UCRL-53919
Distribution Category UC-705

UCRL--53919

DE89 012297

A Structured Command History for UNIX Using a Parallel Distributed Processing Model

Jeremy Y. Uejio

(Master of Science Thesis)

Manuscript date: March 1989

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

Available from: National Technical Information Service • U.S. Department of Commerce
5285 Port Royal Road • Springfield, VA 22161 • A03 • (Microfiche A01)

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

A Structured Command History for UNIX
Using a
Parallel Distributed Processing Model

By

JEREMY Y. UEJIO

B.S. (University of Hawaii, Honolulu) 1984

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

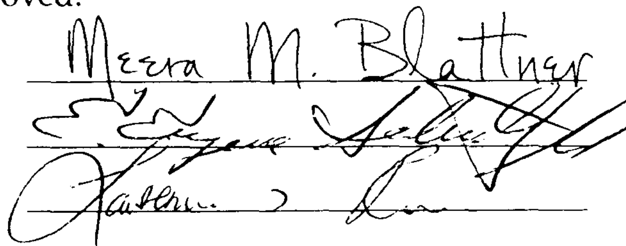
GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:



Committee in Charge

1989

Acknowledgements

I would like to express my gratitude to my thesis advisor, Professor Meera Blattner, and to Dr. Gene Schultz for their advice and encouragement. I would like to thank Professor Lawrence Kou for serving on my thesis committee.

I would also like to thank all my friends especially, Mark, Pat, and Tom for proofreading my thesis, and Ann, Lisa, and Rosie just for listening.

Jeremy Y. Uejio
March 1989
Computer Science

A Structured Command History for UNIX
Using a
Parallel Distributed Processing Model

Abstract

This thesis investigates the use of a structured history to assist users in recalling previously entered complex UNIX commands. A structured history is a database of commands that were previously entered. Two models are presented: the first model uses a conventional database and the second model uses a parallel distributed processing system. The conventional database system can recall commands by pattern matching based on command name, pattern matching on command options, frequency of use, and relative time. A simple prototype, created using the Emacs environment, was useful in recalling previously entered UNIX commands. The parallel distributed processing system stores UNIX commands by decomposing them into two character sequences called bigrams. A prototype implementation used a two layer network, in which each unit represents a bigram. The implementation showed features such as spelling correction and associated command recall. The ability to recall a list of commands satisfying a particular criteria is among the advantages of a structured history.

Table of Contents

Chapter 1: Introduction 1
1.1. Overview 1
1.2. The UNIX user interface 2
1.3. Problems with UNIX 5
1.4. Related work to improve the history mechanism 8
Chapter 2: A Conventional Structured History 13
2.1. Definition of a structured history 13
2.2. Design of the CS history 14
2.3. Implementation of the CS history 17
2.4. Summary 23
Chapter 3: The Parallel Distributed Processing Model 25
3.1. A brief history of PDP models 25
3.2. A basic description of PDP systems 26
3.3. Associative learning types 31
3.4. The operation of associative networks 32
3.5. Some properties of PDP systems 34
3.6. Summary 35
Chapter 4: The PDP History 37
4.1. Design of the PDP history 37
4.2. Implementation 41
4.3. Properties of the PDP history 47
4.4. Summary 47
Chapter 5: A Sample Session Comparing the Models 49
5.1. C shell history 49

5.2. CS history 51
5.3. PDP history 51
Chapter 6: Discussion 55
6.1. Advantages of a structured history 56
6.2. Problems and shortcomings 57
6.3. Future work and other possible designs 60
References 65
Appendix A: Emacs Code 67
A.1. CS history 67
A.2. The PDP history 76
Appendix B: SunNet Files 85
B.1. Network file 85
B.2. Learning file 85

Chapter 1

Introduction

1.1. Overview

This thesis investigates the use of a structured history to assist users in recalling previously entered complex UNIX commands. The UNIX command language is often difficult to use due its inconsistent syntax, its non-mnemonic command names, and its tendency for lengthy commands (Norman, 1981). It is often difficult to recall and re-execute previously entered commands. To alleviate this problem, an enhancement of the existing history mechanism, a structured history, was investigated.

A structured history is a database of commands that were previously executed. Two models are presented: the first model uses a conventional database and the second model uses a parallel distributed processing system. The conventional database system saves only complex commands and can recall them by pattern matching on command name, pattern matching on command options, frequency of use, and relative time. A simple prototype, created in the Emacs environment, was found to be very useful to recall previously entered commands.

The parallel distributed processing system (the PDP history) stores complex UNIX commands by decomposing them into two letter sequences called bigrams. A two layer network, in which each unit represents a bigram, was used. A prototype implementation was created and had features such as spelling correction and associated command recall.

A structured history can save a greater number of commands than the existing UNIX history and allows the user to recall a list of commands satisfying a particular criteria. Several problems still exist with the prototype implementation of the PDP history; however, an actual extension of the C shell history could be practical.

This thesis is organized into six chapters. The first chapter describes the UNIX user interface in greater detail and some of the problems with the command language. The second chapter defines the structured history and describes an implementation (the CS history) using a conventional database under the Emacs environment.

The third chapter is an introduction to parallel distributed processing systems (PDP). The fourth chapter describes the PDP history and a prototype implementation. The fifth chapter compares the existing history, the CS history, and the PDP history. Finally, the sixth chapter summarizes the advantages and disadvantages of a structured history and the implementation, and discusses other possible designs.

1.2. The UNIX user interface

The UNIX operating system, developed by Bell Laboratories nearly 20 years ago (Franklin, 1987), has become extremely popular in the academic and business world. The modular design and tool-box philosophy make UNIX an ideal programming environment for the expert user. However, there are a number of features that make UNIX difficult to use for both novice and expert users.

The bulk of user input consists of a series of command lines. Each line executes a function such as editing a file, compiling a program, or reading electronic mail. For example, a simple session could be the following:

`mail`	- execute the electronic mail program
`ls -l`	- display a listing of files in long format
`rm dead.letter`	- remove the file called "dead.letter"
`logout`	- exit UNIX.

A command line can be divided into two parts: a command name, which is the first word and is the name of the program executing the function; and the command options, which are the rest of the command. The options consist of arguments for the command name, filenames, and other commands (i.e. a user can execute several commands on a single command line with the use of pipes).

The user interface is implemented by a special program called the shell. The shell interprets the users input, controls input and output of user processes, and provides various programming features. There are three common UNIX shells—the standard AT&T Bourne shell, the Berkeley C shell, and an extension of the Bourne shell, the Korn shell. These and other shells provide various functions to assist the user in entering commands. Among such tools are (Sebes, 1987):

- *a command history,* which saves previously entered commands and allows the user to display and re-execute them,
- *command aliases,* which allows the user to define a one word abbreviation for a command line,
- *on-line help,* which contains information about commands identical to the printed manuals,

- *command name and file name completion,* which displays a list of command names or file names that begin with the characters of the user's input,
- *command line editing,* which allows the user to edit a command line before executing it, and
- *spelling correction,* which can correct the spelling of filenames, commands, and user names.

Three of the more widely used tools, the command history, command aliases, and the on-line help, will be discussed in greater detail.

1.2.1. The command history

The C shell saves all the commands chronologically in the history list as they are entered by the user. The UNIX Programmer's Manual (1984) describes the command history feature of the C shell as follows:

> "History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence."

The user can display the history list and choose commands to combine or re-execute. Examples of some history commands are:

`!!`	- execute the last command
`!10`	- execute the 10th command in the history
`!9:0-1`	- execute the command name and the first word of the ninth command (Sebes, 1987).

Only a selected number of the most recent commands in the history list can be saved between login sessions. Typically this number is small, less than 100.

The UNIX history was probably based on the assumption that users would only want to recall recently executed commands and would not need to save old commands.

1.2.2. Command aliases

An entire command line can be abbreviated to one word using the alias feature of the C shell or Korn shell. The aliases created by the user can be saved and automatically set up for future login sessions. The alias command is easy to use for a single session, but requires some planning and knowledge of UNIX to re-establish for future sessions.

1.2.3. On-line help

The on-line help is called "man pages" and contains the same information as the printed manual that is included with UNIX. The man pages are useful for determining the command name and arguments to be used to solve a particular problem, but are difficult to use if the user does not know the requirements for the command. The man pages are also static, that is, they contain the same information and organization regardless of a user's previous usage of commands.

1.3. Problems with UNIX

1.3.1. Some observations

Norman (1981) observes that the command language for UNIX has an inconsistent syntax and many non-mnemonic command names. For example, some commands have arguments that begin with a dash, such as "ls -l," and others do not. Some commands have arguments that are a single letter and others have arguments that are complete words (e.g. "find -name

foo -print"). Some command names are formed from the first two consonants of their respective function names (copy becomes "cp" and remove becomes "rm") while other command names are formed by an apparently arbitrary scheme ("cat" for concatenate or "ed" for editor).

Some command names have no relation to the function that they execute. For example "grep" is a program to search a file for a pattern, "awk" is a pattern matching and scanning language (named after the authors—Aho, Weinberger, and Kernighan), and "troff" is a program which formats files for printing.

In addition, UNIX commands tend to be lengthy because they contain many options and sometimes several commands. (Unfortunately this tendency is inherent in the tool-box philosophy of UNIX.) Some examples from sample sessions are shown below:

```
cat p313.data | graph -gl -c o -l 'Problem 3.15' | psplot | lpr -h
find /u0 -name latex -print > latex.psn &
awk '{print substr ($0,8)}' oldhistory | sort | awk -f freq.awk
cc -O -Dstrrchr=rindex -s -o hier hier.c sftw.o
```

Note the inconsistent arguments such as "-c," "-print," and "-Dstrrchr=rindex." The syntax for each command name can be quite demanding and can make it difficult for the user to compose and remember complex and infrequently used UNIX commands.

1.3.2. *A typical command*

The following is a detailed example of a typical, lengthy, difficult command that a user might execute. Suppose the user wants to print part of a file to a PostScript Laser printer. A possible UNIX command to do this is:

```
ptroff -me -PT4387 -o1-2 introduction.txt
```

where `introduction.txt` is the file to be printed. "Ptroff" actually runs "troff" to format the file, then it runs "pscat" to output the file to a PostScript printer. The options are those for "troff." The "–me" means to use the "me" macros, the "–PT4387" means to send output to a printer called "T4387," and "–o1–2" means to print the first two pages. The requirements and syntax for this command are quite demanding. The user would require the man pages and several incorrect attempts before entering the correct command name and options.

A few weeks later, the user wishes to format another file but has completely forgotten the above command. Unfortunately, the existing UNIX tools for assisting the user are not helpful. The command history could not be used because:

- the user does not remember exactly what command was used or when the command was last executed,
- the command is no longer in the history list (due to the finite size of the list), and
- even if the command were in the list, it would be difficult to find, due to both the length of the list and the simple methods of recalling events.

Other tools such as command aliases would not be useful since the user did not explicitly create an alias for the command after it was first executed. The user would have to revert back to the man pages and once again spend time determining the requirements and syntax for this complex command.

1.4. Related work to improve the history mechanism

1.4.1. Greenberg and Witten's study

Greenberg and Witten (1988) conducted a study to determine principles for the design of history mechanisms. They recorded the commands entered by 168 users with various experience levels using the C shell over a period of four months. They found that there is a 43% chance that the next command occurred within the previous seven commands, a 31% chance that it occurred further back in the history, and a 26% chance that the next command line has not appeared before. (The investigators also found that just over half of the subjects used the C shell history to recall commands, but for only 4% percent of their total commands. Users found the history syntax difficult and not worth the trouble.)

Perhaps if the number of commands saved in the history list were increased, then the chance of the next command being in the history list would greatly increase. Greenberg and Witten found that even extending the history to 25 items would only increase the chance to a little under 60%. There is still a 15% chance that the next command was previously entered, but no longer in the history of recent commands.

There is also the likelihood that many new commands are variations of old commands. For example, in the previous "ptroff" command, the user might enter:

```
ptroff -ms -PT4387 -ol-2 introduction.txt   –use the "ms" macros
ptroff -me -Plwl -ol-2 introduction.txt     –use a different printer
ptroff -me -PT4387 introduction.txt         –print the entire file
```

All these commands would be counted as new commands in the Greenberg and Witten study, however, to the user, they are essentially the same command. In this case, a history mechanism would not be used to perfectly recall a command. Rather, it would be used to remind the user of the basic structure of the "ptroff" command.

In addition to determining the likelihood of command reuse, Greenberg and Witten evaluated three methods of altering the history to increase the chance of a command being in the history list. The three methods are directory sensitivity, pruning duplicates, and partial matches. Directory sensitivity means that separate histories would be used for each directory. Pruning duplicates means that duplicate commands are removed from the history list. Finally, partial matches means that if the next command contains a previous command, then the next command is considered to be in the history of previous commands. They found that a combination of methods has the highest probability for predicting the next command. From their study, they conclude that history mechanisms are useful, and with careful design, can recall the majority of commands entered by the user.

1.4.2. GNU Emacs

GNU Emacs (Stallman, 1986), an extensible editor, includes a UNIX shell mode in which the user can scroll back thru the buffer and search for previously entered commands. The user can also easily edit commands before sending them to the shell. There is a great potential for designing a history mechanism in the shell mode of GNU Emacs.

In addition, the meta-command language of Emacs has a history feature allowing users to scroll through previously entered meta-commands to edit then re-execute them. (This feature, however, is not widely used because Emacs commands are easy to remember and do not contain lengthy complex arguments.)

1.4.3. Other studies

There are several studies on general command reuse not related to UNIX. For example, Yang (1988) describes a conceptual model for user recovery and command reuse, Vitter (1984) has applied his US&R package for undo, skip, and redo to a graphics layout system, and Common Lisp defines a simple mechanism to recall forms (Steele, 1984). However, all of these studies discuss only simple methods of recalling commands, such as chronological lists, and do not consider a more general approach.

Finally, Ishikawa (1987) investigated the use of a parallel distributed processing model to automatically correct spelling errors and find frequently used sequences of UNIX commands. The purpose of this study was to increase the flexibility of command languages when faced with incorrect user input.

This paper investigates the use of an extension of the existing history mechanism, called the structured history, to assist users in recalling previously entered UNIX commands. Two models will be presented: a conventional database model and a parallel distributed processing (PDP) model. The next section defines the structured history and describes an implementation using a conventional database under the GNU Emacs environment.

Chapter 2

A Conventional Structured History

2.1. Definition of a structured history

The purpose of a structured history, like the existing UNIX history, is to provide quick easy access to UNIX commands that were entered by the user. However, unlike the existing history, a structured history saves commands in an organized manner, and not as a simple list, therefore providing a variety of strategies for recalling commands. It also saves most of the commands entered by the user, not just the recent commands.

