Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises

James L. McClelland

Printer-Friendly PDF Version

Second Edition, DRAFT
Send comments and corrections to:
mcclelland@stanford.edu

January 4, 2011

Contents

Preface

This work represents a continuing effort to make parallel-distributed processing models accessible and available to all who are interested in exploring them. The initial inspiration for the handbook and accompanying software came from the students who took the first version of what I called "the PDP class" which I taught at Carnegie Mellon from about 1986 to 1995. Dave Rumelhart contributed extensively to the first edition (McClelland and Rumelhart, 1988), and of course the book incorporated many of the insights and exercises that David contributed to the original PDP Books (Rumelhart et al., 1986; McClelland et al., 1986).

In the mid-1990's, I moved on to other teaching commitments and turned teaching of the course over to David Plaut. Dave used the PDP handbook and software initially, but, due to some limitations in coverage, shifted over to using the LENS simulation environment (Rohde, 1999). Rohde's simulator is very fast and is highly recommended for full strengh, large-training-set, neural network simulations. My lab is now maintaining a version of LENS, available by clicking 'Source Code' at this link.

Upon my move to Stanford in the fall of 2006 I found myself teaching the PDP class again, and at that point I decided to update the original handbook. The key decisions were to keep the core ideas of the basic models as they were originally described; re-implement everything in MATLAB; update the book by adding models that had become core parts of the framework as I know it in the interim; and make both the handbook and the software available on line.

The current version of the handbook is a work in progress. My goal is to finalize the existing material during the first few months of 2010, so that it can then be treated as a stable base for further extensions by others. Information on installation of the software is provided in Appendix A. Appendix B presents a step-by-step example showing how a user can create a simple back-propagation network, and Appendix C offers a User's Guide, approximating an actual reference manual for the software itself. The hope is that, once the framework is in place, we can make it easy for others to add new models and exercises to the framework. If you have one you'd like us to incorporate, please let me know and I'll be glad to work with you on setting it up.

Before we start, I'd like to acknowledge the people who have made the new version of the PDP software a reality. Most important are Sindy John, a programmer who has been working with me for nearly 5 years, and Brenden Lake,

a former Stanford Symbolic Systems major. Sindy had done the vast majority of the coding in the current version of the pdptool software, and wrote the User's Guide. Brenden helped convert several chapters, and added the material on Kohonen networks in Chapter 6. He has also helped tremendously with the implementation of the on-line version of the handbook. Two other Symbolic Systems undergraduates also contributed quite a bit: David Ho wrote the MATLAB tutorial in Chapter 1, and Anna Schapiro did the initial conversion of Chapter 3.

It is tragic that David Rumelhart is no longer able to contribute, leaving me in the position as sole author of this work. I have been blessed and honored, however, to work with many wonderful collaborators, post-docs, and students over the years, and to have benefited from the insights of many others. All these people are the authors of the ideas presented here, and their names will be found in references cited throughout this handbook.

Jay McClelland
Stanford, CA
January, 2010

Chapter 1

Introduction

1.1 WELCOME TO THE NEW PDP HANDBOOK

Several years ago, Dave Rumelhart and I first developed a handbook to introduce others to the parallel distributed processing (PDP) framework for modeling human cognition. When it was first introduced, this framwork represented a new way of thinking about perception, memory, learning, and thought, as well as a new way of characterizing the computational mechanisms for intelligent information processing in general. Since it was first introduced, the framework has continued to evolve, and it is still under active development and use in modeling many aspects of cognition and behavior.

Our own understanding of parallel distributed processing came about largely through hands-on experimentation with these models. And, in teaching PDP to others, we discovered that their understanding was enhanced through the same kind of hands-on simulation experience. The original edition of the handbook was intended to help a wider audience gain this kind of experience. It made many of the simulation models discussed in the two PDP volumes (Rumelhart et al., 1986; McClelland et al., 1986) available in a form that is intended to be easy to use. The handbook also provided what we hoped were accessible expositions of some of the main mathematical ideas that underlie the simulation models. And it provided a number of prepared exercises to help the reader begin exploring the simulation programs.

The current version of the handbook attempts to bring the older handbook up to date. Most of the original material has been kept, and a good deal of new material has been added. All of simulation programs have been implemented or re-implemented within the MATLAB programming environment. In keeping with other MATLAB projects, we call the suite of programs we have implemented the *PDPTool* software.

Although the handbook presents substantial background on the computational and mathematical ideas underlying the PDP framework, it should be used

in conjunction with additional readings from the PDP books and other sources. In particular, those unfamiliar with the PDP framework should read Chapter 1 of the first PDP volume (Rumelhart et al., 1986) to understand the motivation and the nature of the approach.

This chapter provides some general information about the software and the handbook. The chapter also describes 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. Information on how to set up the software (Appendix A), and a user's guide (Appendix C), are provided in Appendices. At the end of the chapter we provide a brief tutorial on the MATLAB computing environment, within which the software is implemented.

1.2 MODELS, PROGRAMS, CHAPTERS AND EXCERCISES

The PDPTool software consists of a set of programs, all of which have a similar structure. Each program implements several variants of a single PDP model or network type. The programs all make use of the same interface and display routines, and most of the commands are the same from one program to the next.

Each program is introduced in a new chapter, which also contains relevant conceptual background for the type of PDP model that is encompassed by the program, and a series of excercises that allow the user to explore the properties of the models considered in the chapter.

In view of the similarity between the simulation programs, 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 *User's Guide* provides another means of learning about specific features of the programs.

1.2.1 Key Features of PDP Models

Here we briefly describe some of the key features most PDP models share. For a more detailed presentation, see Chapter 2 of the PDP book (Rumelhart et al., 1986).

A PDP model is built around a simulated artificial neural network, which consists of units organized into *pools*, and connections among these units organized into *projections*. The minimal case (shown in Figure 2.1) would be a network with a single pool of units and a single projection from each unit in the network to every other unit. The basic idea is that units propagate excitatory and inhibitory signals to each other via the weighted connections. Adjustments may occur to the strengths of the connections as a result of processing. The units and connections constitute the architecture of the network, within which these processes occur. Units in a network may receive external inputs (usually from

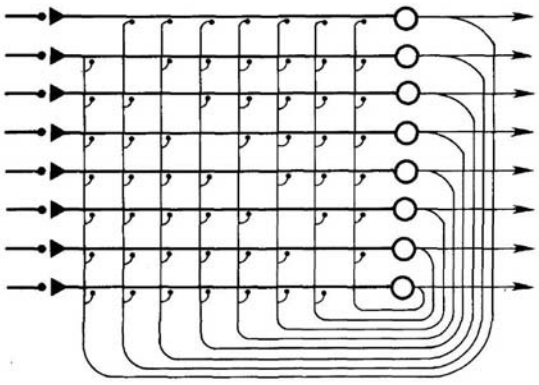

Figure 1.1: A simple PDP network consisting of one pool of units, and one projection, such that each unit recieves connections from all other units in the same pool. Each unit also can receive external input (shown coming in from the left). If this were a pool in a larger network, the units could receive additional projections from other pools (not shown) and could project to units in other pools (as illustrated by the arrows proceeding out of the units to the right. (From Figure 1, p. 162 in McClelland, J. L.& Rumelhart, D. E. (1985). Distributed memory and the representation of general and specific information. *Journal of Experimental Psychology: General, 114*, 159-197. Copyright 1985 by the American Psychological Association. Permission Pending.)

the network's environment, described next), and outputs may be propagated out of the network.

A PDP model also generally includes an environment, which consists of patterns that are used to provide inputs and/or target values to the network. An input pattern specifies external input values for a pool of units. A target pattern specifies desired target activation values for units in a pool, for use in training the network. Patterns can be grouped in various ways to structure the input and target patterns presented to a network. Different groupings are used in different programs.

