CHAPTER 1

Introduction

Parallel distributed processing provides a new way of thinking about perception, memory, learning, and thinking—and about basic computational mechanisms for intelligent information processing in general. These new ways of thinking have all been captured in simulation models. Our own understanding of parallel distributed processing (PDP) has come about largely through hands-on experimentation with these models. And, in teaching PDP to others, we have discovered that their understanding is enhanced through the same kind of hands-on simulation experience.

This handbook is intended to help a wider audience gain this kind of experience. It makes many of the simulation models discussed in the two PDP volumes (McClelland, Rumelhart, & the PDP Research Group, 1986; Rumelhart, McClelland, & the PDP Research Group, 1986) available in a form that is both accessible and easy to use. The handbook also provides what we hope are relatively accessible expositions of some of the main mathematical results that underlie the simulation models. And it provides a number of prepared exercises to help the reader begin exploring the simulation programs.

This handbook is intended for use in conjunction with the two PDP volumes, particularly for users new to PDP. However, readers who already have some familiarity with PDP models should find that it is possible to use the simulation programs without referring to the PDP volumes. Most important for readers who are new to PDP is the introduction to the PDP framework found in Chapters 1 to 4 of the first PDP volume. Other chapters in the PDP volumes are less essential, since this handbook generally reviews specific material where relevant. However, the PDP volumes generally delve more deeply into the relevant theoretical and empirical

background. Rather than repeat this material, we give pointers throughout the handbook to relevant material from the PDP volumes.

In this chapter, we provide some general information about the use of this handbook. We begin by describing the nature of the software that accompanies this handbook and the hardware you will need to run it. Then we describe what is in each chapter and how the chapters are organized. The final sections of this chapter describe some general conventions and design decisions we have made to help the reader make the best possible use of the handbook and the software that comes with it.

THE PROGRAMS: WHAT IS PROVIDED AND WHAT IS NEEDED TO RUN THEM[1]

This book comes with two 5¼" floppy disks, which contain a set of seven simulation programs as well as auxiliary files that are needed to execute the exercises described in the following chapters. The disks also have utilities for making simple graphs from saved results. In keeping with our goal of maximum accessibility, our programs are compiled for use on IBM PCs or PC-compatible hardware.

We also provide the source code for the programs (written in C) so that users can modify the programs and adapt them for their own purposes. This also makes it possible to copy the programs to more powerful computers, where they may be recompiled.

The minimal hardware requirements to run the programs are

- An IBM PC or PC-compatible computer with two floppy disk drives or one floppy and one Winchester disk drive.
- A standard monochrome 24 line by 80 character display.
- At least 256 kbytes of memory.
- The MS-DOS operating system (Version 2.0 or higher).

If you are using a two-floppy system, you will need several floppy disks. Typically, one floppy will hold two programs in the unpacked, ready-to-run state, together with the relevant auxiliary files. To carry out some of the exercises, you will need to be able to edit files; that is, you will need a text editor. It is also preferrable to have more than 256 kbytes of memory, but this is not essential for running any of the basic exercises.

---

[1] Please see the software license agreement at the back of this handbook for several important disclaimers an for licensing information concerning the use, modification, and dissemination of this software.

For use on PCs, a math coprocessor (e.g., the 8087) is strongly recommended. All of the programs do extensive floating-point computation and they will run much faster with the coprocessor than without it. On PCs without a coprocessor, some of the exercises in Chapters 2, 5, and 7 will run slower than is optimal for interactive experimentation.

For recompilation on PCs and PC-compatibles, we recommend Microsoft C, which is what we used to produce the PC-executable versions of the programs. We cannot guarantee that other compilers will offer all of the necessary libraries (especially input/output handling, interrupt handling, and math functions such as *exp* and *square root*) or that they will not have bugs we have not encountered. For recompilation on UNIX systems, all that is required is the Portable C compiler, augmented by the CURSES screen-oriented input/output package. On UNIX systems, the programs will run on regular terminals (such as the DEC VT-100 or Zenith Z19) or under terminal-emulation programs on workstations such as IBM-RTs, MicroVAXes, or Suns.