A structured history consists of three parts: a database of commands, a means of saving commands in the database, and a means of searching the database for commands. The existing UNIX history is a very simple database. Each command line is a data item, all commands are saved, and there are only two ways to access commands: by relative time (position in the history list) or by pattern matching with the first few characters.

A structured history is a more complicated database. Each command is broken into parts and attributes. The two parts of a command are the command name and the options. The attributes of the command can be frequency of use, files accessed, time executed, keyword descriptions, etc. This allows the user to be more specific about the command he wishes to recall. For example, the user may ask for "the most frequent command which accessed the file temp.dat."

Two models of a structured history will be presented. The first model, discussed in this chapter, is a conventional database model in which the parts and attributes of a command are saved and the user must explicitly ask for them. This model is called the conventional structured history or CS history. The second model, discussed in chapters three and four, uses a parallel distributed processing (PDP) system. In this model, a command is broken into a set of two-letter combinations. The attributes, or the structure of the history, is implicitly contained by the nature of PDP systems. This model is called the PDP history.

2.2. Design of the CS history

The design of the CS history consists of a database for commands, a saving method, and a recalling method.

2.2.1. Database design

The database for previously entered UNIX commands has a simple structure:

- Each command is composed of two parts: the command name and the command options (arguments, filenames, etc.).
- Each command contains two attributes: the number of times the command was previously executed (command frequency) and the relative time that the command was executed (similar to the existing C shell history).

2.2.2. Saving commands

Two principles are followed when saving a command.

1) Only complex commands are saved.

There are many different measures of command complexity. The number of

words, the number of options, the length of the command line, or a combination of the above could be used. Some commands, such as "ls" or "cd," are simple enough to remember and re-enter without the help of a history mechanism, thus do not need to be saved. Other commands, such as the "ptroff" example in chapter one, are difficult to remember and should be saved in the history. As a first approximation, the length of the command line was used to measure complexity—only lines longer than five characters were saved in the CS history.

2) Only one copy of a command is saved.

The existing UNIX history saves a command, even if it was executed many times before. Duplicate commands cause the history to grow in length but not in information content. For example, suppose a user enters the "ptroff" command mentioned earlier. Then the user debugs a C program by repeatedly editing the program, running the compiler, then executing the program (i.e. executing the three commands "vi test.c," "cc test.c," and "a.out"). The history list will now be filled with many copies of these three commands. Since the list is of finite (and often of short) length, the "ptroff" command will soon be removed from the list and the user will no longer be able to recall that command.

The CS history, however, will save a command only once, regardless of the number of times it has been executed. (The frequency of use will be updated each time). The command will be saved chronologically at the latest time it was executed. In the above example, the history list will contain the four commands, "vi test.c," "cc test.c," "a.out," and the "ptroff" command.

The above two principles were applied to a sampling of commands taken over a period of several months from the history logs of three experienced UNIX users. These history logs were reduced by at least 80%! The users together executed over 20,000 command lines, but only a little over 3000 were unique and over five characters long.

2.2.3. Recalling commands

There are two ways a user can recall commands: as a single command or as a list of commands. The single command method is used to recall the last command containing a specified string or, if no string was specified, the last command executed. The list of commands method is used to recall a set of commands which meet a given criteria or are sorted in a given manner. The different methods are listed below.

- *Pattern matching using the command name.* The user enters a sequence of letters which might be a command name or part of a command name and the system displays all previously entered commands in which the name contains those letters.
- *Pattern matching using the command options.* The user enters a sequence of letters which might be contained in the command options. Usually, these letters are part of a filename, but they can also be part of the arguments to the command. The system then displays all previously entered commands in which the options contain the specified letters.
- *Relative time of last execution.* The system displays a list of commands sorted chronologically.
- *Frequency of execution.* The system displays a list of commands sorted by the number of times the command was used.

- *Combination of the above.* The user can combine the different methods. For example, the user can display all commands that have options containing "temp.dat," sorted by frequency.

The single command method is useful when the user knows exactly what command to execute, but does not want to re-type the command. (The existing UNIX history provides this method and was probably designed for this purpose.) The list of commands method is useful when the user does not know exactly what command he wants, but can recognize it among similar commands.

2.3. Implementation of the CS history

A simple prototype of a conventional structured history was written using Lisp in the GNU Emacs environment. The source code is listed in Appendix A.1. This prototype used conventional data structures and algorithms and allows a user to recall commands by frequency, time, and pattern matching. It was found to be very useful by the author and several of his colleagues.

2.3.1. Emacs and Lisp

GNU Emacs* (Stallman, 1986) provides an easy to use powerful editor and a Lisp programming environment. Emacs includes a shell mode that calls the C shell, thus making it convenient and relatively easy to add a history to the C shell without actually modifying the C shell. Emacs provides functions to call other processes and redirect both input and output for that process. (This

---

* GNU Emacs is available on most UNIX systems and, although not in the public domain, is free.

feature was used in the PDP implementation to run SunNet from Emacs.) In addition, the Emacs Lisp environment allows incremental coding of Lisp programs and contains an interactive debugger.

2.3.2. *Schematic of the implementation*

The basic design of the implementation is shown in Figure 2.1. The flow of processing is as follows. First, the user executes the "shell" function of Emacs. This function reads the user's history of commands from a predefined file and executes a C shell subprocess for UNIX. All input and output to and from the shell is done thru an Emacs buffer called "*shell*."

Now, the user can: 1) execute UNIX commands by entering them and pressing the return key (<CR>) or 2) execute history commands with the prefix <ctrl>-C and a letter (the "control" key and the "C" key are held down simultaneously, then another letter is pressed)** . In the first case, two things happen: the command is added to the history list, and then the command is sent to the shell subprocess and executed. If the command is "exit," then the shell is terminated and the history file is saved.

In the second case, the CS history either displays a single command which the user can edit then execute, or the history calculates and displays a list of commands in another buffer called "*histo*." The user can scroll thru this list and select a command to be executed or apply a history command to display another list of commands.

**The <ctrl>-C prefix was used because the existing shell-mode of Emacs already uses several <ctrl>-C prefix commands.

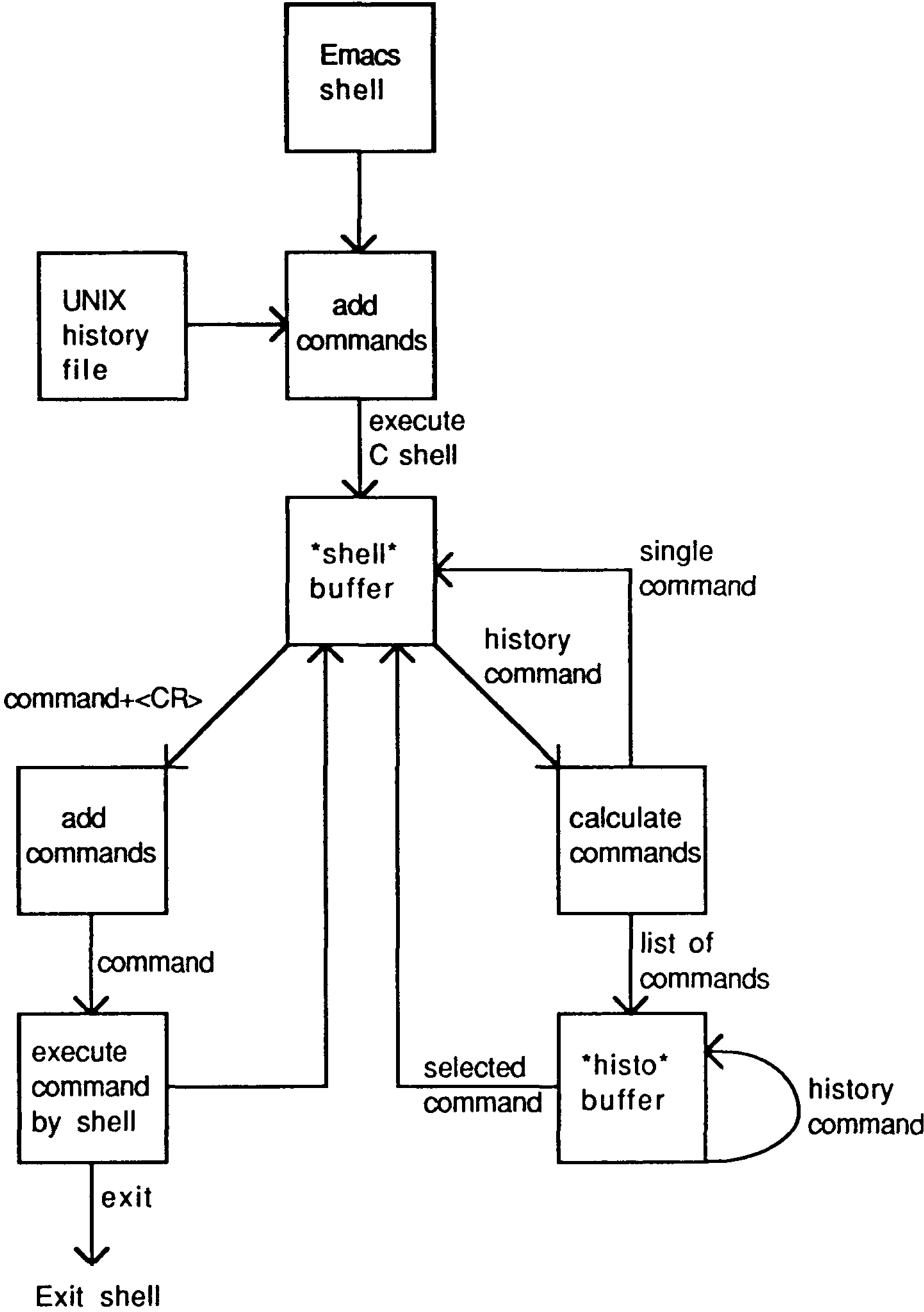


Figure 2.1: Schematic of the conventional structured history

2.3.3. *Primary data structures*

A Lisp data structure called a "cons" is used to hold the command name, the options, and the frequency of use for this command. The first part of the cons (the car of the cons) contains the entire command line as a string. The second part (the cdr) contains a new unique symbol name. When a command line is added, the line is parsed into the command name and the options. Both of these values and the frequency are then placed on the property list of the new symbol. The entire command line is stored to allow quick access to the whole command line for displaying and searching for duplicate commands. The command name, options, and frequency are placed on property lists also to facilitate searching.

Two lists are used to store the previously entered UNIX commands: `histo-list` and `histo-current-list`. `histo-list` contains all the previously entered commands in chronological order. This list is only used to store the commands and is not viewable by the user. `histo-current-list` contains a list of commands in a specific order. This list is a subset of `histo-list` and is displayed to the user.

The two lists allow the user to apply various recall methods to `histo-current-list` without modifying `histo-list`. For example, suppose the user displays all commands that have options containing the file "temp.dat" ordered by frequency of use. First, all commands in `histo-list` that contain the string "temp.dat" are copied to `histo-current-list`. Then `histo-current-list` is sorted by the frequency property. Finally, `histo-current-list` is displayed for the user. `histo-list` remains unchanged.

2.3.4. Saving commands

Only a single copy of commands longer than five characters are saved. The command is saved in the following manner. First, a cons for the command is created. Then, the new cons is added to the end of the history list. If the command line already exists in the history list, then the frequency of the new item is incremented by the frequency of the old item and the old item is removed from the history list. This results in the command being placed at the latest time it was executed. If the command is "exit," then the history file is saved. (The "exit" command sent to the C shell terminates this subprocess.)

2.3.5. Recalling commands

If the user enters the prefix command, <ctrl>-C, the user executes a history feature to recall a command. Some of the features return single commands to the *shell* buffer and others display a list of commands in another buffer called "*histo*." The user can scroll thru the *histo* buffer and select a command to be executed or the user can execute another history command. If the user selects a command, then that command is placed in the *shell* buffer and the user may edit the command before executing it.

The following history recalling features have been implemented (Note: the current string refers to any characters that the user has entered in the *shell* buffer before pressing the prefix command):

- "<ctrl>-C a"—List all commands in the history chronologically.
- "<ctrl>-C c"—List all commands in which the name contains the current string.
- "<ctrl>-C s"—List all commands in which the options contain a given string.

- "<ctrl>-C f"—List all commands by frequency of use.
- "<ctrl>-C l"—Insert the last command (recall a single command) containing the current string or the last command executed if no current string exists.

In the *histo* buffer, the user may combine some of these selection strategies by re-applying the history commands. For example, if the user wanted all the commands containing the string "temp.dat" arranged by frequency, the user simply enters "<ctrl>-C s temp.dat" from the *shell* buffer and then enters "<ctrl>-C f" from the *histo* buffer.

Also in the *histo* buffer, the user can delete commands from the history. Often when a composing a new command, a user will enter several incorrect commands before entering the correct command. The delete function allows the user to remove these unnecessary commands.

2.3.6. *Algorithms*

The features listed above were implemented using the standard list functions of Lisp and special search functions of Emacs. For example, both the pattern matching by name and the pattern matching by options use the function `histo-match-list` shown below:

```
    (defun histo-match-list (property pattern)
       "Returns a list of commands whose PROPERTY matches a given
PATTERN.  If current-buffer is *histo* then use histo-current-list, else
use histo-list."
       (let* ((correct-list (if (string= (buffer-name) "*histo*")
                                 (copy-alist histo-current-list)
                               (copy-alist histo-list)))
              (match-list (if (string= pattern "")   ; if looking for all
                               correct-list           ; items then return
                             (mapcar '(lambda (item) ; all items else
                                                      ; search for pattern
                                        (if (string-match pattern
```

```
                                    (get (cdr item) property))
                                  (item))
                              correct-list))))
      (delq nil match-list)))
```

The entire history list is searched by applying "string-match" (a special Emacs regular expression search function) to each item and concatenating the resulting list of commands using the "mapcar" function (a standard Lisp function).

Other features such as the scrollable lists in the *histo* buffer, editing of commands in the *shell* buffer, and assigning the <ctrl>-C keys, were simply calls to Emacs Lisp functions. Since Emacs is a programmable editor, any editing function can be used in Lisp programs.

2.4. Summary

A structured history is basically a database of previously entered UNIX commands, a saving method, and a recalling method. The command database used in the CS history (conventional structured history) consists of two parts—a command name and the command options—and two attributes—a relative time and the frequency of use. The saving method is based on two principles: only complex commands are saved and only one copy of a command is saved. There are two ways in which commands are recalled: as a single command or as a list of commands. Lists of commands can be recalled by pattern matching with the name or options, by relative time, by frequency of use, or a combination of the above.

A prototype implementation was constructed using GNU Emacs. The user enters the C shell from Emacs and can easily recall previously entered commands.

In the next two sections, a structured history using a parallel distributed processing model will be discussed. First a description of PDP models will be given, then a specific implementation will be described.