A PDP model also consists of a set of processes, including a *test* process and possibly a *train* process, as well as ancillary processes for loading, saving, displaying, and re-initializing. The *test* process presents input patterns (or sets of input patterns) to the network, and causes the network to process the patterns, possibly comparing the results to provided target patterns, and possibly computing other statistics and/or saving results for later inspection. Processing generally takes place in a single step or through a sequence of processing cycles. Processing consists of propagating activation signals from units to other units, multiplying each signal by the connection weight on the connection to the receiving unit from the sending unit. These weighted inputs are summed at the receiving unit, and the summed value is then used to adjust the activations

of each recieving unit for the next processing step, according to a specified activation function. A *train* process presents a series of input patterns (or sets of input patterns), processes them using a process similar to the test process, then possibly compares the results generated to the values specified in target patterns (or sets of provided target patterns) and then carries out further processing steps that result in the adjustment of connections among the processing units.

The exact nature of the processes that take place in both training and testing are essential ingredients of specific PDP models and will be considered as we work through the set of models described in this book. The models described in Chapters 2 and 3 explore processing in networks with modeler-specified connection weights, while the models described in most of the later chapters involve learning as well as processing.

1.3 SOME GENERAL CONVENTIONS AND CONSIDERATIONS

1.3.1 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. The notation is not always consistent with that introduced in the chapters of the PDP volumes or other papers. 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 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 jth input pattern in a set of patterns would be denoted $\mathbf{ipattern_j}$.

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 MATLAB language convention and count from 1. Thus if there are n elements in a vector, the indexes run from 1 to n. 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.

1.3.2 Pseudo-MATLAB 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-MATLAB"; 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 would be found in the actual programs.

Several features of MATLAB need to be understood to read the pseudo-MATLAB code and to work within the MATLAB environment. These are listed in the MATLAB mini-tutorial given at the end of this chapter. Readers unfamiliar with MATLAB will want to consult this tutorial in order to be able to work effectively with the PDPTool Software.

1.3.3 Program Design 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 handbook. 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 program 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.

1.3.4 Exploiting the MATLAB Envioronment

It should be noted that the implementation of the software within the MATLAB environment provides two sources of further flexibility. First, users with a full MATLAB licence have access to the considerable tools of the MATLAB environment available for their use in perparing inputs and in analysing and visualizing outputs from simulations. We have provided some hooks into these visualization tools, but advanced users are likely to want to exploit some of the features of MATLAB for advanced analysis and visualization.

Second, because all of the source code is provided for all programs, it has proved fairly straightforward for users with some programming experience to delve into the code to modify it or add extensions. Users are encouraged to dive in and make changes. If you manage the changes you make carefully, you should be able to re-implement them as patches to future updates.

1.4 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 how of the problems you encounter (Send bug reports, problems, and suggestions to Jay McClelland at mcclelland@stanford.edu). 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.

1.5 MATLAB MINI-TUTORIAL

Here we provide a brief introduction to some of the main features of the MATLAB computing environment. While this should allow readers to understand basic MATLAB operations, there are a many features of MATLAB that are not covered here. The built-in documentation in MATLAB is very thorough, and users are encouraged to explore the many features of the MATLAB environment after reading this basic tutorial. There are also many additional MATLAB tutorials and references available online; a simple Google search for 'MATLAB tutorial' should bring up the most popular ones.

1.5.1 Basic Operations

Comments. Comments in MATLAB begin with "%". The MATLAB interpreter ignores anything to the right of the "%" character on a line. We use this convention to introduce comments into the pseudocode so that the code is easier for you to follow.

```
% This is a comment.
y = 2*x + 1 % So is this.
```

Variables. Addition ("+"), subtraction ("-"), multiplication ("*"), division ("/"), and exponentiation ("^") on scalars all work as you would expect, following the order of operations. To assign a value to a variable, use "=".

```
Length = 1 + 2*3 % Assigns 7 to the variable 'Length'.
square = Length^2 % Assigns 49 to 'square'.
triangle = square / 2 % Assigns 24.5 to 'triangle'.
length = Length - 2 % 'length' and 'Length' are different.
```

Note that MATLAB performs actual floating-point division, not integer division. Also note that MATLAB is case sensitive.

Displaying results of evaluating expressions. The MATLAB interpreter will evaluate any expression we enter, and display the result. However, putting a semicolon at the end of a line will suppress the output for that line. MATLAB also stores the result of the latest expression in a special variable called "ans".

```
3*10 + 8 % This assigns 38 to ans, and prints 'ans = 38'.
3*10 + 8; % This assigns 38 to ans, and prints nothing.
```

In general, MATLAB ignores whitespace; however, it is sensitive to line breaks. Putting "..." at the end of a line will allow an expression on that line to continue onto the next line.

```
sum = 1 + 2 - 3 + 4 - 5 + ... % We can use '...' to
      6 - 7 + 8 - 9 + 10 % break up long expressions.
```

1.5.2 Vector Operations

Building vectors Scalar values between "[" and "]" are concatenated into a vector. To create a row vector, put spaces or commas between each of the elements. To create a column vector, put a semicolon between each of the elements.

```
foo = [1 2 3 square triangle] % row vector
bar = [14, 7, 3.62, 5, 23, 3*10+8] % row vector
xyzzy = [-3; 200; 0; 9.9] % column vector
```

To transpose a vector (turning a row vector into a column vector, or vice versa), use "'".

```
foo' % a column vector
[1 1 2 3 5]' % a column vector
xyzzy' % a row vector
```

We can define a vector containing a range of values by using colon notation, specifying the first value, (optionally) an increment, and the last value.

```
v = 3:10 % This vector contains [3 4 5 6 7 8 9 10]
w = 1:2:10 % This vector contains [1 3 5 7 9]
x = 4:-1:2 % This vector contains [4 3 2]
y = -6:1.5:0 % This vector contains [-6 -4.5 -3 -1.5 0]
z = 5:1:1 % This vector is empty
a = 1:10:2 % This vector contains [1]
```

We can get the length of a vector by using "`length()`".

```
length(v) % 8
length(x) % 3
length(z) % 0
```

Accessing elements within a vector Once we have defined a vector and stored it in a variable, we can access individual elements within the vector by their indices. Indices in MATLAB start from 1. The special index 'end' refers to the last element in a vector.

```
y(2) % -4.5
w(end) % 9
x(1) % 4
```

We can use colon notation in this context to select a range of values from the vector.

```
v(2:5) % [4 5 6 7]
w(1:end) % [1 3 5 7 9]
w(end:-1:1) % [9 7 5 3 1]
y(1:2:5) % [-6 -4.5 0]
```

In fact, we can specify any arbitrary "index vector" to select arbitrary elements of the vector.

```
y([2 4 5]) % [-4.5 -1.5 0]
v(x) % [6 5 4]
w([5 5 5 5 5]) % [9 9 9 9 9]
```

Furthermore, we can change a vector by replacing the selected elements with a vector of the same size. We can even delete elements from a vector by assigning the empty matrix "`[]`" to the selected elements.

```
y([2 4 5]) = [42 420 4200] % y = [-6 42 -3 420 4200]
v(x) = [0 -1 -2] % v = [3 -2 -1 0 7 8 9 10]
w([3 4]) = [] % w = [1 3 9]
```

Mathematical vector operations We can easily add ("`+`"), subtract ("`-`"), multiply ("`*`"), divide ("`/`"), or exponentiate ("`.^`") each element in a vector by a scalar. The operation simply gets performed on each element of the vector, returning a vector of the same size.

```
a = [8 6 1 0]
a/2 - 3 % [1 0 -2.5 -3]
3*a.^2 + 5 % [197 113 8 5]
```

Similarly, we can perform "element-wise" mathematical operations between two vectors of the same size. The operation is simply performed between elements in corresponding positions in the two vectors, again returning a vector of the same size. We use "`+`" for adding two vectors, and "`-`" to subtract two vectors. To avoid conflicts with different types of vector multiplication and division, we use "`.*`" and "`./`" for element-wise multiplication and division, respectively. We use "`.^`" for element-wise exponentiation.

```
b = [4 3 2 9]
a+b % [12 9 3 9]
a-b % [4 3 -1 -9]
a.*b % [32 18 2 0]
a./b % [2 2 0.5 0]
a.^b % [4096 216 1 0]
```

Finally, we can perform a dot product (or *inner product*) between a *row vector* and a *column vector* of the same length by using ("`*`"). The dot product multiplies the elements in corresponding positions in the two vectors, and then takes the sum, returning a scalar value. To perform a dot product, the row vector must be listed before the column vector (otherwise MATLAB will perform an *outer product*, returning a matrix).

```
r = [9 4 0]
c = [8; 7; 5]
r*c % 100
```

