if graphing is not desired.) More than one learning curve can be plotted in the same graph. Initialize the network (**network reset**), and change some learning parameters, say, *eta* (the learning rate) to 0.1 and *alpha* (the momentum factor) to 0.2, and run the network again with `cycle 300`. This network might converge more slowly with these parameter values than the default (eta = 0.2, alpha = 0.9) — compare the two learning curves in the graph. A graph can be erased, cleared, extended, moved, closed, opened, or labeled. Type `help graph` for details.

It is possible to graph any value in the network while running the network with **cycle**. For example, in the graph shown in Figure 3.1 the error for each of the patterns, as well as the overall error, were plotted as functions of learning epoch. To do this, type: [4]

```
glist '$err[0]' 0; glist '$err[1]' 1; glist '$err[2]' 2; glist '$err[3]' 3
```

to plot the errors for the patterns 0, 1, 2, and 3. (The first agruments are quoted because the character $ has a special meaning for the Shell. The second argument is the character to be used as the marker for the plotting of the value. Type **help glist** or see the command *glist* in the **PlaNet Reference** for details.)

[*Comments: List commands of* **PlaNet**] The command *glist* adds an item to the **graph list** in which **PlaNet** keeps a set of items to be plotted in the graph. **PlaNet** keeps a number of other kinds of lists too: **display list**, **print list**, **save list**, **plot list**, and **command list**, all of which will be explained in later sections. These lists are convenient because they allow you to store a set of operations and to perform all or some of them repeatedly. These lists are manipulated with different commands, but these commands share a generic list command syntax for adding and removing items and turning items on and off. Type **help list** or see section I-L "List Commands" in the **PlaNet Reference** for details.

Type:

```
network reset; graph new; nset mark 50; cycle 200
```

[4] The semicolon character ; are used instead of a *carriage return* to separate commands.

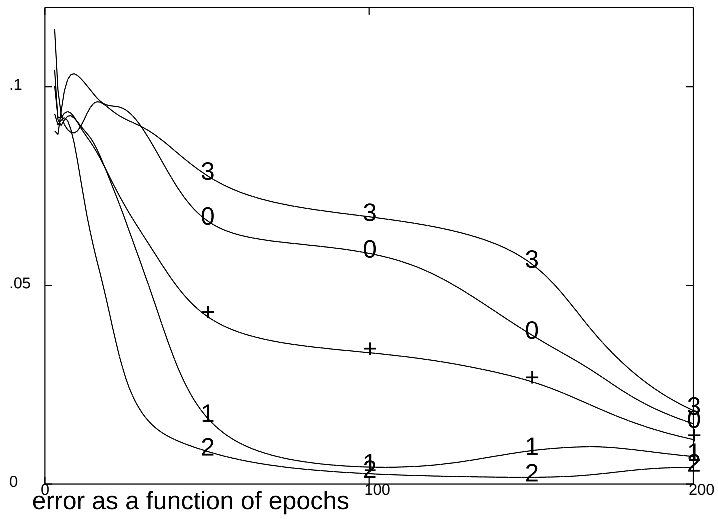

Figure 3.1: Learning curves plotted in a graph. Errors for the four patterns, marked from 0 to 3, and the average error, marked +, are plotted in the graph as functions of learning epochs while running the network with the command `cycle`. Any values in the network can be plotted by putting them into the graph list. The graph was labeled using `graph label`.

and observe how the error for each pattern decreases. If more training cycles are necessary, use `graph extend` to extend the graph to, say, 400 epochs. The command `graph extend` can be used to adjust both the range of X (epochs) values and the range of Y (error) values. To cycle up to epoch 400, type `cycle -400`.

3.1.7 Saving weights with `save` and retrieving them with `read`

You can save the values of connection weights in a network to a file. The command `savefile <file name>` opens a file for subsequently saving the weights. The parameter **save** specifies how frequently the weights should be saved: if *save* is set to N (> 0), weights will be saved at every N cycles (no saving if $N = 0$, which is the default). Weights are appended to the end of the file each time, rather than overwriting it, so be warned that saving weights of a large network too frequently will create a very large file. (If the parameter **saveappend** is *off* (default = *on*) new weights are written over the previously saved weights.) The command `savefile close` closes the current file. It is possible to save just the average error value without saving the weights for the whole network. To do this, set the parameter **save** to 0, and set the parameter **errorsave** to a positive integer that specifies how often to save the error.

The weights can be saved at any time simply by typing `save`. It is also possible to save a variable by the command `save <variable>`. Variables can be saved periodically by specifying the *save list*, a list of vectors to be saved whenever the weights are saved into a file. Type `help savelist` or see the command `savelist` in the **PlaNet Reference** .

Let's try saving weights in a file. Set *save* to 100, and *errorsave* to 10, for example. Then reset the network, open a save file, clear the graph and run, say, 200 cycles.:

```
network reset; savefile w.4.2.4.a        # or whatever file name you like
graph clear; cycle 200
```

The error and the weights now have been saved several times. Close the file by `savefile close`.

A file created with `savefile <file name>` can be retrieved by the command `read <file name>`. Try reading the file you just saved the weights in. Each file actually contains information about