Chapter 3

The Parallel Distributed Processing Model

As stated previously, a structured history provides easy access to UNIX commands that have been previously entered by the user and allows a variety of strategies for recalling these commands. In the conventional system, the user explicitly specifies the types of strategies used to recall a command. In a parallel distributed (PDP) model, however, the recall strategies are not explicitly given. The user simply specifies a string of characters that may be in the command that the user wishes to recall. The PDP history "searches" thru the list of commands and presents the user with a list of likely candidates. The list is implicitly ordered by various recall strategies.

What is a parallel distributed processing model and how does it "search" a list of commands? This chapter defines the PDP model, then discusses the specific type of model used as a command history. The next chapter describes the design of a PDP history and a prototype implementation.

3.1. A brief history of PDP models

Although PDP models (also known as neural networks or connectionist systems) have just recently become popular, they were proposed many years ago. In 1943, McCulloch and Pitts published a paper describing an abstract model of neurons and showed that a network of neurons could simulate certain logical expressions. In 1949, Donald Hebb proposed a mechanism for learning and in 1959, Rosenblatt developed a class of simple neuron-like units called perceptrons.

However, Minsky (1969) showed many limitations of perceptrons. Probably for this reason and computer hardware limitations, interest in PDP models diminished. It was not until the early 80's when Hopfield (1982) showed that a system of McCulloch and Pitts-type neurons demonstrated "collective computational properties" and could be used as memories, that PDP models began to be studied again.

In 1986, Rumelhart and McClelland published *Parallel Distributed Processing* —the "bible" for PDP models. (They first used the term "parallel distributed processing.") Their two volume set describes many different types of PDP models and various psychological and biological applications. Their book provided much of the background for this thesis.

3.2. A basic description of PDP systems

A PDP system consists of four parts: a group of simple processing units, a method of connecting the units, a method of transferring information among units (a propagation rule), and a method of learning.

3.2.1. *The group of simple processing units.*

Each of the simple units has an activation rule, an activation level, and an output function. (See Figure 3.1) The activation level is a function of time and represents the state of the unit. Often it is a binary value—either on or off. The activation rule calculates the activation level based on the net input to the unit and the current activation level. The output is a function of the activation level and is usually the identity function (i.e. output = activation) or a threshold function (output is 1 for activations greater than a given value and 0 otherwise).

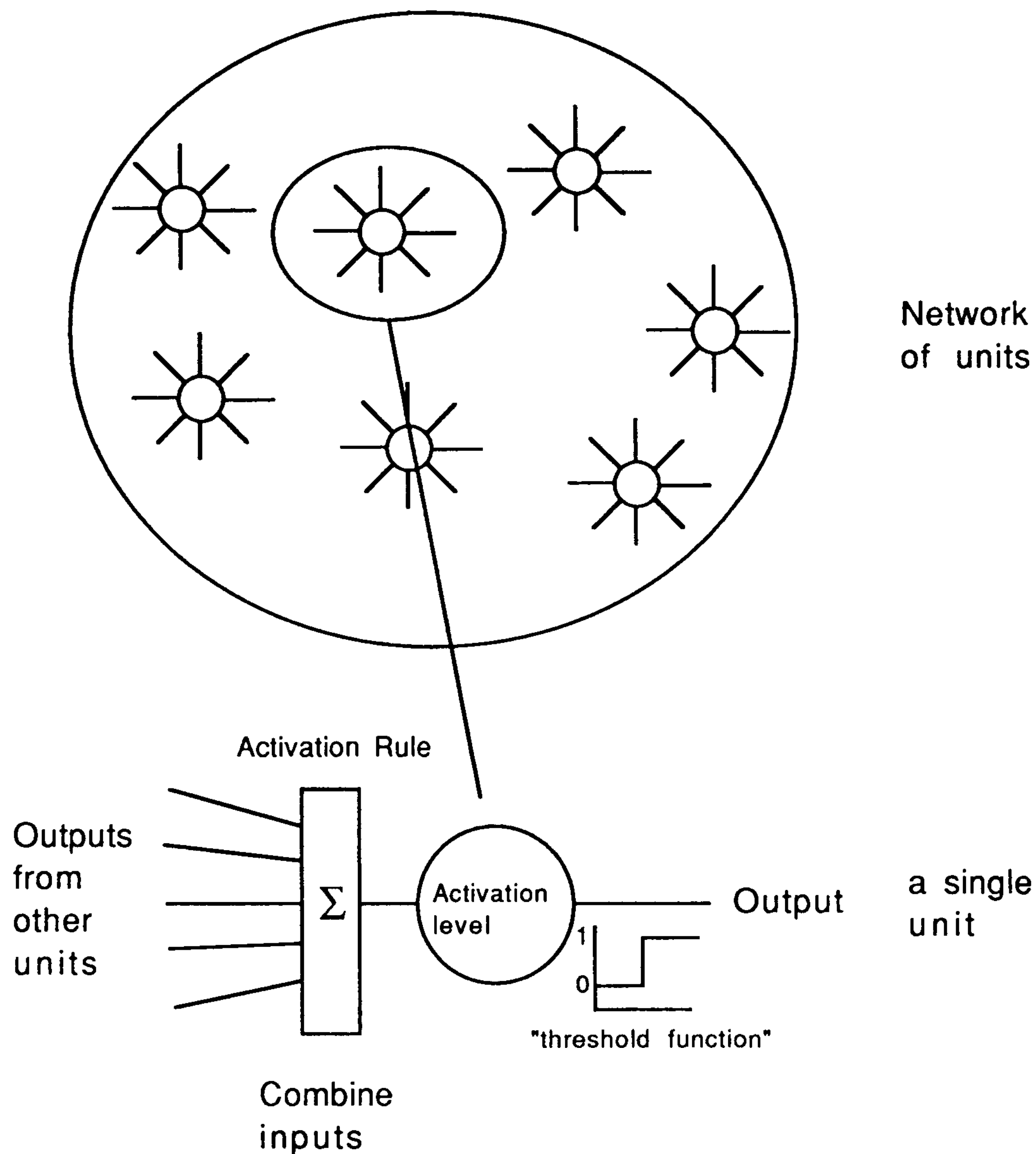


Figure 3.1: A simple processing unit

Some of the units receive their input from outside the network and others receive them from within the network. Some units send their output to the external world and some send their output to other units. The units that

receive input externally are called input units; the units that send output externally are called output units; all the other units are called hidden units. The activations that the input units receive are collectively known as the input pattern. The activation levels of the output units are collectively known as the output pattern. (See Figure 3.2)

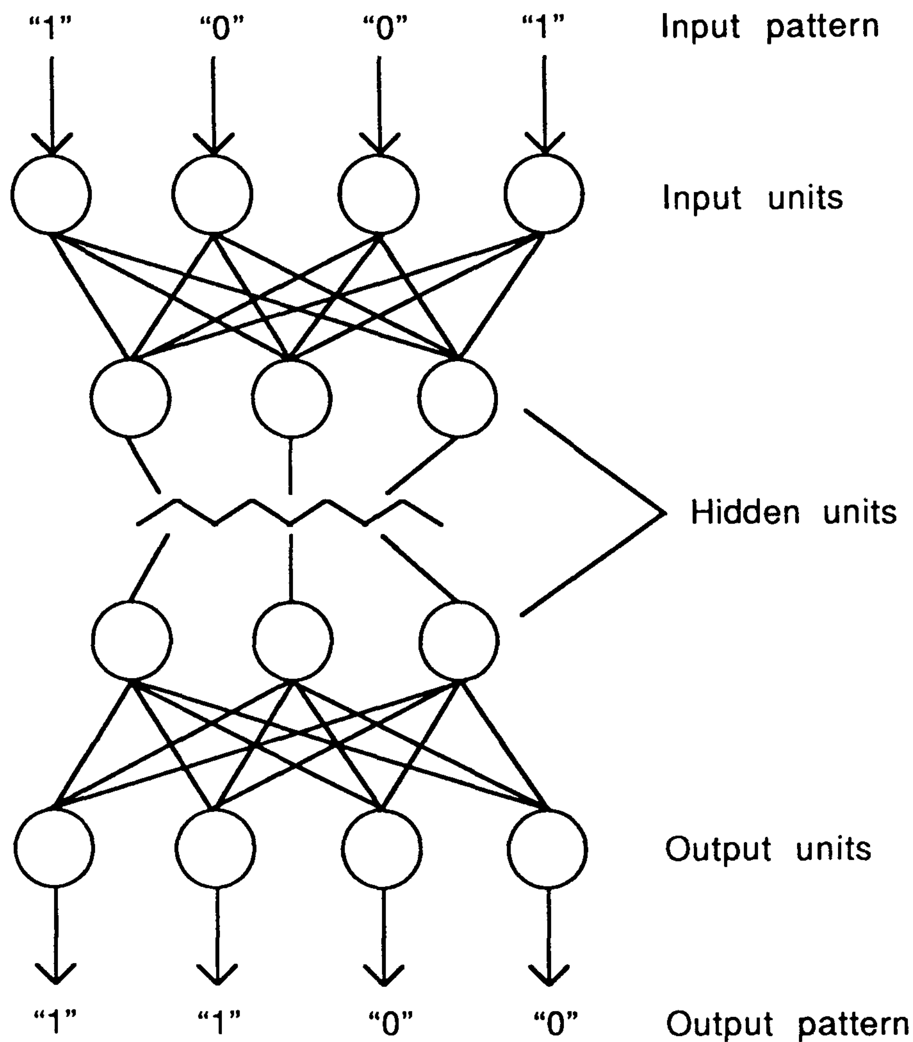
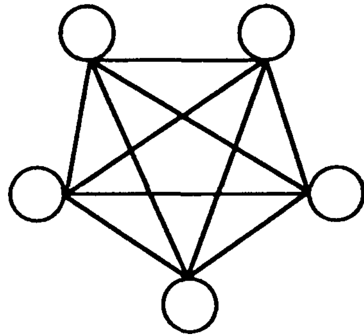
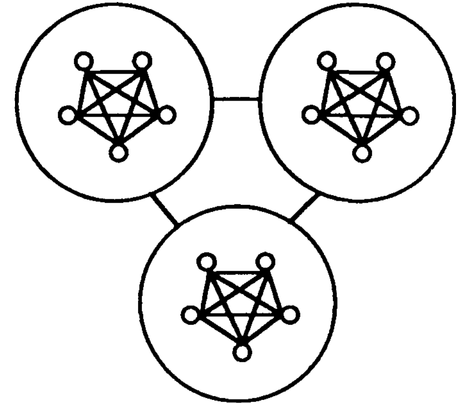


Figure 3.2: Types of units—input or output





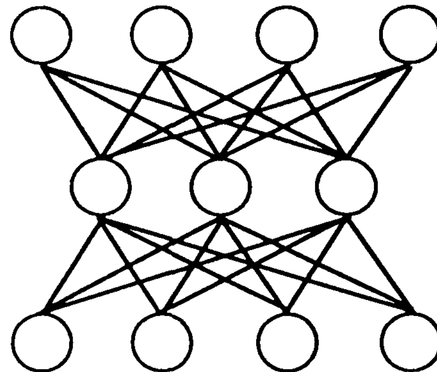


Figure 3.3: Network connections

3.2.2. *The method of connecting the units*

The units in a PDP system can be inter-connected in many different ways. Figure 3.3 shows some possibilities: a) a general network in which each unit is connected to every other unit, b) pools of units in which units within a pool are completely inter-connected and each pool is connected to every other

pool, and c) layers of units in which units are only connected to units in the preceding or following layer.

3.2.3. *The propagation rule*

The propagation rule states how the output of various units is combined to form the inputs to other units. This process is dependent on the strength of connections between the units and the type of connection. Often, there are two types of connections: inhibitory and excitatory. The net input is calculated separately for the two types. The net excitatory input is the weighted sum of the outputs of the excitatory units and the net inhibitory input is the weighted sum of the outputs of the inhibitory units. (The activation rule will then combines these two types of inputs.)

3.2.4. *The learning rule*

Finally, the system may have a learning rule. Rumelhart and McClelland (1986) define learning as "a matter of finding the right connection strengths so that the right patterns of activation will be produced under the right circumstances." Learning is improving the performance of the network by changing the strengths of connection between units.

There are two ways in which a PDP system can learn: with an external set of patterns (supervised learning) or by itself (unsupervised learning). The more popular form of learning is supervised learning. Typically, a set of example patterns is used to train the network. The strengths of connection between the units is changed as the network cycles thru the patterns repeatedly until some measure of error (often a least squares measure) between the example patterns and the patterns calculated by the network is minimized.

3.2.5. Types of systems

Rumelhart and McClelland (1986) classify PDP systems into two learning types:

- Associative learning. The system learns to associate a set of input patterns to a set of output patterns, and
- Regularity discovery. The system tries to find features in a set of patterns. (These models are not used in this paper. See Rumelhart and McClelland for further information.)

3.3. Associative learning types

Associative learning can be divided into three subtypes: auto-associators, pattern associators, and pattern classifiers (Rumelhart & McClelland, 1986).

3.3.1. Auto associators

Auto associators can be thought of as memories. Patterns are stored by repeatedly presenting them to the network. A pattern is recalled by presenting part of it to the network. The network then tries to complete the pattern.

3.3.2. Pattern associators

Pattern associators are also thought of as memories, but, instead of learning single patterns, they learn to associate pairs of patterns. One pattern in the pair is called the input pattern and the other pattern is called the output pattern. The two patterns are usually different. The pairs of patterns are learned by repeatedly presenting them to the network. If a single input pattern is presented to the network, then the corresponding output pattern should be recalled.

3.3.3. *Pattern classifiers*

Pattern classifiers learn to categorize patterns. A set of training patterns and their corresponding categories is first presented to the pattern classifier. Then a new pattern, possibly slightly different from the original patterns, is presented to the network. The pattern classifier will output the appropriate category.

3.4. The operation of associative networks

Associative networks typically operate in two stages: a training or learning stage and a testing stage. In the training stage, the network learns to discriminate the patterns. In the testing stage, the network recognizes the patterns. In both stages, information is propagated through the network.

3.4.1. *Propagation*

Propagation is the activation or flow of information from the input units to the output units. First, an input pattern is presented to the network. The input units calculate their activation and output level based on the strengths of the external input pattern. Other units then calculate their activation and output based on the connection strengths and output levels of the input units. The activation and output levels of the system of units are continuously updated in this manner (i.e. information is propagated thru the network). Finally, the activation levels of some of the units, called the output units, are used to determine the results of the process.

3.4.2. *Learning and the delta rule*

Learning is improving the performance of the network by changing the connection strengths. In the learning stage, a set of training patterns is

repeatedly presented and propagated through the network. In each pass or cycle the connection strengths between units is modified.