OVERVIEW OF THE HANDBOOK

The handbook consists of seven chapters, including this brief introduction, plus several appendixes. Chapters 2 and 3 are devoted to models that focus primarily on *processing*. In Chapter 2, we describe a class of PDP models called interactive activation and competition models. Models of this kind have been explored by a number of investigators, including ourselves and Grossberg (1976, 1978, 1980). The chapter goes over some of the basic mathematical properties of this sort of model and uses the model to illustrate many of the basic processing capabilities of PDP networks. The chapter then introduces the reader to our simulation programs through the **iac** program. This program implements the interactive activation and competition mechanism and applies it to the problems of memory retrieval, spontaneous generalization, and default assignment using the "Jets and Sharks" example described in Chapter 1 of the PDP volumes (originally from McClelland, 1981). (Henceforth, we will refer to chapters in the PDP volumes by *PDP:N*, where *N* is the chapter number. Chapters 1-13 are in Volume 1; Chapters 14-26 are in Volume 2).

Chapter 2 also serves as an introduction to the package of programs as a whole. Commands that are used in all of the programs are described there, and the exercises are set up to give the reader some general facility with the package, as well as specific experience with interactive activation and competition networks.

Chapter 3 considers several related models that fall into the broad category of constraint satisfaction models. These include what we call the *schema* model (*PDP:14*), the Boltzmann machine (*PDP:7*), and the

harmonium (*PDP:6*). The chapter uses several of the examples described in the original chapters, allowing the reader to replicate some of the basic simulations that can be found there.

Chapters 4, 5, and 6 describe PDP models of learning. They can be taken up after Chapter 2 if desired. In Chapter 4, two classical learning rules are introduced: the *Hebb* rule and the *delta* rule. These learning schemes are applied in a simulation program called **pa** that implements a classic type of PDP network, the pattern associator, a simple one-layer feedforward network. This class of networks is analyzed in *PDP:9* and *PDP:11*, and is applied to psychological issues, such as the basis of lawful behavior, in *PDP:18* and *PDP:19*.

The network architecture simulated in Chapter 4 involves only a single layer of modifiable connections. Chapter 5 generalizes the architecture to multilayer, feedforward networks, and introduces mechanisms for training *hidden* units—processing units that do not receive any direct input from the outside. This chapter focuses primarily on the **bp** program, which implements the back propagation algorithm introduced in *PDP:8*.

In Chapter 6, two other architectures for learning are considered: the auto-associator and the competitive-learning network. The auto-associator has been used most extensively by James Anderson (1977) and Kohonen (1977); some applications of the auto-associator to issues of learning and memory are discussed in *PDP:17* and *PDP:25*. The competitive-learning scheme (and variants of it) have also been widely studied (e.g., von der Malsberg, 1973, and Grossberg, 1976); our applications of this scheme are described in *PDP:5*. Things are set up so that readers can proceed from the **pa** model described in Chapter 4 directly to any of the other learning models without missing any essential information.

Chapter 7 considers the use of PDP models to simulate psychological phenomena and presents the **ia** simulation program, which implements the interactive activation model of visual word recognition. This model is mentioned in *PDP:1* and *PDP:16*, and is described in detail in two earlier publications (McClelland & Rumelhart, 1981; Rumelhart & McClelland, 1982). This chapter can be taken up immediately after Chapter 2 if desired.

The appendixes provide reference information that will be of use throughout the book. Appendix A describes how to unpack the programs for the PC and how to set up working directories in which to run the exercises. Appendix B provides a command summary, listing all of the commands along with the programs in which they are available, with a brief description of each and pointers to more information. Appendix C describes the construction of files containing network and display specifications for use with the various programs. Appendix D explains how to use the utility programs that are supplied for making simple graphs. Appendix E provides some feedback on the outcome and interpretation of selected exercises. Appendix F provides an overview of the source code for readers

with a background in C who wish to modify and recompile the programs. Appendix G explains how to recompile the programs for various computers.

ORGANIZATION OF EACH CHAPTER

Each chapter begins with a brief overview, followed by one or more parts, each devoted to a model or a set of related models. Each part begins with a theoretical background section that is designed to provide an accessible introductory presentation of the relevant mathematical background for the models described in this part of the chapter. After the background section, one or more related models are presented. Each model begins with a description of the assumptions of the model and any variations on these assumptions that will be considered. This is followed by a description of the implementation of the essential routines that carry out the key computations prescribed by the model and a description of how the computer simulation of the model is to be run. The description of how the model is run consists of an overview explaining basically how the simulation is to be used, followed by a list of the commands and variables that need be understood to use the simulation program that implements the model. Finally, the presentation of each model ends with a series of exercises, the first of which is used as an example to illustrate in detail how to run the model. The exercises are generally accompanied by hints, which are intended to make sure the reader knows what is needed to carry out the exercise, both in terms of the commands needed to get the program to do what is necessary and in terms of any more conceptual points that might be relevant.