1.5.3 Logical operations

Relational operators We can compare two scalar values in MATLAB using relational operators: "`==`" ("equal to"), "`~=`" ("not equal to"), "`<`" ("less than"), "`<=`" ("less than or equal to") "`>`" ("greater than"), and "`>=`" ("greater than or equal to"). The result is 1 if the comparison is true, and 0 if the comparison is false.

```
1 == 2 % 0
1 ~= 2 % 1
2 < 2 % 0
2 <= 3 % 1
(2*2) > 3 % 1
3 >= (5+1) % 0
3/2 == 1.5 % 1
```

Note that floating-point comparisons work correctly in MATLAB.
The unary operator "`~`" ("not") flips a binary value from 1 to 0 or 0 to 1.

```
flag = (4 < 2) % flag = 0
~flag % 1
```

Logical operations with vectors. As with mathematical operations, using a relational operator between a vector and a scalar will compare each each element of the vector with the scalar, in this case returning a binary vector of the same size. Each element of the binary vector is 1 if the comparison is true at that position, and 0 if the comparison is false at that position.

```
ages = [56 47 8 12 20 18 21]
ages >= 21 % [1 1 0 0 0 0 1]
```

To test whether a binary vector contains *any* 1s, we use "`any()`". To test whether a binary vector contains *all* 1s, we use "`all()`".

```
any(ages >= 21) % 1
all(ages >= 21) % 0
any(ages == 3) % 0
all(ages < 100) % 1
```

We can use the binary vectors as a different kind of "index vector" to select elements from a vector; this is called "logical indexing", and it returns all of the elements in the vector where the corresponding element in the binary vector is 1. This gives us a powerful way to select all elements from a vector that meet certain criteria.

```
ages([1 0 1 0 1 0 1]) % [56 8 20 21]
ages(ages >= 21) % [56 47 21]
```

1.5.4 Control Flow

Normally, the MATLAB interpreter moves through a script linearly, executing each statement in sequential order. However, we can use several structures to introduce branching and looping into the flow of our programs.

If statements. An if statement consists of: one `if` block, zero or more `elseif` blocks, and zero or one `else` block. It ends with the keyword `end`.

Any of the relational operators defined above can be used as a condition for an if statement. MATLAB executes the statements in an `if` block or a `elseif` block only if its associated condition is true. Otherwise, the MATLAB interpreter skips that block. If none of the conditions were true, MATLAB executes the statements in the `else` block (if there is one).

```
team1_score = rand() % a random number between 0 and 1
team2_score = rand() % a random number between 0 and 1

if(team1_score > team2_score)
    disp('Team 1 wins!') % Display "Team 1 wins!"
elseif(team1_score == team2_score)
    disp('It's a tie!') % Display "It's a tie!"
else
    disp('Team 2 wins!') % Display "Team 2 wins!"
end
```

In fact, instead of using a relational operator as a condition, we can use any expression. If the expression evaluates to anything other than 0, the empty matrix `[]`, or the boolean value `false`, then the expression is considered to be "true".

While loops. A while loop works the same way as an if statement, except that, when the MATLAB interpreter reaches the `end` keyword, it returns to the beginning of the while block and tests the condition again. MATLAB executes the statements in the while block repeatedly, as long as the condition is true. A `break` statement within the while loop will cause MATLAB to skip the rest of the loop.

```
i = 3
while i > 0
    disp(i)
    i = i - 1;
end
disp('Blastoff!')

% This will display:
% 3
% 2
% 1
% Blastoff!
```

For loops. To execute a block of code a specific number of times, we can use a for loop. A for loop takes a counter variable and a vector. MATLAB executes the statements in the block once for each element in the vector, with the counter variable set to that element.

```
r = [9 4 0];
c = [8 7 5];

sum = 0;
for i = 1:3 % The counter is 'i', and the range is '1:3'
    sum = sum + r(i) * c(i); % This will be executed 3 times
end

% After the loop, sum = 100
```

Although the "range" vector is most commonly a range of consecutive integers, it doesn't have to be. Actually, the range vector doesn't even need to be created with the colon operator. In fact, the range vector can be any vector whatsoever; it doesn't even need to contain integers at all!

```
my_favorite_primes = [2 3 5 7 11]
for order = [2 4 3 1 5]
    disp(my_favorite_primes(order))
end

% This will display:
% 3
% 7
% 5
% 2
% 11
```

1.5.5 Vectorized Code

Vectorized code is code that describes (and, conceptually) executes mathematical operations on vectors an matrices "all at once". Vectorised code is truer to the parallel "spirit" of the operations being performed in linear algebra, and also to the conceptual framework of PDP. Conceptually, the pseudocode descriptions of our algorithms (usually) should not involve the sequential repetition of a for loop. For example, when computing the input to a unit from other units, there is no reason for the multiplication of one activation times one connection weight to "wait" for the previous one to be completed. Instead, each multiplication should be though of as being performed independently and simultaneously. And in fact, vectorized code can execute much faster that code written explicitly as a for loop. This effect is especially pronounced when processing can be split across several processors.

Writing vectorised code. Let's say we have two vectors, `r` and `c`.

```
r = [9 4 0];
c = [8;7;5];
```

We have seen two ways to perform a dot product between these two vectors. We can use a for loop:

```
sum = 0;
for i = 1:3
    sum = sum + r(i) * c(i);
end
% After the loop, sum = 100
```

However, the following "vectorized" code is more concise, and it takes advantage of MATLAB's optimization for vector and matrix operations:

```
sum = r*c; % After this statement, sum = 100
```

Similarly, we can use a for loop to multiply each element of a vector by a scalar, or to multiply each element of a vector by the corresponding element in another vector:

```
for i = 1:3
    r(i) = r(i) * 2;
end
% After the loop, r = [18 8 0]

multiplier = [2;3;4];
for j = 1:3
    c(j) = c(j) * multiplier(j);
end
% After the loop, c = [16 21 20]
```

However, element-wise multiplication using `.*` is faster and more concise:

```
r * 2; % After this statement, r = [18 8 0]

multiplier = [2;3;4];
c = c .* multiplier; % After this statement, c = [16 21 20]
```

Chapter 2

Interactive Activation and Competition

Our own explorations of parallel distributed processing began with the use of interactive activation and competition mechanisms of the kind we will examine in this chapter. We have used these kinds of mechanisms to model visual word recognition (McClelland and Rumelhart, 1981; Rumelhart and McClelland, 1982) and to model the retrieval of general and specific information from stored knowledge of individual exemplars (McClelland, 1981), as described in *PDP:1*. In this chapter, we describe some of the basic mathematical observations behind these mechanisms, and then we introduce the reader to a specific model that implements the retrieval of general and specific information using the "Jets and Sharks" example discussed in *PDP:1* (pp. 25-31).

After describing the specific model, we will introduce the program in which this model is implemented: the **iac** program (for interactive activation and competition). The description of how to use this program will be quite extensive; it is intended to serve as a general introduction to the entire package of programs since the user interface and most of the commands and auxiliary files are common to all of the programs. After describing how to use the program, we will present several exercises, including an opportunity to work with the Jets and Sharks example and an opportunity to explore an interesting variant of the basic model, based on dynamical assumptions used by Grossberg (e.g., (Grossberg, 1978)).

2.1 BACKGROUND

The study of interactive activation and competition mechanisms has a long history. They have been extensively studied by Grossberg. A useful introduction to the mathematics of such systems is provided in Grossberg (1978). Related mechanisms have been studied by a number of other investigators, including Levin (1976), whose work was instrumental in launching our exploration of

PDP mechanisms.

An interactive activation and competition network (hereafter, *IAC network*) consists of a collection of processing units organized into some number of competitive pools. There are excitatory connections among units in different pools and inhibitory connections among units within the same pool. The excitatory connections between pools are generally bidirectional, thereby making the processing *interactive* in the sense that processing in each pool both influences and is influenced by processing in other pools. Within a pool, the inhibitory connections are usually assumed to run from each unit in the pool to every other unit in the pool. This implements a kind of competition among the units such that the unit or units in the pool that receive the strongest activation tend to drive down the activation of the other units.