Most forms of learning are Hebbian learning (Hebb, 1949). Simply stated: the strength of the connection between units must change as a function of the activation of the units. For example, suppose unit A is connected to unit B as in Figure 3.4. If a certain input pattern activates unit A and the corresponding output pattern includes unit B, then the connection strength between A and B will be increased. More precisely, the change in connection strength, ΔW, from A to B is:

$$\Delta W = N A_{out} B_{act}$$

where N is a learning parameter, A_{out} is the output of unit A, and B_{act} is the activation level of unit B (Bayle, 1988).

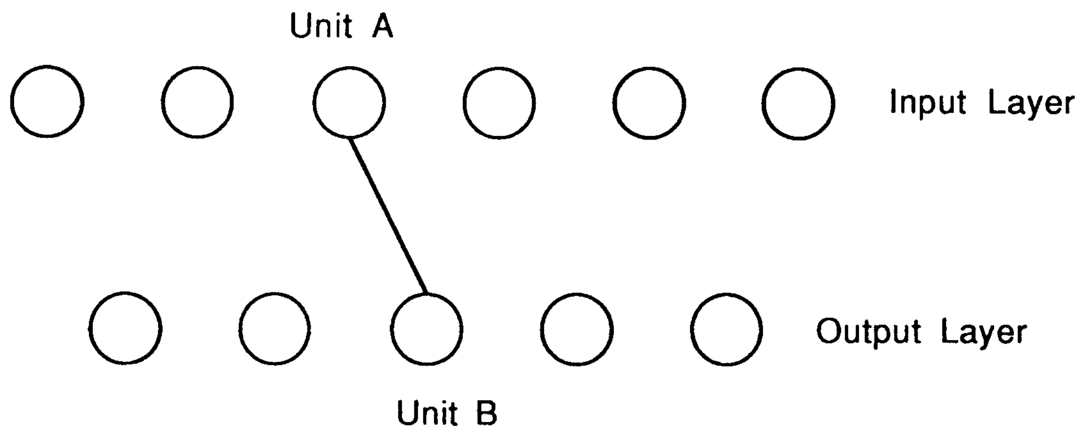


Figure 3.4: Example for learning rules

The most simple form of Hebbian learning is for the change in connection strength to be directly proportional to the activation levels of the two units. Unfortunately this type of learning can only learn completely unrelated sets of patterns thus a more powerful variation of Hebbian learning, called the delta rule, is often used.

3.4.2.1. the delta rule

The basic notion of the delta rule is to compare a desired output pattern with the actual output pattern and then modify the weights to reduce the difference. This is done for each pattern in the training set. The delta rule is:

$$\Delta W = N\ A_{out}\ (t - B_{act})$$

where N is a learning parameter, A_{out} is the output of unit A, B_{act} is the activation level of unit B, and t is the desired activation level for unit B (Bayle, 1988).

3.5. Some properties of PDP systems

PDP systems have the following properties:

- *Information is distributed in the connection strengths.* With conventional data structures, information is contained in records or specific slots. In a PDP model, however, information about items is distributed in many different connection strengths throughout the network.
- *Fault tolerant and error correcting.* Since the information is contained in many connection strengths, there is a redundant amount of infor-

mation. A perfect copy of the input pattern is not needed because a PDP system can overlook certain defects.

- *Pattern completion.* A PDP system can be given an incomplete input pattern and will be able to make a "best guess" and give an appropriate output pattern.
- *Ability to generalize.* A PDP system can learn a series of patterns and implicitly group them by features.
- *Show graceful degradation.* In a hardware implementation, where units are represented by individual CPUs, if a few CPUs fail, then the network can still function. The network will gracefully degrade in performance as more CPUs fail.
- *Find rules and learn by examples.* Rules are learned by training over example patterns and not by explicitly stating parameters. This means that the same network can be used for a variety of applications without the programmer having to describe the details of each application.

The error correcting nature and the ability to learn by examples are desirable features for a UNIX command history. Error correction allows the user to misspell a command or to enter an incomplete command and have the system recall the correct command. The ability to learn by examples allows the system to be automatically configured to each user.

3.6. Summary

A PDP model consists of four parts: a collection of processing units, a method of connecting the units, a propagation rule, and a learning rule. There are many varieties of PDP models. Different functions have been used to calculate the activation and output levels; the units have been arranged in various

ways; and many propagation and learning rules have been used. Associative networks operate by first learning a set of training patterns, then by being tested with a new pattern. The ability to learn by examples and the ability to output the appropriate pattern given an incomplete input pattern, are two features that are useful for a UNIX command history.

Chapter 4

The PDP History

4.1. Design of the PDP history

Recall that a structured history has three parts: a database for commands, a saving method, and a recalling method. In a PDP history, the database is a PDP network, the saving method is a form of learning, and the recalling method is a form of testing the network.

4.1.1. The PDP system for commands

In a PDP history, each command is not broken into parts and attributes as in the CS history, but rather into component letters. Each command is composed of two character sequences called bigrams (Kohonen, 1988). For example, the command "`ls -la`" is composed of the bigrams: "`ls`," "`s `," "` -`," "`-l`," and "`la`."

A two layer network, as shown in Figure 4.1, was used. Both layers contain an equal number of units. Each unit in the input layer represents a unique bigram and each unit in the output layer represents a unique bigram. The activation level of the units in the network represents the likelihood that the respective bigram is present in the command.

The PDP history decomposes each command in the history into a set of bigrams. A PDP system is then used to associate each of the bigrams, composing a command, to one another. In the recalling phase, the user enters a string of characters (a few bigrams) and the history converts it to a command or list of commands of which the user can select to re-execute. The

PDP system is used to "expand" the few bigrams that the user entered into a list of bigrams. Then a conventional system converts these bigrams into a command or a list of commands.

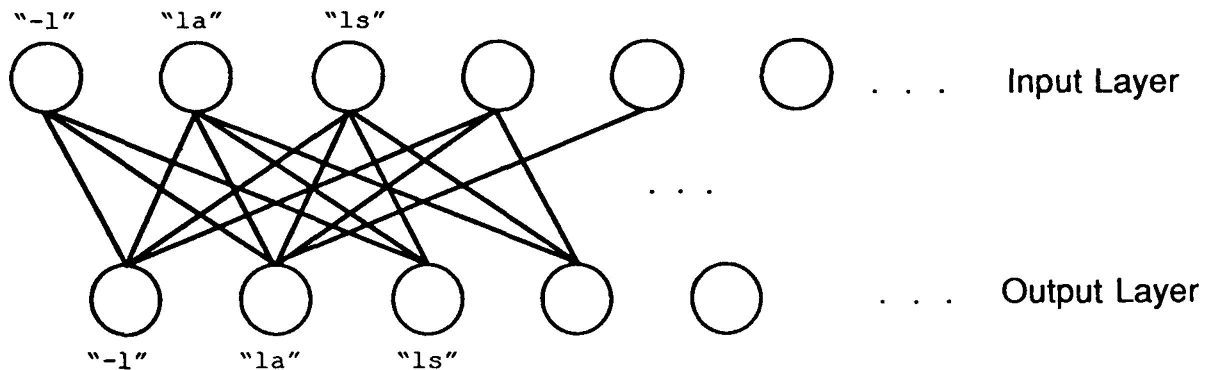


Figure 4.1: Two layer network

A brief schematic of the PDP history is shown in Figure 4.2. First a UNIX history file is learned by the network and a UNIX shell is executed. The user searches for a command to re-execute by entering a string of characters that might be in the command. The history tests this pattern and returns a list of possible commands. Finally, the user can select the appropriate command and edit or execute it.

4.1.2. *Saving commands (learning)*

Only one rule, command complexity, is used to determine which commands are to be saved and learned by the history. All commands over five characters in length, including duplicates, are saved. The saving or learning phase consists of a converter to encode the list of UNIX commands to bigrams and the PDP network to store the bigrams. Each bigram activates its respective unit. These units then modify the connection strengths between themselves

to learn the set of bigrams representing a particular command. This process is applied repeatedly to all commands in the history list until the difference between the presented bigrams and the output of the network is a minimum.

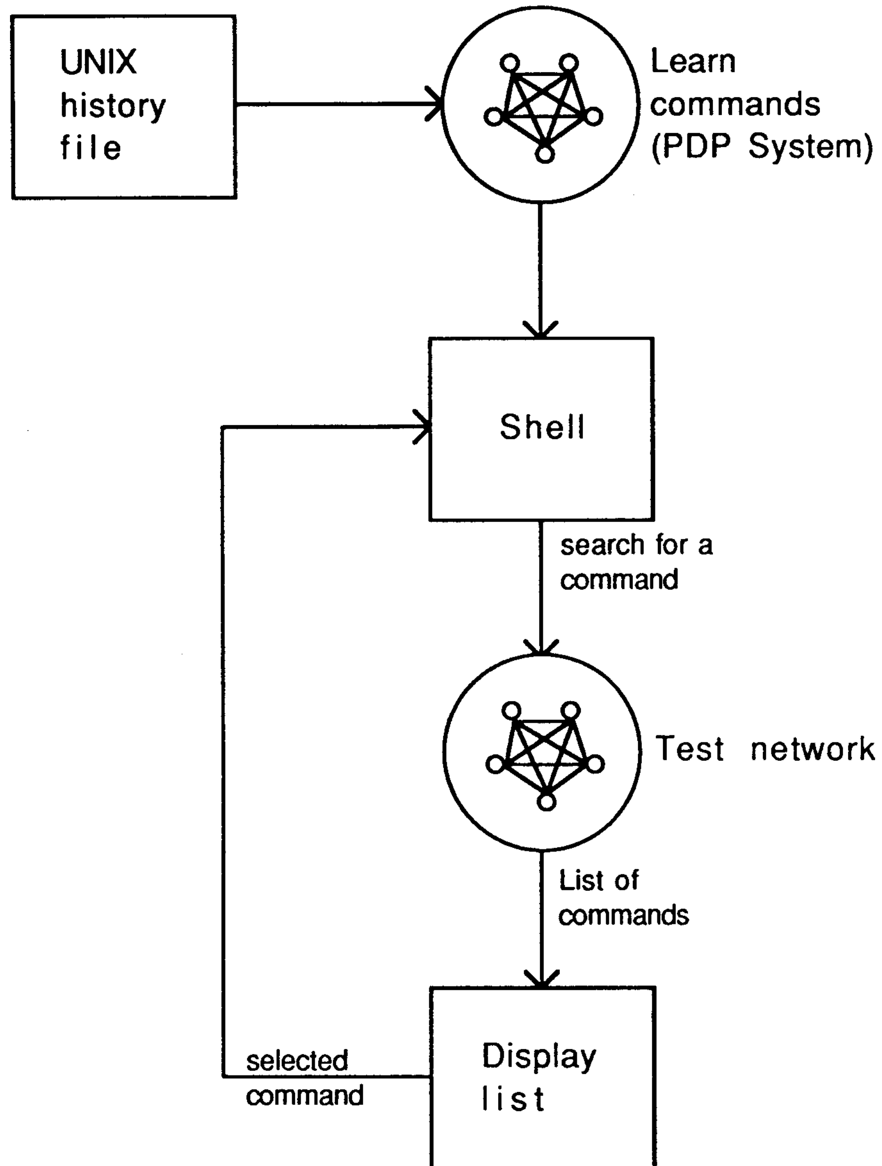


Figure 4.2: A brief schematic of the PDP history

The delta rule was used as the basis for learning. It is basically a variation of Hebbian learning—the connection strengths between the bigrams composing a command is increased each time the command is learned. Recall that the delta rule for unit A giving input to unit B is:

$$\Delta W = N\, A_{out} (t - B_{act})$$

where N is a learning parameter, A_{out} is the output of unit A, B_{act} is the activation level of unit B, and t is the desired activation level for unit B. This formula is applied to each of the connection strengths between the two layers of the network.

4.1.3. Recalling commands (testing)

The recalling phase consists of an interpreter to convert a user's input to a test pattern of bigrams for the network, the PDP network, another interpreter to convert the output of the network to a list of commands, and finally a routine to send the selected UNIX command to the shell. The user attempts to recall a command by entering a string of characters which might be contained in the command. The user's input is converted to a test pattern consisting of a list of bigrams. The test pattern is then presented to the network, and the network attempts to activate any associated bigrams and returns a list of bigrams.

Since the PDP network has learned sets of bigrams, the incomplete pattern of bigrams that the user entered is likely to activate a complete set (i.e. a specific command). However, the incomplete pattern may be ambiguous so the network will activate several sets of bigrams—in other words, several commands.

The system then determines which commands contain a substantial number of these activated bigrams and presents the list of commands to the user. In this way, the user can select the appropriate command, and the system does not have to perfectly predict the user's intended command. This scheme gives a heavier weight to longer commands which contain more bigrams. This is desirable since long commands are generally more difficult to remember. This scheme also returns a list of commands rather than a single command. It is likely that the user's input will be ambiguous and several sets of bigrams will be activated.

Bigrams are used because they allow the user to misspell some of the words but also keep some sequencing information. Single letters would allow for more spelling errors. However, a two layer network is not complex enough to learn sequencing information for words. Trigrams (three letter sequences) can also be used, but, since there are more trigrams than bigrams, there would be more units in the network, and learning would be slower. In addition, UNIX command names are often only two letters long, therefore trigrams would not encode them properly.

4.2. Implementation

A prototype system was implemented using GNU Emacs Lisp and SunNet (a neural network simulator). Emacs was used to encode and decode the set of UNIX commands for the neural network and to provide a user interface to SunNet. Due to time limitations (and other problems discussed later), the PDP history was not actually connected to the UNIX shell. Only the learning

phase and the recalling phase were implemented. The Emacs Lisp source code is shown in Appendix A.2.

Figure 4.3 shows a detailed schematic of the prototype. In the learning phase, a list of UNIX commands is learned. First the unique bigrams are found, then the commands are converted to bigrams and encoded to SunNet input format. Finally, SunNet cycles thru the input many times to learn the commands.

In the recalling phase, the user enters a string of characters then the system replies with a list of possible commands. First, the users input is decomposed into bigrams. These are then encoded to SunNet input format. SunNet tests the network and outputs a pattern which contains the activation levels of the bigrams in the network. The UNIX commands that contain a substantial number of bigrams with a high activation level are displayed. Finally, the user can select the appropriate command from this list.

Before describing the details of the two phases, SunNet will briefly be described.

4.2.1. SunNet and Emacs

A neural network simulator called SunNet (Miyata, 1987) was used to implement the network. SunNet is a general purpose tool for "constructing, running and examining a PDP (parallel distributed processing) or connectionist network...it allows you to deal with a network at a fairly high-level of conceptualization, and yet provides the flexibility to construct networks of almost arbitrary structure and size...." (Miyata, 1987)

SunNet allows the user to specify a full description of the network including the number of units, the method of connecting the units, the propagation rule, and the learning rule. SunNet requires two input files: a network file which describes the network, and a pattern file which contains the patterns to be learned. The network file used for the PDP history is shown in Appendix B. It describes a two layer network and the delta rule for learning.