MODELS AND PROGRAMS

In general, the relation of models to programs may be many to one. That is, more than one model may be implemented by the same program; the different models are implemented by means of switches that alter the program's behavior. This makes more efficient use of disk space and cuts down some on the number of different programs that need to be learned about. Furthermore, the programs generally make use of the same interface and display routines, and most of the commands are the same from one program to the next.

In view of the similarity between the simulation models, the information that is given when each new program is introduced is restricted primarily to what is new. Readers who wish to dive into the middle of the book, then, may find that they need to refer back to commands or features that were

introduced earlier. The command summary in Appendix B should help make this as painless as possible.

SOME GENERAL CONVENTIONS AND CONSIDERATIONS

In planning this handbook, we had to make some design decisions and to adopt some fairly arbitrary conventions. Here we will describe some of the general conventions that are used in the book and in the computer programs.

Mathematical Notation

We have adopted a mathematical notation that is internally consistent within this handbook and that facilitates translation between the description of the models in the text and the conventions used to access variables in the programs. Unfortunately, this means that the notation is not always consistent with that introduced in the relevant chapters of the PDP volumes. Here follows an enumeration of the key features of the notation system we have adopted. We begin with the conventions we have used in writing equations to describe models and in explicating their mathematical background.

Scalars. Scalar (single-valued) variables are given in italic typeface. The names of parameters are chosen to be mnemonic words or abbreviations where possible. For example, the decay parameter is called *decay*.

Vectors. Vector (multivalued) variables (e.g., the vector of activations of a set of units) are given in boldface; for example, the external input pattern is called **extinput**. An element of such a vector is given in italic typeface with a subscript. Thus, the ith element of the external input is denoted $extinput_i$. Vectors are often members of larger sets of vectors; in this case, a whole vector may be given a subscript. For example, the ith input pattern in a set of patterns would be denoted $\mathbf{ipattern}_i$.

Weight matrices. Matrix variables are given in uppercase boldface; for example, a weight matrix might be denoted **W**. An element of a weight matrix is given in lowercase italic, subscripted first by the row index and then by the column index. The row index corresponds to the index of the receiving unit, and the column index corresponds to the index of the sending unit. Thus the weight to unit i from unit j would be found in the jth column of the ith row of the matrix, and is written w_{ij}.

Counting. We follow the C language convention and count from 0. Thus if there are n elements in a vector, the indexes run from 0 to $n-1$. Time is a bit special in this regard. Time 0 (t_0) is the time before processing begins; the state of a network at t_0 can be called its "initial state." Time counters are incremented as soon as processing begins within each time step.

Pseudo-C Code

In the chapters, we occasionally give pieces of computer code to illustrate the implementation of some of the key routines in our simulation programs. The examples are written in "pseudo-C"; details such as declarations are left out. Note that the pseudocode printed in the text for illustrating the implementation of the programs is generally not identical to the actual source code; the program examples are intended to make the basic characteristics of the implementation clear rather than to clutter the reader's mind with the details and speed-up hacks that are to be found in the actual programs.

Several features of C need to be understood to read the pseudo-C code. These are listed below.

Comments. Comments in C programs begin with "/*" and end with "*/". Thus the following would be treated as a comment by the compiler:

```
/* This is a comment */
```

We use this convention to introduce comments into the pseudocode so that the code is easier for you to follow.

If statements. Often in the pseudocode we will make use of *if* statements. These should be fairly self-explanatory in most cases. Sometimes, however, we use the form:

```
if (x)
```

where x is some variable. This expression means "if the value of x is not equal to 0." For flag variables, which have the value 0 when off, this corresponds to "if the flag is on." Another notation for the same thing would be

```
if (x != 0)
```

The exclamation point means "not."

Semicolons and curly braces. Semicolons and curly braces are key features of C syntax. The semicolon is used to terminate a statement. Open- ("{") and close- ("}") curly braces are used to group statements together to be treated as a single statement. Thus

```
if (expression) statement;
```

can be expanded to

```
if (expression) {
        statement;
        statement;
        . . .
}
```

Curly braces are also used to bracket the body of a subroutine.