The units in an IAC network take on continuous activation values between a maximum and minimum value, though their output—the signal that they transmit to other units—is not necessarily identical to their activation. In our work, we have tended to set the output of each unit to the activation of the unit minus the *threshold* as long as the difference is positive; when the activation falls below threshold, the output is set to 0. Without loss of generality, we can set the threshold to 0; we will follow this practice throughout the rest of this chapter. A number of other output functions are possible; Grossberg (1978) describes a number of other possibilities and considers their various merits.

The activations of the units in an IAC network evolve gradually over time. In the mathematical idealization of this class of models, we think of the activation process as completely continuous, though in the simulation modeling we approximate this ideal by breaking time up into a sequence of discrete steps. Units in an IAC network change their activation based on a function that takes into account both the current activation of the unit and the net input to the unit from other units or from outside the network. The net input to a particular unit (say, unit i) is the same in almost all the models described in this volume: it is simply the sum of the influences of all of the other units in the network plus any external input from outside the network. The influence of some other unit (say, unit j) is just the product of that unit's output, $output_j$, times the strength or weight of the connection to unit i from unit j. Thus the net input to unit i is given by

$$net_i = \sum_j w_{ij} output_j + extinput_i. \tag{2.1}$$

In the IAC model, $output_j = [a_j]^+$. Here, a_j refers to the activation of unit j, and the expression $[a_j]^+$ has value a_j for all $a_j > 0$; otherwise its value is 0. The index j ranges over all of the units with connections to unit i. In general the weights can be positive or negative, for excitatory or inhibitory connections, respectively.

Human behavior is highly variable and IAC models as described thus far are completely deterministic. In some IAC models, such as the interactive activation model of letter perception (McClelland and Rumelhart, 1981) these deterministic activation values are mapped to probabilities. However, it became clear in

detailed attempts to fit this model to data that intrinsic variability in processing and/or variability in the input to a network from trial to trial provided better mechanisms for allowing the models to provide detailed fits to data. McClelland (1991) found that injecting normally distributed random noise into the net input to each unit on each time cycle allowed such networks to fit experimental data from experiments on the joint effects of context and stimulus information on phoneme or letter perception. Including this in the equation above, we have:

$$net_i = \sum_j w_{ij} output_j + extinput_i + normal(0, noise) \tag{2.2}$$

Where $normal(0, noise)$ is a sample chosen from the standard normal distribution with mean 0 and standard deviation of *noise*. For simplicity, *noise* is set to zero in many IAC network models.

Once the net input to a unit has been computed, the resulting change in the activation of the unit is as follows:

If $(net_i > 0)$,

$$\Delta a_i = (max - a_i) net_i - decay(a_i - rest).$$

Otherwise,

$$\Delta a_i = (a_i - min) net_i - decay(a_i - rest).$$

Note that in this equation, *max*, *min*, *rest*, and *decay* are all parameters. In general, we choose $max = 1$, $min \leq rest \leq 0$, and *decay* between 0 and 1. Note also that a_i is assumed to start, and to stay, within the interval $[min, max]$.

Suppose we imagine the input to a unit remains fixed and examine what will happen across time in the equation for Δa_i. For specificity, let's just suppose the net input has some fixed, positive value. Then we can see that Δa_i will get smaller and smaller as the activation of the unit gets greater and greater. For some values of the unit's activation, Δa_i will actually be negative. In particular, suppose that the unit's activation is equal to the resting level. Then Δa_i is simply $(max - rest) net_i$. Now suppose that the unit's activation is equal to *max*, its maximum activation level. Then Δa_i is simply $(-decay)(max - rest)$. Between these extremes there is an equilibrium value of a_i at which Δa_i is 0. We can find what the equilibrium value is by setting Δa_i to 0 and solving for a_i:

$$\begin{aligned} 0 &= (max - a_i) net_i - decay(a_i - rest) \\ &= (max)(net_i) + (rest)(decay) - a_i(net_i + decay) \\ a_i &= \frac{(max)(net_i) + (rest)(decay)}{net_i + decay} \end{aligned} \tag{2.3}$$

Using $max = 1$ and $rest = 0$, this simplifies to

$$a_i = \frac{net_i}{net_i + decay} \tag{2.4}$$

What the equation indicates, then, is that the activation of the unit will reach equilibrium when its value becomes equal to the ratio of the net input divided by

the net input plus the decay. Note that in a system where the activations of other units—and thus of the net input to any particular unit—are also continually changing, there is no guarantee that activations will ever completely stabilize—although in practice, as we shall see, they often seem to.

Equation 3 indicates that the equilibrium activation of a unit will always increase as the net input increases; however, it can never exceed 1 (or, in the general case, max) as the net input grows very large. Thus, max is indeed the upper bound on the activation of the unit. For small values of the net input, the equation is approximately linear since $x/(x + c)$ is approximately equal to x/c for x small enough.

We can see the decay term in Equation 3 as acting as a kind of restoring force that tends to bring the activation of the unit back to 0 (or to $rest$, in the general case). The larger the value of the decay term, the stronger this force is, and therefore the lower the activation level will be at which the activation of the unit will reach equilibrium. Indeed, we can see the decay term as scaling the net input if we rewrite the equation as

$$a_i = \frac{net_i/decay}{(net_i/decay) + 1} \tag{2.5}$$

When the net input is equal to the decay, the activation of the unit is 0.5 (in the general case, the value is $(max + rest)/2$). Because of this, we generally scale the net inputs to the units by a strength constant that is equal to the decay. Increasing the value of this strength parameter or decreasing the value of the decay increases the equilibrium activation of the unit.

In the case where the net input is negative, we get entirely analogous results:

$$a_i = \frac{(min)(net_i) - (decay)(rest)}{net_i - decay} \tag{2.6}$$

Using $rest = 0$, this simplifies to

$$a_i = \frac{(min)(net_i)}{net_i - decay} \tag{2.7}$$

This equation is a bit confusing because net_i and min are both negative quantities. It becomes somewhat clearer if we use $amin$ (the absolute value of min) and $anet_i$ (the absolute value of net_i). Then we have

$$a_i = -\frac{(amin)(anet_i)}{anet_i + decay} \tag{2.8}$$

What this last equation brings out is that the equilibrium activation value obtained for a negative net input is scaled by the magnitude of the minimum ($amin$). Inhibition both acts more quickly and drives activation to a lower final level when min is farther below 0.

2.1.1 How Competition Works

So far we have been considering situations in which the net input to a unit is fixed and activation evolves to a fixed or stable point. The interactive activation and competition process, however, is more complicated than this because the net input to a unit changes as the unit and other units in the same pool simultaneously respond to their net inputs. One effect of this is to amplify differences in the net inputs of units. Consider two units a and b that are mutually inhibitory, and imagine that both are receiving some excitatory input from outside but that the excitatory input to a (e_a) is stronger than the excitatory input to b (e_b). Let γ represent the strength of the inhibition each unit exerts on the other. Then the net input to a is

$$net_a = e_a - \gamma(output_b) \tag{2.9}$$

and the net input to b is

$$net_b = e_b - \gamma(output_a) \tag{2.10}$$

As long as the activations stay positive, $output_i = a_i$, so we get

$$net_a = e_a - \gamma a_b \tag{2.11}$$

and

$$net_b = e_b - \gamma a_a \tag{2.12}$$

From these equations we can easily see that b will tend to be at a disadvantage since the stronger excitation to a will tend to give a a larger initial activation, thereby allowing it to inhibit b more than b inhibits a. The end result is a phenomenon that Grossberg (1976) has called "the rich get richer" effect: Units with slight initial advantages, in terms of their external inputs, amplify this advantage over their competitors.