The pattern file contains a series of input-output training pairs. Since this network was used as an auto-associative network, the input and output patterns are identical. Patterns can either be the integer values from 0-9, the letter "a" (corresponding to a "10"), or floating point numbers. For the PDP history, only a "0," for the minimum value, and an "a" for the maximum value was used.

For the input pattern, the position of each character represents the external input on each unit (representing a bigram) of the first layer in the network. For example, if there are four units in the input layer and the pattern "00a0" is specified, then a "0" will be given to the first two units, an "a" to the third unit, and a "0" to the last unit. Similarly, for the output pattern, the position of each character represents the activation level to be learned by each unit of the last layer.

4.2.2. Saving commands

The commands previously entered by the user are learned in the manner shown in Figure 4.3. First, all the unique bigrams that compose the

commands are found. The list of bigrams is alphabetized and stored in a file for later reference by the recalling phase.

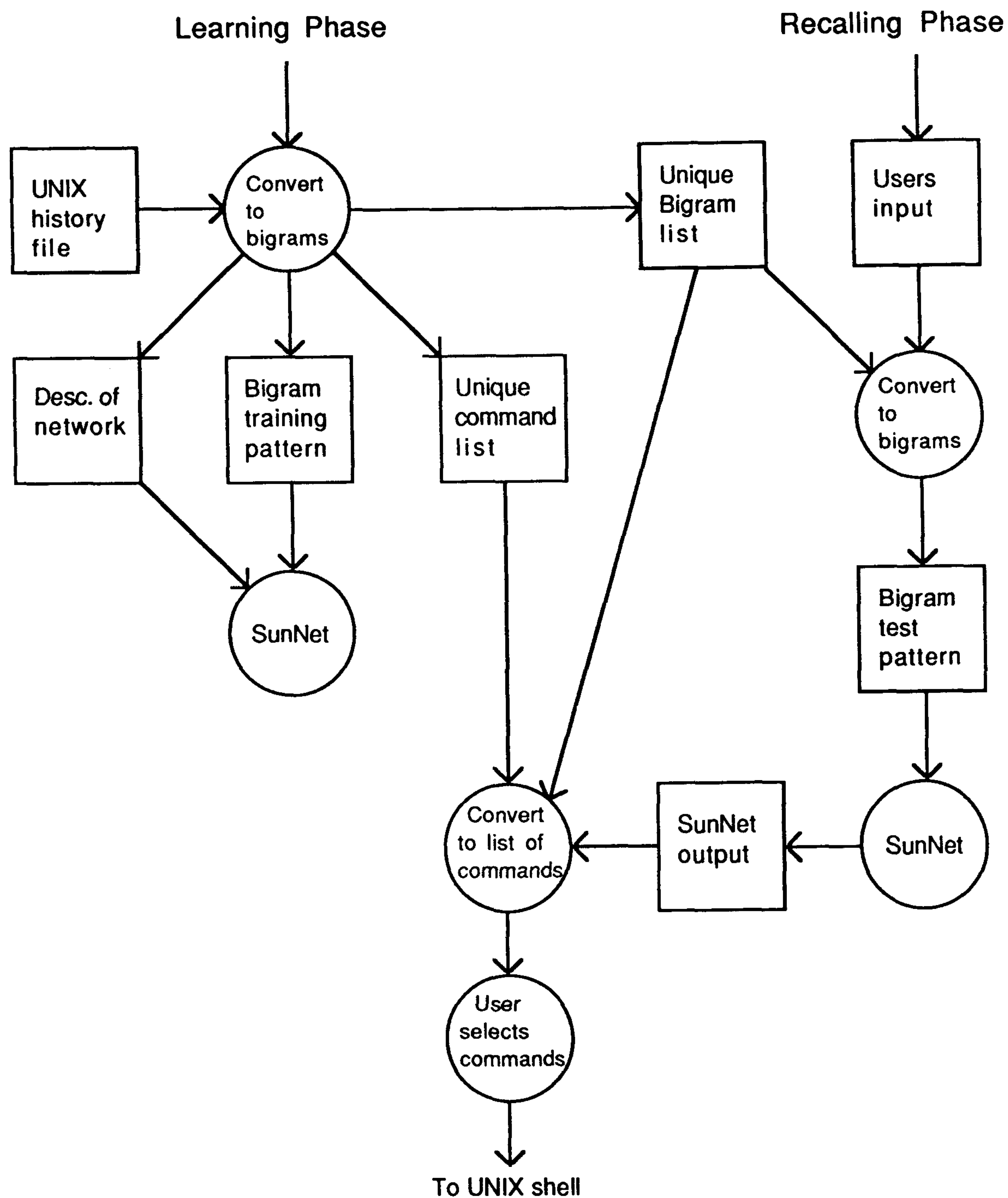


Figure 4.3: Detailed schematic of the PDP history

Next, three files are created: the network file, which is a description of the network; a unique commands file, which contains a list of commands that were previously entered but without duplicates; and the pattern file, which contains the training patterns for SunNet. The network file specifies the size of the network, the type of learning, and other parameters for the network. The unique commands file is used later for indexing in the recalling processing.

The pattern file is created in the following manner. For each command, execute the following steps:

1. Decompose the command to a list of unique bigrams.
2. Output a string of 0's and an "a" as the input pattern. The position of the "a" is determined by the position of each bigram in the list of unique bigrams. All other bigrams are represented by a "0."
3. Output an identical string of 0's and a's as the output pattern.

Finally, SunNet reads in the network file and the pattern file. The user enters the number of times that SunNet must cycle to learn the patterns. Typically, this is about 10 cycles. The learning parameter, N, was set to 0.1.

4.2.3. Recalling commands

In the recalling phase, the user attempts to recall a command by entering characters or words which might be in the command. Figure 4.3 depicts the recalling process. First, the user's input is converted to a list of bigrams. The bigrams are then converted to a training pattern as described above. This pattern is submitted to the network for activation. SunNet then produces a

list of floating point numbers. Each number corresponds to the presence of a bigram. The higher the number, the more likely the bigram appears in the command being searched for. The list of numbers is indexed according to the unique bigram file created in the learning phase.

The bigrams with the highest values (at least a 90% activation level) are looked up in the unique command list created in the learning phase. The resulting list of commands is sorted by the number of bigrams. Only those commands which contain more than one of the activated bigrams are presented to the user. There are two types of these bigrams: those that the user entered and those that SunNet activated but the user did not enter. The user entered bigrams are given twice the weight as the network activated bigrams so that commands which contain only activated bigrams are less likely to be displayed.

Finally, the user can select a command to execute from the list of commands. This last step was not implemented for two reasons. First, the simulation of a neural network was slow. A typical history requires several hours to be learned. Second, the design of the history does not allow commands to be added incrementally. That is, the entire history must be relearned after each new command is executed. Since it takes several hours to do this, it was not practical to implement the shell. The design of the shell, however, is similar to that of the conventional structured history. The user should be able to select a command from a list of commands, then edit or execute the command.

4.3. Properties of the PDP history

The PDP history has three major properties:

- It allows for misspellings when users recall a command,
- It recalls several associated commands besides the command that was desired by the user, and
- It presents a sorted list of these commands to the user. The list is sorted by the number of activated bigrams.

4.4. Summary

The PDP history consists of a PDP network wrapped in a few conventional algorithms. UNIX commands are learned by decomposing them into their constituent bigrams and then presenting these bigrams to the PDP network. The network learns to associate each bigram to one another. The user can recall a previously entered command by giving the PDP history a short string of letters. The history converts these letters to bigrams and presents the bigrams to the network. The network returns a group of bigrams that are associated to the user's bigrams and then a conventional algorithm converts this group to a list of commands. Finally, the user can select a command to be executed.

GNU Emacs was used to implement the algorithms for encoding commands to bigrams and decoding bigrams to commands. SunNet was used to implement a two-layer network to learn the commands.

Chapter 5

A Sample Session Comparing the Models

5.1. C shell history

An example comparing the usage of the existing UNIX history, the CS history, and the PDP history is shown in Table 5.1. The first column is an excerpt from a log of a user's history taken over a period of several weeks. Roughly 600 command lines were executed, but only 37 of them are shown in this table. The commands are arranged chronologically from the most recent command at the bottom, to the least recent command at the top.

The second column of Table 5.1 show which commands are saved by the existing UNIX history. An "X" means that the command is saved and is accessible to the user. For this example, the C shell environment variable, history, is set to 20 so only the last 20 commands are saved. This variable can be larger, however it is generally set to the length of the screen because the history command itself will simply scroll the entire list on the terminal.

Suppose the user wishes to re-execute the most recent "nroff" command found near the bottom of the command list. Using the C shell history, the user would type "!nroff." However, suppose the user wishes to re-execute the "nroff" command in the middle of the list. Using the C shell history, the user would be unable to access this command since it was executed several weeks ago and is no longer in the history list. (Even if it were in the history list, entering "!nroff" would retrieve the "nroff" command just recently executed and not the desired command.)

Table 5.1: A comparison of the commands that are saved by the existing history, the CS history, and the PDP history.

Commands	UNIX history	CS history	PDP history
mail			
cat oldhistory		X	X
more oldhistory		X	X
awk '{print substr($0,5)}' oldhistory		X	X
awk '{print substr($0,8)}' oldhistory		X	X
vi freq.awk		X	X
cd thesis		X	X
vi introduction.txt		X	X
ptroff -me -PT4387 -o1-2 introduction.txt		X	X
logout			X
... a few weeks later ...			
nroff -me stacks.p		X	X
nroff -ms stacks.p		X	X
... a few weeks later ...			
tail gif-l*		X	X
tail GIF-L*		X	X
rm GIF-* (		X	X
rm GIF-*		X	X
ls -la			
logout	X		X
mail	X		
ls	X		
cd temp	X	X	X
clear	X		
vi hier.c	X	X	X
cat Makefile.bsd	X	X	X
cc -O -Dstrrchr=rindex -s -o hier hier.c sftw.o	X	X	X
ls	X		
man getopt	X	X	X
cat Makefile.bsd	X	X	X
ls	X		
cd News/emacs	X	X	X
ls	X		
ls -la e*	X	X	X
nroff -o1-2 emacsdoc.nroff	X	X	X
emacs	X		
ps agu	X	X	X
mail	X		
logout	X	X	X

5.2. CS history

The CS history allows the user to access commands entered both recently and a long time ago. The third column in Table 5.1 shows the commands that are saved by the CS history. All commands, except short commands such as "ls" and duplicate commands such as the first two occurrences of the "logout" command, are saved.

Suppose the user wishes to re-execute the most recent "nroff" command. In the CS history, the user simply types "nroff" and then presses "<ctrl>-C l" for the last command feature. Suppose the user wishes to re-execute the "nroff" command found in the middle of the history. In this case, the user once again types "nroff," but instead presses "<ctrl>-C c" for a list of commands whose name contains the string "nroff."

The user can directly scroll thru the list presented by the CS history and select a command to edit then execute. In the existing C shell history, the user is only presented with a chronologically ordered list of all the commands and must specify items by number or a simple string and not by a direct manipulation technique. If a pattern matching search is used, then only the most recent command is recalled. The CS history can use a pattern match search to display either a list of commands or return the single most recent command.

5.3. PDP history

The fourth column of Table 5.1 shows the commands that are saved by the PDP history. This column is similar to the CS history column. However,

duplicate commands, such as all three occurrences of the "logout" command, are saved.

Suppose the user again wishes to re-execute the "nroff" commands. In the PDP history, the user simply enters the string "nroff." The PDP system decomposes the string to its constituent bigrams—"nr," "ro," "of," and "ff"—and then propagates these bigrams through the network. Other bigrams are activated and the system returns the list of commands shown below:

```
ptroff -me -PT4387 -ol-2 introduction.txt
nroff -ol-2 emacsdoc.nroff
nroff -ms -ol-2 stacks.p
nroff -ms stacks.p
vi introduction.txt
nroff -me stacks.p
nroff stacks.p
psroff -ms -P219 -oi-2 stacks.p
psroff -me -P219 -ol-2 stacks.p
```

This list is ordered by the number of activated bigrams. (Note that this list includes some commands that are not shown in the history excerpt of Table 5.1, however, they do occur in the entire history list of 600 commands.) Some commands, such as "vi introduction.txt," are listed but do not contain the string "nroff." "Nroff" actually activated many bigrams, among them the bigrams found in the "ptroff" command. The string "introduction.txt" occurs in both the "ptroff" command and the "vi" command, therefore the "vi" command was also returned. This ability to recall associated commands that are not directly requested is unique to the PDP nature, and is not found in the existing UNIX history nor the CS history.

The PDP history can also recall a command even if the user misspells the query for it. For example, suppose the user wishes to recall the "vi" command and enters the string "imtro" instead of "intro." The PDP history would recall the following commands:

```
vi introduction.txt
ptroff -me -PT4387 -o1-2 introduction.txt
mail -s 'Could you get the printout?' trj@trj
```

Although the "mail" command was also recalled, it contains fewer activated bigrams and is placed at the end of the list.

Both the CS history and the PDP history can recall a list of commands which contain a specific string. The PDP history, however, will also recall related commands and can account for user misspellings. The PDP history arranges the list of recalled commands according to the number of activated bigrams that the commands contain.

Chapter 6

Discussion

In summary, this project developed two prototype implementations of a structured history: a conventional database model and a parallel distributed processing model. The conventional database model was implemented in the Emacs programming environment. This model saved a single copy of all commands entered by the user that were over five characters long. The user could recall commands by pattern matching based on the name or the options, by relative time, by frequency of use, or by a combination of the above. This model successfully recalled these commands.

The parallel distributed processing model was not fully implemented (see below for a full discussion of the limitations of this model). A neural network simulator, called SunNet, and the Emacs environment was used to implement the model. This implementation saved all commands (including duplicates) entered by the user that were over five characters in length. To recall a command, the user simply entered a string of characters that might be in the command. The PDP model allowed for misspellings in the user input and recalled associated commands in addition to the desired command.

The following discussion describes the general advantages of a structured history, some of the problems of the prototype models, and future work to improve these models.

6.1. Advantages of a structured history

The existing UNIX history is very limited. It has only a few simple recalling methods and cannot be used to recall commands that were not recently executed. A structured history provides the following advantages over the existing UNIX history:

- *A list of possible commands is recalled rather than a single command.* The user can apply a variety of search strategies to recall a list of commands and can then directly select a command to edit or execute. In the existing UNIX history, the user can also display a chronological list of all commands. However, the user cannot directly select a command, and must recall the command by a number or by the beginning letters of the command.
- *In the CS history, the user has access to a wider range of search methods.* The user can be more specific and ask for a command by both name and frequency, or the user can be more general and ask for all the commands containing a specific string. Instead of just saying "I want the last command.," "I want the last command with the pattern 'foo' ," or "I want the 10th command.," the user can say, "I want the commands which I've used more than a dozen times and which accesses the file 'temp.dat' ."
- *In the PDP history, associated commands are recalled.* The user can specify a fairly ambiguous string of characters and the PDP history will recall all commands that are associated with these characters.
- *More commands are saved.* Since duplicate commands and simple commands are not saved, a greater number of the more important, hard to recall, complex commands can be stored.