C loop constructs. In our programs the typical loop construct is the *for* loop. *For* loops look like this:

```
for (i = 0; i < n ; i++) {
      statements;
}
```

The parentheses contain three special statements called the initialization statement, the end test, and the incrementation statement, respectively. In this example we initialize the variable i to 0. The end test is a statement that is evaluated at the beginning of each pass, before executing the statements in the loop; if the end test fails, control passes to the statement after the closing curly brace. In this case, the end test fails when the value of i is greater than or equal to n. The incrementation statement is executed at the end of each pass through the loop before the end test is done to see if the loop will be executed again. The notation

```
i++
```

indicates incrementation of the index i by 1. Thus, the above expression executes the statements enclosed in the curly braces once for each value of i from 0 through $n-1$, for a total of n passes through the loop.

Array indexes. Array indexes in C are enclosed in square brackets, with the fastest moving array element in the right-most position. Thus,

```
w[i][j]
```

refers to the element in the *i*th row and *j*th column of the array. Contiguous elements in the array are the contiguous members of the same row. The above notation, then, is the notation that refers to the weight to unit *i* from unit *j*, and corresponds to w_{ij}.

Incrementing and related constructs. We have already described the notation for incrementing a variable by 1. To increment a variable by the value of an arbitrary expression, the notation is

<variable> += <expression>;

Thus,

*x += a*b;*

means "increment *x* by *a* times *b*." There are also related expressions for decrementing (–=), multiplying (*=), and dividing (/=). Thus,

x /= 7;

means "set *x* to *x* divided by 7."

The reader should note that there are a number of features and conventions in C that are exploited extensively in the actual code that can only be understood with some background in this language. Users who do not know C will not be able to interpret the actual code. The place to go for this background is the book *The C Programming Language* by Kernighan and Ritchie (1978).

Computer Programs and User Interface

Our goals in writing the programs were to make them both as flexible as possible and as easy as possible to use, especially for running the core exercises discussed in each chapter of this book. We have achieved these somewhat contradictory goals as follows. Flexibility is achieved by allowing the user to specify the details of the network configuration and of the layout of the displays shown on the screen at run time, via files that are read and interpreted by the program. Ease of use is achieved by providing the user with the files to run the core exercises and by keeping the command interface and the names of variables consistent from program to program wherever possible. Full exploitation of the flexibility provided by the programs requires the user to learn how to construct network configuration files and display configuration (or template) files, but this is only necessary when the user wishes to apply a program to some new problem of his or her own.

Another aspect of the flexibility of the programs is their permissiveness. In general, we have allowed the user to examine and set as many of the variables in each program as possible, including basic network configuration variables that should not be changed in the middle of a run. The worst that can happen is that the programs will crash under these circumstances; it is, therefore, wise not to experiment with changing them if losing the state of a program would be costly.

BEFORE YOU START

Before you dive into your first PDP model, we would like to offer both an exhortation and a disclaimer. The exhortation is to take what we offer here, not as a set of fixed tasks to be undertaken, but as raw material for your own explorations. We have presented the material following a structured plan, but this does not mean that you should follow it any more than you need to to meet your own goals. We have learned the most by experimenting with and adapting ideas that have come to us from other people rather than from sticking closely to what they have offered, and we hope that you will be able to do the same thing. The flexibility that has been built into these programs is intended to make exploration as easy as possible, and we provide source code so that users can change the programs and adapt them to their own needs and problems as they see fit.

The disclaimer is that we cannot be sure the programs are perfectly bug free. They have all been extensively tested and they work for the core exercises; but it is possible that some users will discover problems or bugs in undertaking some of the more open-ended extended exercises. If you have such a problem, we hope that you will be able to find ways of working around it as much as possible or that you will be able to fix it yourself. In any case, please let us know of the problems you encounter.[2] While we cannot offer to provide consultation or fixes for every reader who encounters a problem, we will use your input to improve the package for future users.

---

[2] Send bug reports, problems, and suggestions to PDP Software Inquiries, c/o Texts Manager, MIT Press, 55 Hayward Street, Cambridge, MA 02142. Enclose a stamped, self-addressed envelope, and we will send an acknowledgment of your problem along with accumulated advice, fixes, work-arounds, and information about availability of subsequent releases.