2.1.2 Resonance

Another effect of the interactive activation process has been called "resonance" by Grossberg (1978). If unit a and unit b have mutually excitatory connections, then once one of the units becomes active, they will tend to keep each other active. Activations of units that enter into such mutually excitatory interactions are therefore sustained by the network, or "resonate" within it, just as certain frequencies resonate in a sound chamber. In a network model, depending on parameters, the resonance can sometimes be strong enough to overcome the effects of decay. For example, suppose that two units, a and b, have bidirectional, excitatory connections with strengths of 2 x *decay* . Suppose that we set each unit's activation at 0.5 and then remove all external input and see what happens. The activations will stay at 0.5 indefinitely because

$$\Delta a_a = (1 - a_a)net_a - (decay)a_a$$

$$= (1 - 0.5)(2)(decay)(0.5) - (decay)(0.5)$$
$$= (0.5)(2)(decay)(0.5) - (decay)(0.5)$$
$$= 0$$

Thus, IAC networks can use the mutually excitatory connections between units in different pools to sustain certain input patterns that would otherwise decay away rapidly in the absence of continuing input. The interactive activation process can also activate units that were not activated directly by external input. We will explore these effects more fully in the exercises that are given later.

2.1.3 Hysteresis and Blocking

Before we finish this consideration of the mathematical background of interactive activation and competition systems, it is worth pointing out that the rate of evolution towards the eventual equilibrium reached by an IAC network, and even the state that is reached, is affected by initial conditions. Thus if at time 0 we force a particular unit to be on, this can have the effect of slowing the activation of other units. In extreme cases, forcing a unit to be on can totally block others from becoming activated at all. For example, suppose we have two units, a and b, that are mutually inhibitory, with inhibition parameter *gamma* equal to 2 times the strength of the decay, and suppose we set the activation of one of these units—unit a—to 0.5. Then the net input to the other—unit b—at this point will be (-0.5) (2) $(decay) = -decay$. If we then supply external excitatory input to the two units with strength equal to the decay, this will maintain the activation of unit a at 0.5 and will fail to excite b since its net input will be 0. The external input to b is thereby blocked from having its normal effect. If external input is withdrawn from a, its activation will gradually decay (in the absence of any strong resonances involving a) so that b will gradually become activated. The first effect, in which the activation of b is completely blocked, is an extreme form of a kind of network behavior known as hysteresis (which means "delay"); prior states of networks tend to put them into states that can delay or even block the effects of new inputs.

Because of hysteresis effects in networks, various investigators have suggested that new inputs may need to begin by generating a "clear signal," often implemented as a wave of inhibition. Such ideas have been proposed by various investigators as an explanation of visual masking effects (see, e.g., (Weisstein et al., 1975)) and play a prominent role in Grossberg's theory of learning in neural networks, see Grossberg (1980).

2.1.4 Grossberg's Analysis of Interactive Activation and Competition Processes

Throughout this section we have been referring to Grossberg's studies of what we are calling interactive activation and competition mechanisms. In fact, he

uses a slightly different activation equation than the one we have presented here (taken from our earlier work with the interactive activation model of word recognition). In Grossberg's formulation, the excitatory and inhibitory inputs to a unit are treated separately. The excitatory input (e) drives the activation of the unit up toward the maximum, whereas the inhibitory input (i) drives the activation back down toward the minimum. As in our formulation, the decay tends to restore the activation of the unit to its resting level.

$$\Delta a = (max - a)e - (a - min)i - decay(a - rest) \tag{2.13}$$

Grossberg's formulation has the advantage of allowing a single equation to govern the evolution of processing instead of requiring an *if* statement to intervene to determine which of two equations holds. It also has the characteristic that the direction the input tends to drive the activation of the unit is affected by the current activation. In our formulation, net positive input will always excite the unit and net negative input will always inhibit it. In Grossberg's formulation, the input is not lumped together in this way. As a result, the effect of a given input (particular values of e and i) can be excitatory when the unit's activation is low and inhibitory when the unit's activation is high. Furthermore, at least when min has a relatively small absolute value compared to max, a given amount of inhibition will tend to exert a weaker effect on a unit starting at rest. To see this, we will simplify and set $max = 1.0$ and $rest = 0.0$. By assumption, the unit is at rest so the above equation reduces to

$$\Delta a = (1)e - (amin)(i) \tag{2.14}$$

where $amin$ is the absolute value of min as above. This is in balance only if $i = e/amin$.

Our use of the net input rule was based primarily on the fact that we found it easier to follow the course of simulation events when the balance of excitatory and inhibitory influences was independent of the activation of the receiving unit. However, this by no means indicates that our formulation is superior computationally. Therefore we have made Grossberg's update rule available as an option in the **iac** program. Note that in the Grossberg version, noise is added into the excitatory input, when the *noise* standard deviation parameter is greater than 0.

2.2 THE IAC MODEL

The IAC model provides a discrete approximation to the continuous interactive activation and competition processes that we have been considering up to now. We will consider two variants of the model: one that follows the interactive activation dynamics from our earlier work and one that follows the formulation offered by Grossberg.

The IAC model is part of the part of the PDPTool Suite of programs, which run under MATLAB. A document describing the overall structure of the PDP-

tool called the *PDPTool User Guide* should be consulted to get a general understanding of the structure of the PDPtool system.

Here we describe key characteristics of the IAC model software implementation. Specifics on how to run exercises using the IAC model are provided as the exercises are introduced below.

2.2.1 Architecture

The IAC model consists of several units, divided into *pools*. In each pool, all the units are assumed to be mutually inhibitory. Between pools, units may have excitatory connections. The model assumes that these connections are bidirectional, so that whenever there is an excitatory connection from unit i to unit j, there is also an excitatory connection from unit j back to unit i. These constraints are enforced in the connection matrices used in models within the iac framework, not in the program code itself.

2.2.2 Visible and Hidden Units

In an IAC network, there are generally two classes of units: those that can receive direct input from outside the network and those that cannot. The first kind of units are called *visible* units; the latter are called *hidden* units. Thus in the IAC model the user may specify a pattern of inputs to the visible units, but by assumption the user is not allowed to specify external input to the hidden units; their net input is based only on the outputs from other units to which they are connected.

2.2.3 Activation Dynamics

Time is not continuous in the IAC model (or any of our other simulation models), but is divided into a sequence of discrete steps, or *cycles*. Each cycle begins with all units having an activation value that was determined at the end of the preceding cycle. First, the inputs to each unit are computed. Then the activations of the units are updated. The two-phase procedure ensures that the updating of the activations of the units is effectively synchronous; that is, nothing is done with the new activation of any of the units until all have been updated.

The discrete time approximation can introduce instabilities if activation steps on each cycle are large. This problem is eliminated, and the approximation to the continuous case is generally closer, when activation steps are kept small on each cycle.

2.2.4 Parameters

In the IAC model there are several parameters under the user's control. Most of these have already been introduced. They are

max The maximum activation parameter.

min The minimum activation parameter.

rest The resting activation level to which activations tend to settle in the absence of external input.

decay The decay rate parameter, which determines the strength of the tendency to return to resting level.

estr This parameter stands for the strength of external input (i.e., input to units from outside the network). It scales the influence of external signals relative to internally generated inputs to units.

alpha This parameter scales the strength of the excitatory input to units from other units in the network.

gamma This parameter scales the strength of the inhibitory input to units from other units in the network.

In general, it would be possible to specify separate values for each of these parameters for each unit. The IAC model does not allow this, as we have found it tends to introduce far too many degrees of freedom into the modeling process. However, the model does allow the user to specify strengths for the individual connection strengths in the network.

The *noise* parameter is treated separately in the IAC model. Here, there is a pool-specific variable called 'noise'. How this actually works is described under Core Routines below.

2.2.5 Pools and Projections

The main thing to understand about the way networks work is to understand the concepts *pool* and *projection*. A *pool* is a set of units and a *projection* is a set of connections linking two pools. A network could have a single pool and a single projection, but usually networks have more constrained architectures than this, so that a pool and projection structure is appropriate.

All networks have a special pool called the bias pool that contains a single unit called the bias unit that is always on. The connection weights from the bias pool to the units in another pool can take any value, and that value then becomes a constant part of the input to the unit. The bias pool is always pool(1). A network with a layer of input units and a layer of hidden units would have two additional pools, pool(2) and pool(3) respectively.

Projections are attached to units receiving connections from another pool. The first projection to each pool is the projection from the bias pool, if such a projection is used (there is no such projection in the *jets* network). A projection can be from a pool to itself, or from a pool to another pool. In the *jets* network, there is pool for the visible units and a pool for the hidden units, and there is a self-projection (projection 1 in both cases) and also a projection from the other pool (projection 2 in each case).