6.2. Problems and shortcomings

Both the CS history and the PDP history had several problems with GNU Emacs. In addition, the PDP history had some design problems and some implementation problems with the PDP simulator.

6.2.1. *Emacs*

GNU Emacs lisp provides a powerful prototyping environment. However, it has the following drawbacks:

- *The shell mode is inadequate.* The shell mode has several deficiencies involving the terminal emulation type and can't be used to completely replace the UNIX shell.
- *There is no floating point number support.* This was most apparent in converting the output of SunNet (floating point activation levels) to UNIX commands and sorting them by activation levels.
- *It is not Common Lisp compatible.* It was sometimes difficult to manipulate Lisp structures. For example, there is a lack of some looping constructs such as dolist and dotimes.
- *It is slow.* In the CS history, it could take several minutes to read in the history from a file. In the PDP history, it often took several minutes to convert the history of commands to bigrams. (This was not a major problem since the speed of the PDP history was determined by SunNet, which could take many hours to learn a set of commands.)
- *It is oriented towards editing and buffer manipulation.* This feature is useful for an editor (which is the purpose of Emacs) but made some algorithms difficult to implement because they had to be applied to lines in an editing buffer instead of a lisp internal structure.

6.2.2. *SunNet*

The most significant problem with SunNet is that of any neural network simulator—a great deal of processing time is required to train the network. The example in Section 5.3 (Table 5.1) took many hours to learn using a Sun 3/60 workstation. SunNet is too slow for a practical implementation. Perhaps a dedicated neural network, rather then a general purpose simulator, could be used in a real implementation. Since the network used here contains only two layers and uses delta learning, it could be practical to write a PDP history extension to the C shell.

In addition, SunNet required an excessive amount of CPU and memory resources. For example, column 5 in Table 5.1 was executed remotely on a Sun 3/60 with 4 Mb. of RAM. Another user on the system console was unable to open any windows from Suntools.

6.2.3. *PDP history*

There are three major problems and shortcomings of the PDP history.

1) There is no forgetting or unlearning. Once a command is learned, there is no way to unlearn it. This problem could be partially solved by including a feature similar to that of the delete function in the CS history. The entire list of commands could be displayed and the user could delete the undesired commands. However, the PDP system would have to be reset and the entire history list of commands would have to be relearned.

2) There is no incremental learning. All commands must be learned at once. New commands entered by the user cannot be added to the history because

the new command might contain bigrams that are not in the initial list of commands. For example, for lower case letters only, there are a total of 26x26 or 676 bigrams, but as few as 300 might be used in the initial command history. If the new command contained a bigram that was not in the command history, then part of the new command could not be accessed. If all 676 bigrams were used to encode the initial command history, the network would be too slow.

3) *The PDP history has not been applied to the shell.* Since the history was slow and there was no incremental learning, no attempt was made to interface the PDP history to a UNIX shell.

4) *Sometimes completely unrelated commands are recalled.* This is both an advantage and a disadvantage. It is an advantage because associated commands are recalled. It is a disadvantage because the associations might be very weak. In the example of the previous chapter, the command "`mail -s 'Could you get the printout?' trj@trj`" was recalled given the string "imtro." This string activated too many bigrams, some totally unrelated. This excess activation is probably due to the limitations of delta learning and networks with no hidden layers (see Minsky, 1969 for a further discussion on the limitations of these networks). The network could not properly learn the distinction between the "mail" command and the "vi" command, therefore it simply grouped the two together.

Despite the drawbacks of GNU Emacs, the CS history was found to be useful to the author and several of his colleagues. The PDP history, however, was not used due to the problems discussed above. The implementations of the

two histories are only prototypes; therefore, the idea of a structured history could still be practical if implemented as part of the C shell.

6.3. Future work and other possible designs

6.3.1. *Command complexity measures*

The saving method of a structured history eventually determines the maximum number of commands that can be recalled. As a first approximation, command complexity was defined as command length. Only commands longer than five characters were saved. Other measures such as the number of words, the number of arguments, or a combination of the two could also be used. An accurate complexity measure would require a study of how users remember and forget commands.

6.3.2. *Search methods for the CS history*

Only the command name, the command options, the relative time, and the frequency of use were used as search methods in the CS history. In an informal survey of UNIX users, users had the most difficulty in recalling the arguments for a command, but had little difficulty in recalling the command name. Therefore, recalling commands by name and options was considered the most important features of a structured history. Since the existing UNIX history can list all the commands by the relative time that they were executed, the CS history kept this feature and used the relative time as a recalling method. Frequency of use was used as an example for other search methods.

Three other methods could also be considered: categories, sequencing groups, and relations. Commands could be grouped into functional categories—such as file commands, text formatting commands, and administrative

commands—or commands could be grouped into project categories such as thesis commands, homework commands, and personal commands. Some UNIX programs (command names) have an option that would classify them into one category and another option that classify them into another category. Since each command line is saved independently of other command lines, commands with the same name could be recalled separately using the category method, or recalled together using the command name pattern matching method.

Some commands are often used in a sequence. For example, the commands "vi test.c," "cc -o test test.c," and "test arg1 arg2" would be used together to debug a C program. A structured history could save these commands as a group, allowing the user to easily recall the next command in the sequence even months after the commands were first executed.

Finally, commands can be grouped by relations. A command in a structured history can be retrieved by its relationship to other commands. For example, a command name may be the result of a compilation of a C program and the command options may include the name of a file that must be created with a special program. A structured history would recall the command name, the command for compilation, and the command for creating the required file.

6.3.3. *The network design for the PDP history*

The PDP history used a distributed representation for the commands, i.e. several units in the network represent an entire UNIX command. Initially, a local representation was used. In this case, two types of units were used—bigram units and command units. Learning was associating a set of bigram

units to a specific command unit. In this method, learning was much slower and it was difficult to express command complexity (i.e. the number of activated bigrams) as a strategy for ordering the commands that were recalled.

A distributed representation, however, was difficult because sequencing information (e.g. the fact that the letter "n" comes before "r" in the command "nroff") could not easily be expressed. This was solved by using bigrams and a post-processing conventional system to convert the bigrams to commands.

Bigrams were used because they hold some sequencing information and also compensate for minor spelling errors by activating several units for a word. For example, suppose the word "emacs" is encoded. Four units are used: "em," "ma," "ac," and "cs." If the user enters "emaxs" instead, then two of the bigrams, "em" and "ma," are still the same. There is still a good chance that "emacs" will be at least partially activated. If single letters are used, then "emacs" would require five units. If the user enters "emaxs" instead, then there is a good chance that "emacs" will be activated, however, since sequencing information is not stored, many other commands with the letters "e," "m," "a," "x," or "s" will also be activated.

Other encodings, such as trigrams (three letter sequences) or full words, would not allow for as many misspellings as bigrams. In the above example, "emacs" would be encoded to only three trigrams: "ema," "mac," and "acs." A simple misspelling such as "emaxs" would only match one trigram, "ema," probably not enough to activate "emacs." In addition, there are more trigrams than bigrams. For 26 letters, there are 26x26 or 676 bigrams,

however, there are 26x26x26 or 17,576 trigrams. Therefore, there would be more units and the performance of the network would be slower.

6.3.4. Is a PDP history practical?

Is it possible to implement a practical PDP structured history? Despite the fact that SunNet was unacceptably slow, it seems plausible that an optimized network coded in C or assembly could run fast enough to be usable. However, this is only possible if the incremental learning problem is solved. Perhaps some method of learning a small range of commands after each command is entered and then relearning the entire history at the end of the session might solve this problem.

A combination of a PDP network and conventional algorithms would be the best system for a structured history. The CS history has the advantage of speed and the ability to delete commands. The PDP history, on the other hand, allows for minor spelling errors and can recall associated commands. Perhaps a system which uses a PDP model for pattern matching and a conventional model for other search methods would give the system all the advantages of the PDP nature and yet would allow the user to specifically ask for particular commands.

6.3.5. Application to other systems

A structured history is a tool that can be applied to many other systems besides UNIX. The VMS operating system, for example, has a command history mechanism called "recall" which is similar to the UNIX history except that the user can easily edit the command line and scroll through the history of commands a line at a time (VAX/VMS Manual, 1985). The Emacs environment also contains a command history. However, it too is a simple

chronological list of commands and is not very powerful. A structured approach would save more commands and provide the user with greater flexibility in recalling commands. It also allows the user to choose the command from a list of commands which meet a specific criteria.

A structured history could be applied to other operating systems, control systems, or any system that uses a complex command language and requires the user to re-enter difficult to remember commands.

References

Franklin, D. (1987). "UNIX: rights and wrongs." *UNIX papers*. Indianapolis, Indiana: Howard W. Sams and Co. 3-40.

Greenberg, S., & Witten, I. H. (1988). "How users repeat their actions on computers: principles for design of history mechanisms." *Proceedings ACM SIGCHI 1988*, 171-178.

Hebb, D. O. (1949). *The organization of behavior*. New York: Wiley.

Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Sciences*, *79*, 2554-2558.

Ishikawa, M. (1987). "Toward Interfaces which Learn User Behaviors." Unpublished report. UC San Diego.

Kochan, S. G., & Wood, P. H. (1985). *UNIX shell programming*. Indianapolis: Hayden Books.

Kohonen, T., & Ventä, O. (1988). "A content-addresssing software method for the emulation of neural networks." *IEEE international conference on neural networks*, *1*, 191-198.

McCulloch, W. S., & Pitts, W. (1943). "A logical calculus of the ideas imminent in nervous activity." *Bulletin of mathematical biophysics*, *5*, 115-133.

Minsky, M., & Papert, S. (1969). *Perceptrons*. Cambridge: MIT Press.

Miyata, Y. (1987). "SunNet." Ver. 5.2. Computer program. UC San Diego.

Norman, D. A. (1981). "The trouble with UNIX." *Datamation*, 27(12), 139-150.

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing: explorations in the microstructure of cognition*. Vol 1 and 2. Cambridge, MA: MIT press.

Sebes, J. (1987). "Comparing UNIX shells." *UNIX papers*. Indianapolis, Indiana: Howard W. Sams and Co. 123-151.

Stallman, R. (1986). *GNU emacs manual*. (6th ed.) Vol 18. Free Software Foundation.

Steele, G. L. (1984). *Common Lisp: the language.* Digital Equipment Corp.

UNIX User's Manual—Reference Guide (1984). Berkeley, CA: Computer Science Division. Ver. 4.2 Dist.

VAX/VMS DCL Dictionary (1985). Digital Equipment Corp. Ver. 4.2.

Vitter, J. S. (1984). "US&R: a new framework for redoing." *IEEE Software,* 1, 39-52.

Yang, Y. (1988). "A new conceptual model for interactive user recovery and command reuse facilities." *Proceedings ACM SIGCHI 1988,* 165-170.

Appendix A

Emacs Code

A.1. CS history

The Emacs Lisp code for the CS history is contained in the file history.el. The Emacs functions for implementing the shell was modified in three areas:

- The shell defun was modified slightly to execute a function to read in the history from a file before starting the shell.
- The defun shell-send-input was modified slightly to execute a function to add the command to the history list.
- New key bindings were added to the mode map for the shell to execute the history functions.

A new mode, histo-mode, was created. This mode was used to display the lists of commands. Key bindings were removed to make the buffer read-only and new key bindings were added for the history functions.

The file history.el is shown below:

```
;;
;; history.el
;;
;; Contains routines for a conventional structured history
;;
;; To run: Use M-x "shell" to enter the emacs inferior shell mode.
;;         All text that is entered into the shell mode will be saved
;;         by the history.
;; For help on the history mode use C-h m. (display documentation of
;;         current major mode.)
;;
(defvar histo-list nil
  "global variable containing history list of commands.")

(defvar histo-current-list nil
  "global variable containing history list of commands being
displayed.")
```

```
(defvar last-input-command nil
  "global variable containing the last command executed.")

(defconst histo-file-name "~/.history_emacs"
  "Name of file containing the history commands. Normally
is used by histo-save-list and histo-init-list to store commands and
recall them, respectively.")

(defvar histo-mode-map nil)

(defun histo-mode ()
  "Major mode for interacting with history feature of shell.

The following commands are useful for interacting with the command
history:
\\{histo-mode-map} "

  (interactive)
  (kill-all-local-variables)
  (setq major-mode 'histo-mode)
  (setq mode-name "Shell History")
  (use-local-map histo-mode-map)
  (run-hooks 'histo-mode-hook))

 (progn
  (setq histo-mode-map (make-keymap))
  (suppress-keymap histo-mode-map)
  (define-key histo-mode-map "\C-m" 'histo-send-command)
  (define-key histo-mode-map "\C-ca" 'histo-all-commands)
  (define-key histo-mode-map "\C-cc" 'histo-recall-in-other-buffer)
  (define-key histo-mode-map "\C-cf" 'histo-frequent-commands)
  (define-key histo-mode-map "\C-ch" 'describe-mode)
  (define-key histo-mode-map "\C-cq" 'histo-return-to-shell)
  (define-key histo-mode-map "\C-cs" 'histo-search-args)
  (define-key histo-mode-map "\177" 'undefined)
  (define-key histo-mode-map "a" 'histo-all-commands)
  (define-key histo-mode-map "c" 'histo-recall-in-other-buffer)
  (define-key histo-mode-map "d" 'histo-delete-command)
  (define-key histo-mode-map "f" 'histo-frequent-commands)
  (define-key histo-mode-map "h" 'describe-mode)
  (define-key histo-mode-map "\M-OA" 'histo-backward-line)
  (define-key histo-mode-map "\M-OB" 'histo-forward-line)
  (define-key histo-mode-map "n" 'histo-forward-line)
  (define-key histo-mode-map "p" 'histo-backward-line)
  (define-key histo-mode-map "q" 'histo-return-to-shell)
  (define-key histo-mode-map "s" 'histo-search-args))

 ;; Now set some keys in shell mode to access history functions

 (progn
  (define-key shell-mode-map "\C-ca" 'histo-all-commands)
  (define-key shell-mode-map "\C-cc" 'histo-complete-command)
  (define-key shell-mode-map "\C-cf" 'histo-frequent-commands)
  (define-key shell-mode-map "\C-ch" 'describe-mode)
  (define-key shell-mode-map "\C-cl" 'histo-last-command)
  (define-key shell-mode-map "\C-cs" 'histo-search-args))

 ;;
```

```
;; End of key and mode definitions.
;; Define functions:
;;

(defun histo-add-command (begin end)
  "Insert region specified by markers BEGIN and END to end of histo-
list. Saves histo-list if command is 'exit'."

  (if (string-equal (car (histo-add-command-aux 'histo-list
                                                begin end 1))
                    "exit")
      (histo-save-to-file)))

(defun histo-add-command-aux (command-list begin end freq)
  "Auxilliary function for histo-add-command. Returns the command added.
Add to COMMAND-LIST the string from BEGIN to END of current buffer with
plist 'freq=FREQ."
  (let* ((command (histo-get-command-from-buffer begin end freq))
         (element (assoc (car command) (symbol-value command-list))))
    (setq last-input-command command)
    (if (> (marker-position end)         ; save only cmds > 5 chars
           (+ (marker-position begin) 5))
        (progn
          (if element         ; command already in command-list
              (progn          ; so incre. freq of cmd by freq of element
                (put (cdr command) 'freq
                     (+ (get (cdr command) 'freq)
                        (get (cdr element) 'freq)))
                (set command-list
                     (delq element (symbol-value command-list)))))
          (set command-list     ; add to end of command-list
               (nconc (symbol-value command-list) (list command)))))
    command))

(defun histo-file (name arg freq)
  "Auxilliary function for histo-read-from-file. Creates a command and
adds it to histo-list.  Takes 3 args: NAME ARG and FREQ."
  (let* ((command-list 'histo-list)
         (command (histo-make-command name arg freq))
         (element (assoc (car command) (symbol-value command-list))))
    (setq last-input-command command)
    (if element           ; command already in command-list
        (progn            ; add 1 to frequency then delete element
          (put (cdr command) 'freq
               (+ (get (cdr command) 'freq)
                  (get (cdr element) 'freq)))
          (set command-list
               (delq element (symbol-value command-list)))))
    (set command-list     ; add to end of command-list
         (nconc (symbol-value command-list) (list command)))
    command))

(defun histo-all-commands ()
  "Displays all commands in histo-list in buffer *histo*."
  (interactive)
  (setq histo-current-list histo-list)
  (histo-display-current-list))
```

```
(defun histo-backward-line ()
  (interactive)
  (forward-line -1))

(defun histo-complete-command ()
  "Try to complete command before point using histo-list. "
  (interactive)
  (shell-get-input-region)     ; calculate last-input-start and end
  (let* ((beg (marker-position last-input-start))
         (end (marker-position last-input-end))
         (command-name (buffer-substring beg end))
         (completion (histo-match-list 'name command-name)))
    (cond ((eq completion t))
          ((null completion)
           (message "Can't find command name containing \"%s\""
                    command-name)
           (ding))
          ((= 1 (length completion))
           (message "Done")
           (delete-region beg end)
           (insert (car (car completion))))
          (t
           (message "Making completion list...")
           (setq histo-current-list completion)
           (histo-recall-in-other-buffer command-name)
           (message "Making completion list...%s" "done")))))

(defun histo-delete-command ()
  "Deletes a command in both the *histo* buffer, histo-current-list,
and histo-list."
  (interactive)
  (let ((begin (make-marker))
        (end (make-marker))
        (command nil))
    (beginning-of-line)
    (move-marker begin (point))
    (end-of-line)
    (move-marker end (point))
    (beginning-of-line)
    ; delete from buffer, histo-list, and histo-current-list
    (setq command (assoc (car (histo-get-command-from-buffer
                               begin end 1))
                         histo-list))
    (setq histo-list (delq command histo-list))
    (setq command (assoc (car (histo-get-command-from-buffer
                               begin end 1))
                         histo-current-list))
    (setq histo-current-list (delq command histo-current-list))
    (kill-line 1)))

(defun histo-display-current-list ()
  "Displys histo-current-list in *histo* buffer."
  (interactive)
  (pop-to-buffer "*histo*")
  (erase-buffer)
  (insert "  ")
  (insert (mapconcat 'car histo-current-list "\n  "))
  (histo-mode))
```

```
(defun histo-forward-line ()
  (interactive)
  (forward-line 1))

(defun histo-frequent-commands ()
  "Displays commands ordered by frequency.
If in *histo* buffer then use buffer contents, else use histo-list."
  (interactive)
  (let ((freq-commands (histo-match-list 'freq "")))
    (sort freq-commands
          '(lambda (iteml item2)
                         (< (get (cdr iteml) 'freq)
                            (get (cdr item2) 'freq))))
    (message "Most frequent commands.")
    (setq histo-current-list freq-commands)
    (histo-display-current-list)))

(defun histo-get-command-from-buffer (begin end freq)
  "Get a line from current buffer from BEGIN to END and return as a
command. Assume command has FREQ for it's frequency. Assume command is
of the form 'name args'.  Returns (commandname . name######)"
  (save-excursion
    (let ((my-begin 0)
          (name "")
          (args ""))
      ; get rid of leading spaces and set my-begin correctly
      (goto-char begin)
      (fixup-whitespace)
      ; move forward if not at beg. of line
      (if (not (bolp)) (forward-char 1))
      (setq my-begin (point))
      (forward-word 1)
      (if (eolp)  ; if at end of line then no arguments
          (setq name (buffer-substring my-begin (point)))
        (fixup-whitespace)  ; else try to figure out what the args are
        ; fixup-whitespace will add space at end of line if there are no
        ; arguements.  To correct this get rid of trailing spaces after
        ; fixing up command and arguements.  Must save the excursion
        ; since goto-char sets the point.
        (save-excursion
          (goto-char end)
          (delete-horizontal-space)
          (move-marker end (point)))
        (setq name (buffer-substring my-begin (point)))
        (if (not (eolp))   ; now check again for end of line.
            (setq args (buffer-substring (+ 1 (point)) end))))
      (histo-make-command name args freq))))

(defun histo-last-command ()
  "Puts the last command beginning with whatever command begins with."
  (interactive)
  (shell-get-input-region)
  (let* ((beg (marker-position last-input-start))
         (end (marker-position last-input-end))
         (command-name (buffer-substring beg end))
         (completion
          (car (reverse
```

```
                (all-completions command-name
                                (append histo-list
                                        (list last-input-command))))))
    (cond ((null completion)
           (message "Can't find completion for \"%s\"" command-name)
           (ding))
          (t
           (message "Done.")
           (delete-region beg end)
           (insert completion)))))

(defun histo-make-command (name args freq)
  "Make an element of histo-list with command name NAME and
arguements ARG and frequency FREQ.
Returns an alist of the form ('name args' . 'name#rnd-no#').
The symbol name#rnd-no# is name concatenated with a random number.
This symbols property list contains the name, args, and freq also."
  (let ((new-symbol (intern (concat name
                                    (int-to-string (random))))))
    (put new-symbol 'name name)
    (put new-symbol 'args args)
    (put new-symbol 'freq freq)
    (cons (progn
            (if (string-equal args "")
                name
              (concat name " " args)))
          new-symbol)))

(defun histo-match-list (property pattern)
  "Returns a list of commands whose PROPERTY matches a given PATTERN.
If current-buffer is *histo* then use histo-current-list, else
use histo-list."
  (let* ((correct-list (if (string= (buffer-name) "*histo*")
                           (copy-alist histo-current-list)
                         (copy-alist histo-list)))
         (match-list (if (string= pattern
                                  "")   ; if looking for all items
                         correct-list          ; then return all items
                       (mapcar '(lambda (item) ; else search for pattern
                                  (if (string-match pattern
                                                    (get (cdr item)
                                                         property))
                                      item))
                               correct-list))))
    (delq nil match-list)))

(defun histo-read-from-buffer (command-list)
  "Read strings from *histo* into COMMAND-LIST and return command-list."
  (goto-char (point-min))
  (let ((begin (make-marker))
        (end (make-marker))
        (freq 0))
    (while (not (eobp))
      (beginning-of-line)
      (move-marker begin (point))
      (forward-word 1)
      (setq freq (string-to-int (buffer-substring begin (point))))
        (move-marker begin (point)) ; skip over freq. to real cmd.
```

```
      (end-of-line)
      (move-marker end (point))
      (histo-add-command-aux command-list begin end
                              freq )  ; don't exit on 'exit' command
      (forward-line)))
  ; return command-list just read from buffer
  (symbol-value command-list))

(defun histo-read-from-file ()
  "Reads histo-list from file histo-file-name"
  (interactive)
  (histo-reset)
  (message "reading history from %s..." histo-file-name)
  (find-file histo-file-name)
  (rename-buffer "*histo*")
  ; new format begins with "(".
  (if (string-equal "("
                    (buffer-substring (point)
                                      (+ 1 (point))))
      (eval-current-buffer)
    (histo-read-from-buffer 'histo-list))
  (message "reading history from %s...done." histo-file-name)
  (erase-buffer))

(defun histo-recall-in-other-buffer (command-name)
  "Show selective histo-list in another buffer for COMMAND-NAME."
  (interactive "sEnter command name: ")
  (let ((match-list (histo-match-list 'name command-name)))
    (setq histo-current-list match-list))
  (histo-display-current-list))

(defun histo-reset ()
  "Reset histo-list to nil. "
  (interactive)
  (setq histo-list nil))

(defun histo-return-to-shell ()
  "Close *histo* window and return to *shell* window"
  (interactive)
  (bury-buffer)
  (delete-window)
  (switch-to-buffer "*shell*"))

(defun histo-save-to-file ()
  "Saves *histo* buffer to histo-file-name.
Format of file is 'freq command'"
  (interactive)
  (switch-to-buffer "*histo*")
  (erase-buffer)
  (histo-save-to-file-aux)
  (write-file histo-file-name)
  (kill-buffer (current-buffer)))

(defun histo-save-to-file-aux ()
  "prints a Lisp readable file of the histo-list."
  (mapcar '(lambda (item)
             (insert "(histo-file ")
             (insert (prin1-to-string (get (cdr item) 'name)))
```

```
                (insert " ")
                (insert (prinl-to-string (get (cdr item) 'args)))
                (insert " ")
                (insert (prinl-to-string (get (cdr item) 'freq)))
                (insert ")\n"))
            histo-list))

(defun histo-search-args (pattern)
  "Prompt user for a string then display commands whose arguements
contain given string."
  (interactive "sEnter string: ")
  (let ((match-list (histo-match-list 'args pattern)))
    (if (not match-list)
        (message "String not found.")
      (setq histo-current-list match-list)
      (histo-display-current-list))))

(defun histo-send-command ()
  "Send a command from histo buffer to shell buffer and
switch-to-shell."
  (interactive)
  (beginning-of-line)
  (let ((temp-mark (point))
        (command))
    (end-of-line)
    (setq command (buffer-substring temp-mark (point)))
    (bury-buffer)
    (delete-window)
    (switch-to-buffer "*shell*")
    (shell-get-input-region)
    (delete-region last-input-start last-input-end)
    (insert command)))

(defun shell-get-input-region ()
  "Sets the two markers last-input-start and last-input-end to delimit
cmd. At end of buffer, delimits all text after last output to eob.
When not at end, copies current line to the end of the buffer and
sends it, after first attempting to discard any prompt at the beginning
of the line by matching the regexp that is the value of
shell-prompt-pattern if possible.
This reg exp should start with \"^\".  Search is bounded to current
line."

  (interactive)
  (end-of-line)
  (if (eobp)
      ; if at end then delimit all text after last output
      (progn
        (move-marker last-input-start
                     (process-mark (get-buffer-process
                        (current-buffer))))
        (move-marker last-input-end (point)))
    ; else copy current line to end of buffer.
    (beginning-of-line)
    (re-search-forward shell-prompt-pattern
                       (save-excursion (end-of-line) (point))
                       t)
    (let ((copy (buffer-substring (point)
```

```
                                          (progn (forward-line 1)
                                                 (1- (point))))))
      (goto-char (point-max))
      (move-marker last-input-start (point))
      (insert copy)
      (move-marker last-input-end (point)))
    ;; Even if we get an error trying to hack the working directory,
    ;; still send the input to the subshell.
    (condition-case ()
        (save-excursion
          (goto-char last-input-start)
          (shell-set-directory))
      (error (funcall shell-set-directory-error-hook)))))

(defun shell-send-input ()
  "Send input to subshell.
At end of buffer, sends all text after last output as input to the
subshell, including a newline inserted at the end. When not at end,
copies current line to the end of the buffer and sends it, after first
attempting to discard any prompt at the beginning of the line by
matching the regexp that is the value of shell-prompt-pattern if
possible.
This reg exp should start with \"^\".  Search is bounded to current
line. Replaces same function in shell.el."

  (interactive)
  (shell-get-input-region)    ; sets last-input-start and last-input-end
  (goto-char (marker-position last-input-end))

  ; save command in histo-list
  (let ((process (get-buffer-process (current-buffer)))
        (command-start last-input-start)
        (command-end (make-marker)))
    (move-marker command-end
                 (marker-position last-input-end))

    ; send csh the command--add CR to command
    (insert ?\n)
    (move-marker last-input-end (point))
    (process-send-region process last-input-start last-input-end)
    (set-marker (process-mark process) (point))
    (copy-region-as-kill command-start command-end)
    (histo-add-command command-start command-end)
    (goto-char (point-max))))
```

A.2 The PDP history

The Emacs Lisp code for the PDP history is contained in file encode.el. These routines decompose the commands into bigrams, convert bigrams to commands, and provide an interface to SunNet. A new mode called SunNet-mode was created to run the SunNet program from within Emacs.