The connection to hidden unit i from visible unit j is:

$$net.pool(2).projection(2).weight(i, j)$$

2.2.6 The Core Routines

Here we explain the basic structure of the core routines used in the **iac** program.

reset. This routine is used to reset the activations of units to their resting levels and to reset the time—the current cycle number—back to 0. All variables are cleared, and the display is updated to show the network before processing begins.

cycle. This routine is the basic routine that is used in running the model. It carries out a number of processing cycles, as determined by the program control variable *ncycles*. On each cycle, two routines are called: *getnet* and *update*. At the end of each cycle, if pdptool is being run in gui mode, then the program checks to see whether the display is to be updated and whether to pause so the user can examine the new state (and possibly terminate processing). The routine looks like this:

```
function cycle

for cy = 1: ncycles
     cycleno = cycleno + 1;
    getnet();
    update();
% what follows is concerned with
% pausing and updating the display
   if guimode && display_granularity == cycle
      update_display();
   end
end
```

The *getnet* and *update* routines are somewhat different for the standard version and Grossberg version of the program. We first describe the standard versions of each, then turn to the Grossberg versions.

Standard getnet. The standard *getnet* routine computes the net input for each pool. The net input consists of three things: the external input, scaled by *estr*; the excitatory input from other units, scaled by *alpha*; and the inhibitory input from other units, scaled by *gamma*. For each pool, the *getnet* routine first accumulates the excitatory and inhibitory inputs from other units, then scales the inputs and adds them to the scaled external input to obtain the net input. If the pool-specific noise parameter is non-zero, a sample from the standard normal distribution is taken, then multiplied by the value of the 'noise' parameter, then added to the excitatory input.

Whether a connection is excitatory or inhibitory is determined by its sign. The connection weights from every sending unit to a pool(wts) are examined. For all positive values of wts, the corresponding excitation terms are incremented by $pool(sender).activation(index) * wts(wts > 0)$. This operation uses matlab logical indexing to apply the computation to only those elements of the array that satisfy the condition. Similarly, for all negative values of wts, $pool(sender).activation(index) * wts(wts < 0)$ is added into the inhibition terms. These operations are only performed for sending units that have positive activations. The code that implements these calculations is as follows:

```
function getnet

for i=1:numpools
   pool(i).excitation = 0.0;
   pool(i).inhibition = 0.0;
   for sender = 1:numprojections_into_pool(i)
      positive_acts_indices =  find(pool(sender).activation > 0);
      if ~isempty(positive_acts_indices)
         for k = 1:numelements(positive_acts_indices)
             index = positive_acts_indices(k);
             wts = projection_weight(:,index);
             pool(i).excitation (wts>0) = pool(i).excitation(wts>0)
                        + pool(sender).activation(index) * wts(wts>0);
             pool(i).inhibition (wts<0) = pool(i).inhibition(wts<0)
                        + pool(sender).activation(index) * wts(wts<0);
          end
       end
       pool(i).excitation = pool(i).excitation * alpha;
       pool(i).inhibition = pool(i).inhibition * gamma;
       if (pool(i).noise)
           pool(i).excitation = pool(i).excitation +
             Random('Normal',0,pool(i).noise,size(pool(1).excitation);
       end
       pool(i).netinput = pool(i).excitation + pool(i).inhibition
                          + estr * pool(i).extinput;
end
```

Standard update. The *update* routine increments the activation of each unit, based on the net input and the existing activation value. The vector *pns* is a logical array (of 1s and 0s), 1s representing those units that have positive netinput and 0s for the rest. This is then used to index into the activation and netinput vectors and compute the new activation values. Here is what it looks like:

```
function update
for i = 1:numpools
```

```
    pns = find(pool(i).netinput > 0);
    if ~isempty(pns)
       pool(i).activation(pns) = pool(i).activation(pns)
            + (max- pool(i).activation(pns))*pool(i).netinput(pns)
            - decay*(pool(i).activation(pns) - rest);
    end
    nps = ~pns;
    if ~isempty(nps)
       pool(i).activation(nps) = pool(i).activation(nps)
            + (pool(i).activation(nps) -min))*pool(i).netinput(nps)
            - decay*(pool(i).activation(nps) - rest);
    end
    pool(i).activation(pool(i).activation > max) = max;
    pool(i).activation(pool(i).activation < min) = min;
end
```

The last two conditional statements are included to guard against the anomalous behavior that would result if the user had set the *estr*, *istr*, and *decay* parameters to values that allow activations to change so rapidly that the approximation to continuity is seriously violated and activations have a chance to escape the bounds set by the values of *max* and *min*.

Grossberg versions. The Grossberg versions of these two routines are structured like the standard versions. In the *getnet* routine, the only difference is that the net input for each pool is not computed; instead, the excitation and inhibition are scaled by *alpha* and *gamma*, respectively, and scaled external input is added to the excitation if it is positive or is added to the inhibition if it is negative:

```
   pool(i).excitation = pool(i).excitation * alpha;
   pool(i).inhibition = pool(i).inhibition * gamma;
   posext = find(net.pool(i).extinput > 0);
   negext = find(net.pool(i).extinput < 0);
   pool(i).excitation(posext) = pool(i).excitation(posext)
              +  estr * pool(i).extinput(posext);
   pool(i).inhibition(negext) = pool(i). inhibition (negext)
              +  estr * pool(i).extinput(negext);
```

In the *update* routine the two different versions of the standard activation rule are replaced by a single expression. The routine then becomes

```
function update
pool(i).activation = pool(i).activation
         + (max - pool(i).activation) .* pool(i).excitation
         + (pool(i).activation - min) .* pool(i).inhibition
         - decay * (pool(i).activation - rest);
```

```
pool(i).activation(pool(i).activation > max) = max;
pool(i).activation(pool(i).activation < min) = min;
```

The program makes no explicit reference to the IAC network architecture, in which the units are organized into competitive pools of mutually inhibitory units and in which excitatory connections are assumed to be bidirectional. These architectural constraints are imposed in the network file. In fact, the **iac** program can implement any of a large variety of network architectures, including many that violate the architectural assumptions of the IAC framework. As these examples illustrate, the core routines of this model—indeed, of all of our models—are extremely simple.

2.3 EXERCISES

In this section we suggest several different exercises. Each will stretch your understanding of IAC networks in a different way. Ex. 2.1 focuses primarily on basic properties of IAC networks and their application to various problems in memory retrieval and reconstruction. Ex. 2.2 suggests experiments you can do to examine the effects of various parameter manipulations. Ex. 2.3 fosters the exploration of Grossberg's update rule as an alternative to the default update rule used in the **iac** program. Ex. 2.4 suggests that you develop your own task and network to use with the **iac** program.

If you want to cement a basic understanding of IAC networks, you should probably do several parts of Ex. 2.1 , as well as Ex. 2.2 The first few parts of Ex. 2.1 also provide an easy tutorial example of the general use of the programs in this book.

Ex2.1. Retrieval and Generalization

Use the **iac** program to examine how the mechanisms of interactive activation and competition can be used to illustrate the following properties of human memory:

Retrieval by name and by content.

Assignment of plausible default values when stored information is incomplete.

Spontaneous generalization over a set of familiar items.

The "data base" for this exercise is the Jets and Sharks data base shown in Figure 10 of *PDP:1* and reprinted here for convenience in Figure 2.1. You are to use the **iac** program in conjunction with this data base to run illustrative simulations of these basic properties of memory. In so doing, you will observe behaviors of the network that you will have to explain using the analysis of IAC networks presented earlier in the "Background section".

The Jets and The Sharks

Name	Gang	Age	Edu	Mar	Occupation
Art	Jets	40's	J.H.	Sing.	Pusher
Al	Jets	30's	J.H.	Mar.	Burglar
Sam	Jets	20's	COL.	Sing.	Bookie
Clyde	Jets	40's	J.H.	Sing.	Bookie
Mike	Jets	30's	J.H.	Sing.	Bookie
Jim	Jets	20's	J.H.	Div.	Burglar
Greg	Jets	20's	H.S.	Mar.	Pusher
John	Jets	20's	J.H.	Mar.	Burglar
Doug	Jets	30's	H.S.	Sing.	Bookie
Lance	Jets	20's	J.H.	Mar.	Burglar
George	Jets	20's	J.H.	Div.	Burglar
Pete	Jets	20's	H.S.	Sing.	Bookie
Fred	Jets	20's	H.S.	Sing.	Pusher
Gene	Jets	20's	COL.	Sing.	Pusher
Ralph	Jets	30's	J.H.	Sing.	Pusher
Phil	Sharks	30's	COL.	Mar.	Pusher
Ike	Sharks	30's	J.H.	Sing.	Bookie
Nick	Sharks	30's	H.S.	Sing.	Pusher
Don	Sharks	30's	COL.	Mar.	Burglar
Ned	Sharks	30's	COL.	Mar.	Bookie
Karl	Sharks	40's	H.S.	Mar.	Bookie
Ken	Sharks	20's	H.S.	Sing.	Burglar
Earl	Sharks	40's	H.S.	Mar.	Burglar
Rick	Sharks	30's	H.S.	Div.	Burglar
Ol	Sharks	30's	COL.	Mar.	Pusher
Neal	Sharks	30's	H.S.	Sing.	Bookie
Dave	Sharks	30's	H.S.	Div.	Pusher

Figure 2.1: Characteristics of a number of individuals belonging to two gangs, the Jets and the Sharks. (From "Retrieving General and Specific Knowledge From Stored Knowledge of Specifics" by 1. L. McClelland, 1981, *Proceedings of the Third Annual Conference of the Cognitive Science Society.* Copyright 1981 by J. L. McClelland. Reprinted by permission.)

Starting up. In MATLAB, make sure your path is set to your pdptool folder, and set your current directory to be the iac folder. Enter 'jets' at the MATLAB command prompt. Every label on the display you see corresponds to a unit in the network. Each unit is represented as two squares in this display. The square to the left of the label indicates the external input for that unit (initially, all inputs are 0). The square to the right of the label indicates the activation of that unit (initially, all activation values are equal to the value of the *rest* parameter, which is -0.1).

If the colorbar is not on, click the 'colorbar' menu at the top left of the display. Select 'on'. To select the correct 'colorbar' for the jets and sharks exercise, click the colorbar menu item again, click 'load colormap' and then select the jmap colormap file in the iac directory. With this colormap, an activation of 0 looks gray, -.2 looks blue, and 1.0 looks red. Note that when you

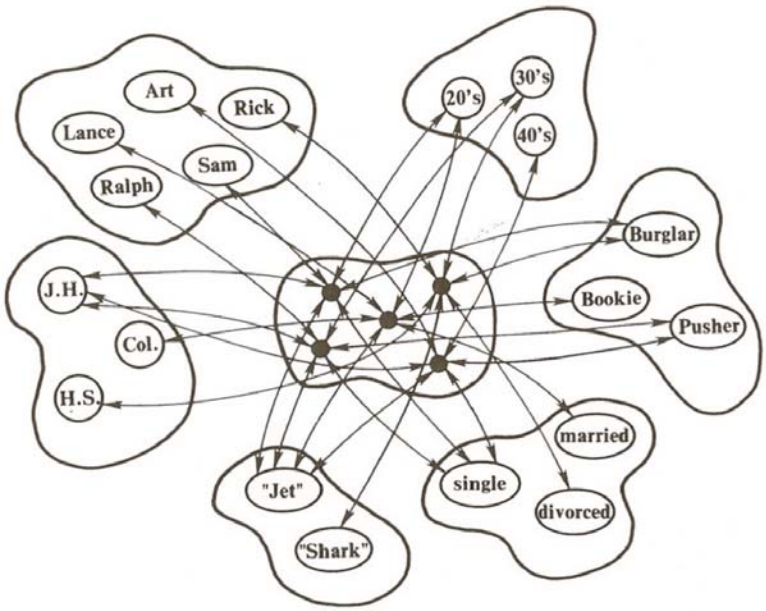

Figure 2.2: The units and connections for some of the individuals in Figure 2.1. (Two slight errors in the connections depicted in the original of this figure have been corrected in this version.) (From "Retrieving General and Specific Knowledge From Stored Knowledge of Specifics" by J. L. McClelland, 1981, *Proceedings of the Third Annual Conference of the Cognitive Science Society.* Copyright 1981 by J. L. McClelland. Reprinted by permission.)

hold the mouse over a colored tile, you will see the numeric value indicated by the color (and you get the name of the unit, as well). Try right-clicking on the colorbar itself and choosing other mappings from 'Standard Colormaps' to see if you prefer them over the default.

The units are grouped into seven pools: a pool of *name* units, a pool of *gang* units, a pool of *age* units, a pool of *education* units, a pool of *marital status* units, a pool of *occupation* units, and a pool of *instance* units. The *name* pool contains a unit for the name of each person; the *gang* pool contains a unit for each of the gangs the people are members of (Jets and Sharks); the *age* pool contains a unit for each age range; and so on. Finally, the *instance* pool contains a unit for each individual in the set.

The units in the first six pools can be called *visible* units, since all are assumed to be accessible from outside the network. Those in the gang, age, education, marital status, and occupation pools can also be called property units. The instance units are assumed to be inaccessible, so they can be called *hidden* units.

Each unit has an inhibitory connection to every other unit in the same pool. In addition, there are two-way excitatory connections between each instance unit and the units for its properties, as illustrated in Figure 2.2 (Figure 11 from *PDP:1*). Note that the figure is incomplete, in that only some of the name and instance units are shown. These names are given only for the convenience of the user, of course; all actual computation in the network occurs only by way of the connections.

Note: Although conceptually there are six distinct visible pools, and they have been grouped separately on the display, internal to the program they form a single pool, called pool(2). Within pool(2), inhibition occurs only among units within the same conceptual pool. The pool of instance units is a separate pool (pool(3)) inside the network. All units in this pool are mutually inhibitory.

The values of the parameters for the model are:

$max = 1.0$
$min = -0.2$
$rest = -0.1$
$decay = 0.1$
$estr = 0.4$
$alpha = 0.1$
$gamma = 0.1$

The program produces the display shown in Figure 2.3. The display shows the names of all of the units. Unit names are preceded by a two-digit unit number for convenience in some of the exercises below. The visible units are on the left in the display, and the hidden units are on the right. To the right of each visible unit name are two squares. The first square indicates the external input to the unit (which is initially 0). The second one indicates the activation of the unit, which is initially equal to the value of the *rest* parameter.

Since the hidden units do not receive external input, there is only one square to the right of the unit name for these units, for the unit's activation. These units too have an initial activation activation level equal to *rest*.

On the far right of the display is the current cycle number, which is initialized to 0.

Since everything is set up for you, you are now ready to do each of the separate parts of the exercise. Each part is accomplished by using the interactive activation and competition process to do pattern completion, given some probe that is presented to the network. For example, to retrieve an individual's properties from his name, you simply provide external input to his name unit, then allow the IAC network to propagate activation first to the name unit, then from there to the instance units, and from there to the units for the properties of the instance.

Retrieving an individual from his name. To illustrate retrieval of the properties of an individual from his name, we will use Ken as our example. Set the

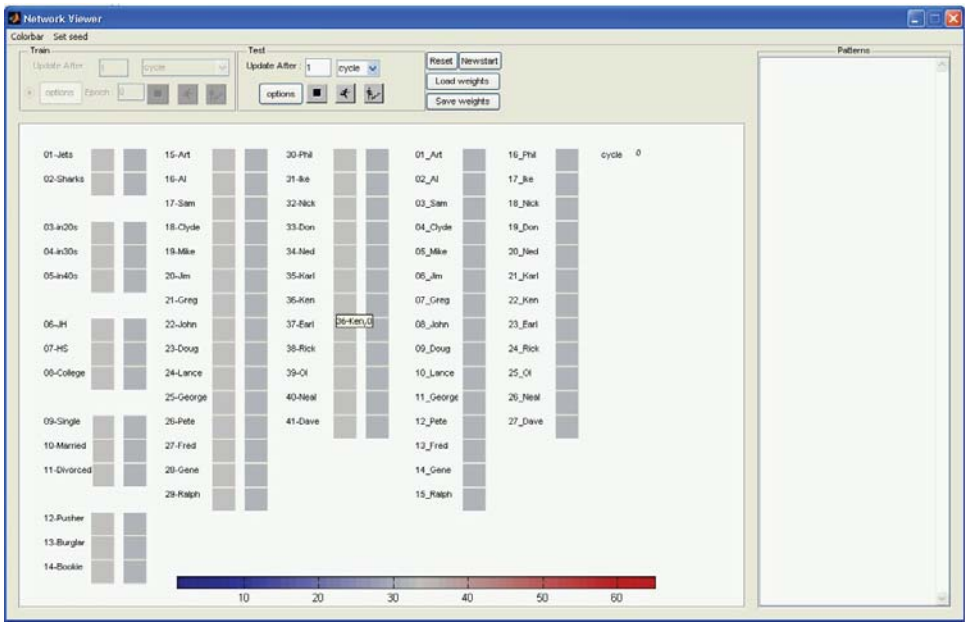

Figure 2.3: The initial display produced by the **iac** program for Ex. 2.1.

external input of Ken's name unit to 1. Right-click on the square to right of the label *36-Ken*. Type 1.00 and click enter. The square should turn red.