The file encode.el is shown below:

```
;;
;; encode.el
;;
;; created more or less on 9/22/88.  Contains functions to encode a
;; Unix command history to SunNet input format, to run SunNet, and to
;; test SunNet.
;;
;; To run model:
;;            1. open a file of UNIX commands
;;            2. run auto-assoc while in that buffer
;;            3. run run-SunNet with about 15 cycles
;;            4. wait a few hours or so
;;            5. test network with auto-test-command
;;            6. press <return> when SunNet finishes cycling since
;;               auto-test does not know how long it takes to propagate.
;;            7. Observe output in *output* buffer
;;            8. quit SunNet either from the SunNet window or use
;;               quit-SunNet
;;
(defvar SunNet nil
  "global variable containing the process pointer for SunNet.")

(defvar SunNet-mode-map nil)

(defvar SunNet-output-size 0
  "size of output for SunNet. Used by routines to create input for
SunNet.")

(defvar SunNet-input-size 0
  "size of input for SunNet. Used by routines to create input for
SunNet.")
;;
;; Set up SunNet mode
;;
(if SunNet-mode-map
    nil
  (setq SunNet-mode-map (make-sparse-keymap))
  (define-key SunNet-mode-map "\C-m" 'send-SunNet))
```

```
(defun auto-assoc ()
  "Encodes Unix history commands in current-buffer by using the bigram
list in file temp.bigrams. Puts results in file temp.pat.  The
result is a string of zeros and a's.  Does it for autoassoc. two layer."
  (interactive)
  (unique-bigrams)
  (setq SunNet-output-size SunNet-input-size)
  (goto-char (point-min))
  (let ((out-list nil))
    (goto-char (point-min))
    (while (not (eobp))
      (setq out-list (nconc out-list (list (create-training-pattern))))
      (forward-line))
    (find-file "temp.pat")
    (erase-buffer)
    (insert (mapconcat 'identity out-list "\n"))
    (save-buffer)
    (kill-buffer (current-buffer))
    (create-network-file SunNet-input-size SunNet-input-size)
    (message "%s" "auto-assoc encoding finished.")))

(defun auto-list-commands (bigram-list)
  "Returns the commands in temp.unique.cmds which contain items
in BIGRAM-LIST."
  (interactive "xEnter list of bigrams: ")
  (find-file "temp.unique.cmds")
  (goto-char (point-min))
  (let* ((number-of-lines (count-lines (point-min) (point-max)))
         (command-list ())
         (begin (make-marker))
         (end (make-marker))
         (counter 0)
         (number-of-matches 0))
    (while (not (eobp))
      (beginning-of-line)
      (set-marker begin (point))
      (end-of-line)
      (set-marker end (point))
      (setq counter (length bigram-list))
      (setq number-of-matches 0)
      (while (> counter 0)
        (setq counter (1- counter))
        (goto-char begin)
        (if (search-forward (nth counter bigram-list) end t)
            (setq number-of-matches (1+ number-of-matches))))
      (setq command-list
            (append command-list
                    (list (list number-of-matches
                                (buffer-substring begin end)))))
      (forward-line))
    (kill-buffer (current-buffer))
    (mapconcat '(lambda (item)
                  (mapconcat 'identity item " "))
               command-list
               "\n")))

(defun auto-test-command (command)
  "Tests the COMMAND using SunNet.  Uses file temp.test.pat, temp.cmds,
```

```
and temp.bigrams."
  (interactive "sEnter command to test: ")
  (create-test-pattern command)
  (if (not SunNet)
      (message "Warning, SunNet was not running.")
    (test-SunNet))
  (read-string "Press <return> when SunNet finishes cyclying...")
  (convert-SunNet-print-file)
  (switch-to-buffer "*output*")
  (erase-buffer)
  (message "%s" "manipulating bigrams...")
  ; make output into columns and sort it according to activation levels
  (call-process "sh" nil t t "/usr/users/uejio/bin/column"
                "temp.prn" "temp.bigrams")
  (delete-blank-lines)
  (goto-char (point-max))
  (goto-char (point-min))
  ; remove lines don't begin with a 10. or a 9.
  (while (not (eobp))
    (beginning-of-line)
    (if (or (string= "10." (buffer-substring (point) (+ 3 (point))))
            (string= "9."  (buffer-substring (point) (+ 2 (point)))))
        (forward-line)
        (kill-line 1)))
  (sort-numeric-fields -1 (point-min) (point-max))
  (write-region (point-min) (point-max) "temp.prn.bigrams")
  ; now get the bigram list and list commands containing bigrams
  (message "%s" "Converting bigrams to commands...")
  (remove-first-field)  ; first two fields are 0.000
  (remove-first-field)
  ; want bigrams in command to have greater weight so add them
  ; to buffer before sending buffer to be converted to commands.
  (goto-char (point-min))
  (insert command "\n")
  (goto-char (point-min))
  (let ((word-list (clean-bigrams (line-to-bigram-list))))
    (goto-char (point-min))
    (kill-line 1)
    (insert (mapconcat 'identity word-list "\n") "\n"))
  (let* ((current (current-buffer))
         (word-list (buffer-to-bigram-list))
         (string-0-or-1 "0")
         (command-list (auto-list-commands word-list)))
    (switch-to-buffer current)
    (erase-buffer)
    (insert command-list))
  ; remove lines that begin with a 0 or a 1 (space)
  (goto-char (point-min))
  (while (not (eobp))
    (beginning-of-line)
    (setq string-0-or-1 (buffer-substring (point) (+ 2 (point))))
    (if (or (string= "0 " string-0-or-1)
            (string= "1 " string-0-or-1))
        (kill-line 1)
      (forward-line)))
  (sort-numeric-fields -1 (point-min) (point-max))
  (goto-char (point-min))
  (message "%s" "Finished testing command."))
```

```
(defun buffer-to-bigram-list ()
  "Returns the contents of the current buffer as a list of words."
  (interactive)
  (goto-char (point-min))
  (let ((word-list ()))
    (while (not (eobp))
      (setq word-list
            (append word-list
                    (line-to-bigram-list)))
      (forward-line))
    word-list))

(defun clean-bigrams (word-list)
  "Removes spaces and other characters from bigrams in WORD-LIST.
word-list is a list of bigrams.  Returns cleaned list."
  (find-file "temp.clean")
  (let ((new-word-list ()))
    (mapcar '(lambda (item)
               (erase-buffer)
               (insert item)
               (goto-char (point-min))
               (if (not (re-search-forward "[ $*#()]" (point-max) t))
                   (setq new-word-list
                         (append new-word-list
                                 (list item)))))
            word-list)
    (save-buffer)
    (kill-buffer (current-buffer))
    new-word-list))

(defun convert-SunNet-print-file ()
  "Converts temp.prn to vertical instead of horizontal columns."
  (interactive)
  (find-file "temp.prn")
  (goto-char (point-min))
  (replace-regexp " " "
")
  (save-buffer)
  (kill-buffer (current-buffer)))

(defun create-network-file (input-size output-size)
  "Creates a network-file called temp.net for use with SunNet."
  (find-file "temp.net")
  (erase-buffer)
  (insert "## temp.net -- created by emacs lisp:  create-network-file\n"
          "command set print 1\n"
          "command set errorsave 50\n"
          "command set save 50\n"
          "command set max 10.0\n"
          "command set netmax 20.0\n"
          "command set floatpattern off\n"
          "define Ninput " (int-to-string input-size) "\n"
          "define Noutput " (int-to-string output-size) "\n"
          "load delta2Layer\n"
          "command pattern temp.pat\n"
          "command savefile temp.save\n")
  (save-buffer))
```

```
(defun create-test-pattern (command)
  "Converts the string COMMAND to a SunNet test pattern.  Writes
string of a's and 0's as the input pattern and a string of 0's as the
output pattern to file temp.test.pat."
  (find-file "temp.test.pat")
  (erase-buffer)
  (insert command)
  (beginning-of-line)
  (let ((in-string (concat (index-word-list (line-to-bigram-list))
                           "     "
                           (string-of-zeros SunNet-output-size))))
    (erase-buffer)
    (insert in-string))
  (save-buffer)
  (kill-buffer (current-buffer)))

(defun create-training-pattern ()
  "Converts the current line to a SunNet training pattern for an
auto-associative net. Returns a string consisting of a's and 0's."
  (let ((in-string (index-word-list (line-to-bigram-list))))
    (concat in-string "    " in-string)))

(defun index-word-list (word-list)
  "Indexes the list of words in WORD-LIST using file temp.bigrams.
Returns a string of zeros and a's where each 'a' represents a bigram
found in temp.bigrams in the word-list."
  (interactive "xEnter a list of words (lisp-expr): ")
  (let* ((current (current-buffer))
         (n (length word-list))
         (out-array ()))
    (find-file "temp.bigrams")
    (setq out-array (make-vector (count-lines (point-min)
                                              (point-max))
                                 "0"))
    (while (> n 0)
      (setq n (1- n))
      (goto-char (point-min))
      (if (search-forward (nth n word-list) (point-max) t)
          (aset out-array
                (1- (count-lines (point-min) (point))) ; index from 0
                "a")))
    (kill-buffer (current-buffer))
    (mapconcat 'identity out-array "")))

(defun line-to-bigram-list ()
  "Converts a line of text in current buffer to a list of bigrams."
  (interactive)
  (let ((word-list nil))
    (beginning-of-line)
    (while (not (eolp))
      (setq word-list
            (append word-list
                    (list (buffer-substring (point) (+ 2 (point))))))
      (forward-char 2)
      (if (not (eolp))
          (backward-char 1)))
    word-list))
```

```
(defun quit-SunNet ()
  "Sends 'quit' to SunNet and sets SunNet to nil."
  (interactive)
  (send-SunNet "quit")
  (setq SunNet nil))

(defun recompile ()
  "Recompile encode.el to encode.elc."
  (interactive)
  (byte-compile-file "encode.el")
  (load-file "encode.elc")
  (message "%s" "Recompiling done. Encode.elc loaded."))

(defun remove-first-field ()
  "Removes the first field of the current buffer."
  (interactive)
  (goto-char (point-min))
  (while (not (eobp))
    (kill-word 1)
    (delete-char 1)
    (forward-line)))

(defun run-SunNet (cycles)
  "Runs SunNet in buffer *SunNet* using network file temp.net for
CYCLES."
  (interactive "nEnter number of cycles: ")
  (switch-to-buffer "*SunNet*")
  (start-SunNet)
  (send-SunNet "network temp.net")
  (send-SunNet (concat "cycle " (int-to-string cycles))))

(defun send-SunNet (&optional command)
  "Send input to SunNet."
  (interactive)
  ; if called with optional command then insert the command first
  (if (not SunNet)
      (start-SunNet))
  (switch-to-buffer "*SunNet*")
  (goto-char (point-max))
  (if command
      (insert command))
    (insert ?\n)
    (let ((input-start (make-marker))
          (input-end (make-marker))
          (input-string nil)
          (process (get-buffer-process (current-buffer))))
      (move-marker input-start
                   (process-mark (get-buffer-process (current-buffer))))
      (move-marker input-end (point))
      (setq input-string (buffer-substring input-start input-end))
      (if (string-equal input-string "quit\n")
          (setq SunNet nil))
      (process-send-string process input-string)
      (set-marker (process-mark process) (point))
      (goto-char (point-max))))

(defun start-SunNet ()
```

```
  "Starts SunNet in buffer *SunNet* and sets the global variable SunNet
to the SunNet process."
  (interactive)
  (if SunNet
      (message "%s" "SunNet already running...")
    (setq SunNet (start-process "SunNet" "*SunNet*" "SunNet"))
    (switch-to-buffer "*SunNet*")
    (set-marker (process-mark SunNet) (point-max))
    (SunNet-mode)))

(defun SunNet-mode ()
  "SunNet mode"
  (interactive)
  (setq major-mode 'SunNet-mode)
  (setq mode-name "SunNet")
  (use-local-map SunNet-mode-map))

(defun string-of-zeros (num)
  "Returns NUM string of zeros."
  (let ((out-array (make-vector num "0")))
    (mapconcat 'identity out-array "")))

(defun test-SunNet ()
  "Tests SunNet using pattern in file temp.test.pat and procedure
activate.
Output written to temp.prn."
  (interactive)
  (send-SunNet "printlist clear")
  (send-SunNet "printlist Output")
  (send-SunNet "printfile temp.prn")
  (send-SunNet "pattern temp.test.pat activate")
  (send-SunNet "present all"))

(defun unique-bigrams ()
  "finds unique bigrams in current buffer.
Puts results in file called temp.bigrams."
  (interactive)
  (let ((temp-name (current-buffer)))
    (write-region (point-min) (point-max) "temp.cmds")
    (unique-cmds)
    (find-file "temp.bigrams")
    (erase-buffer)
    (insert-file "temp.unique.cmds")
    (goto-char (point-min))
    (while (not (eobp))
      (beginning-of-line)
      (forward-char)
      (while (not (eolp))
        (insert (buffer-substring (point) (1+ (point))) "\n")
        (forward-char))
      (forward-line))
    ; now get rid of some special characters and junk.
    (goto-char (point-min))
    (replace-regexp "[ $*&#()]" "")
    ; now get rid of lines that are shorter than 2 chars
    (goto-char (point-min))
    (let ((line-length nil))
      (while (not (eobp))
```

```
        (beginning-of-line)
        (setq line-length (length (buffer-substring (point)
                                                     (progn
                                                       (end-of-line)
                                                       (point)))))
        (if (< line-length 2)
            (progn
              (beginning-of-line)
              (kill-line 1))
          (forward-line))))
    (call-process-region (point-min)
                         (point-max)
                         "sort"
                         t t t "-u")
    (setq SunNet-input-size (count-lines (point-min) (point-max)))
    (save-buffer)
    (kill-buffer (current-buffer))
    (switch-to-buffer temp-name)))

(defun unique-cmds ()
  "finds all the unique cmds in file temp.cmds and writes it to
temp.unique.cmds"
  (interactive)
  (let ((temp-name (current-buffer))
        (cmd nil)
        (temp-buffer "")
        (current-line (make-marker)))
    (find-file "temp.cmds")
    (setq temp-buffer (buffer-substring (point-min) (point-max)))
    (kill-buffer (current-buffer))
    (find-file "temp.unique.cmds")
    (erase-buffer)
    (insert temp-buffer)
    (goto-char (point-min))
    (while (not (eobp))
      (beginning-of-line)
      (setq cmd (buffer-substring (point) (progn (end-of-line)
                                                  (point))))
      ; (sit-for 1)
      ; mark beginning of line.
      (beginning-of-line)
      (set-marker current-line (point))
      (end-of-line)
      ; should use regular expression search for ^string$.
      (if (re-search-forward (concat "^" cmd "$") (point-max) t)
          (progn
            (goto-char current-line)
            (kill-line 1))
        (forward-line))))
  (save-buffer)
  (kill-buffer (current-buffer)))
```

Appendix B

SunNet Files

B.1. Network file

The network file defines the parameters for SunNet and usually loads the file containing a generic description of the network. The parameters Ninput and Noutput are the number of units in the input layer and output layer, respectively. Various other parameters define the printing and saving features. The following is an example network file:

```
command set print 1
command set errorsave 50
command set max 10.0
command set save 50
command set netmax 20.0
command set floatpattern off
define Ninput 360
define Noutput 360
load delta2Layer
command pattern temp.pat
command savefile temp.save
```

B.2. Learning file

The learning file contains a description of a generic network, the propagation rule (called activate), and the learning rule (called learn). The learning file used for the examples in this paper is shown below.

```
## delta2Layer -- use delta rule for learning.
## define Ninput and Noutput as desired.
## this is a 2 layer program and has no hidden layer

## DEFINITIONS OF LAYERS

layer Input Ninput
layer Output Noutput
```

```
## DEFINITIONS OF INPUT/TARGET BUFFERS

target Noutput
input Ninput

## DEFINITIONS OF CONNECTIONS

connect InputOutput Input to Output symmetric

## PROCEDURE FOR ACTIVATING NETWORK FORWARD

procedure activate
        input           Input
        forward         InputOutput
        activation      Output
end

## PROCEDURE FOR TRAINING NETWORK

procedure learn
        call            activate
        target          Output
        Output:delta = (target-Output)
        learn           InputOutput
        learnbias       Output
end
```