To run the network, you need to set the number of cycles you wish the network to run for (default is 10), and then click the button with the running man cartoon. The number of cycles passed is indicated in the top right corner of the network window. Click the run icon once now. Alternatively, you can click on the step icon 10 times, to get to the point where the network has run for 10 cycles.

The PDPtool programs offer a facility for creating graphs of units' activations (or any other variables) as processing occurs. One such graph is set up for you. The panels on the left show the activations of units in each of the different visible pools excluding the name pool. The activations of the name units are shown in the middle. The activations of the instance units are shown in two panels on the right, one for the Jets and one for the Sharks. (If this window is in your way you can minimize (iconify) it, but you should not close it, since it must still exist for its contents to be reset properly when you reset the network.)

What you will see after running 10 cycles is as follows. In the Name panel, you will see one curve that starts at about .35 and rises rapidly to .8. This is the curve for the activation of unit *36-Ken*. Most of the other curves are still at or near rest. (Explain to yourself why some have already gone below rest at this point.) A confusing fact about these graphs is that if lines fall on top of each other you only see the last one plotted, and at this point many of the

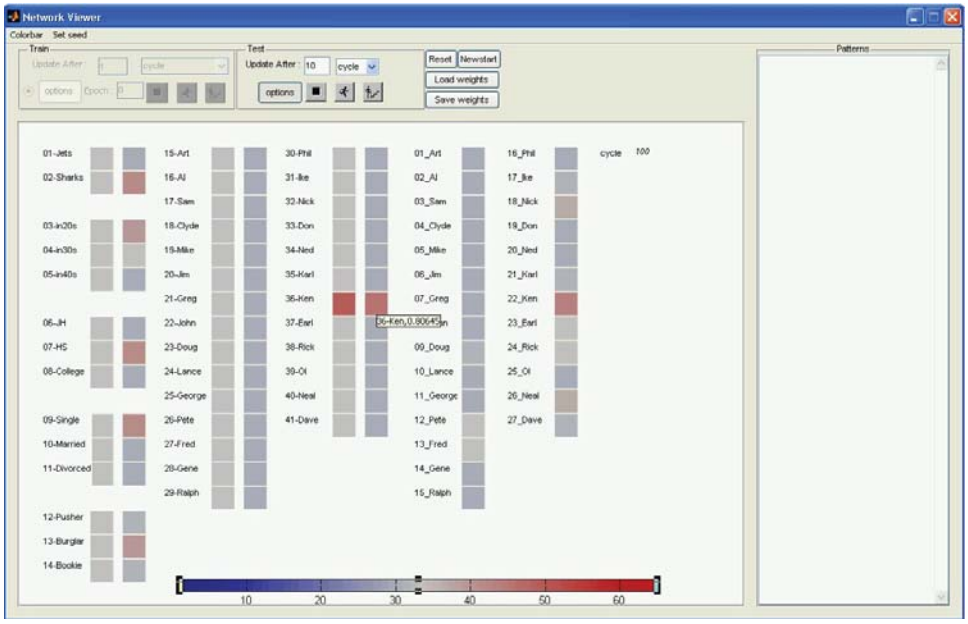

Figure 2.4: The display screen after 100 cycles with external input to the name unit for Ken.

lines do fall on top of each other. In the instance unit panels, you will see one curve that rises above the others, this one for hidden unit *22_Ken*. Explain to yourself why this rises more slowly than the name unit for Ken, shown in the Name panel.

Two variables that you need to understand are the *update after* variable in the test panel and the *ncycles* variable in the *testing options* popup window. The former (update after) tells the program how frequently to update the display while running. The latter (ncycles) tells the program how many cycles to run when you hit run. So, if ncycles is 10 and update after is 1, the program will run 10 cycles when you click the little running man, and will update the display after each cycle. With the above in mind you can now understand what happens when you click the stepping icon. This is just like hitting run except that the program stops after each screen update, so you can see what has changed. To continue, hit the stepping icon again, or hit run and the program will run to the next stopping point (i.e. next number divisible by *ncycles*.

As you will observe, activations continue to change for many cycles of processing. Things slow down gradually, so that after a while not much seems to be happening on each trial. Eventually things just about stop changing. Once you've run 100 cycles, stop and consider these questions.

A picture of the screen after 100 cycles is shown in Figure 2.4. At this point, you can check to see that the model has indeed retrieved the pattern

for Ken correctly. There are also several other things going on that are worth understanding. Try to answer all of the following questions (you'll have to refer to the properties of the individuals, as given in Figure 2.1).

Q.2.1.1.

None of the visible name units other than Ken were activated, yet a few other instance units are active (i.e., their activation is greater than 0). Explain this difference.

Q.2.1.2.

Some of Ken's properties are activated more strongly than others. Why?

Save the activations of all the units for future reference by typing: *saveVis = net.pool(2).activation* and *saveHid = net.pool(3).activation*. Also, save the Figure in a file, through the 'File' menu in the upper left corner of the Figure panel. The contents of the figure will be reset when you reset the network, and it will be useful to have the saved Figure from the first run so you can compare it to the one you get after the next run.

Retrieval from a partial description. Next, we will use the **iac** program to illustrate how it can retrieve an instance from a partial description of its properties. We will continue to use Ken, who, as it happens, can be uniquely described by two properties, *Shark* and *in20s*. Click the reset button in the network window. Make sure all units have input of 0. (You will have to right-click on Ken and set that unit back to 0). Set the external input of the *02-Sharks* unit and the *03-in20s* unit to 1.00. Run a total of 100 cycles again, and take a look at the state of the network.

Q.2.1.3.

Describe the differences between this state and the state after 100 cycles of the previous run, using *savHid* and *savVis* for reference. What are the main differences?

Q.2.1.4.

Explain why the occupation units show partial activations of units other than Ken's occupation, which is Burglar. While being succinct, try to get to the bottom of this, and contrast the current case with the previous case.

Default assignment. Sometimes we do not know something about an individual; for example, we may never have been exposed to the fact that Lance is a Burglar. Yet we are able to give plausible guesses about such missing information. The **iac** program can do this too. Click the reset button in the

network window. Make sure all units have input of 0. Set the external input of *24-Lance* to 1.00. Run for 100 cycles and see what happens. Reset the network and change the connection weight between *10_Lance* and *13-Burglar* to 0. To do that, type the following commands in the main MATLAB command prompt:

$net.pool(3).proj(2).weight(10, 13) = 0;$
$net.pool(2).proj(2).weight(13, 10) = 0;$

Run the network again for 100 cycles and observe what happens.

Q.2.1.5.

> Describe how the model was able to fill in what in this instance turns out to be the correct occupation for Lance. Also, explain why the model tends to activate the *Divorced* unit as well as the *Married* unit

Spontaneous generalization. Now we consider the network's ability to retrieve appropriate generalizations over sets of individuals—that is, its ability to answer questions like "What are Jets like?" or "What are people who are in their 20s and have only a junior high education like?" Click the 'reset' button in the network window. Make sure all units have input of 0. Be sure to reinstall the connections between *13-Burglar* and *10_Lance* (set them back to 1). You can exit and restart the network if you like, or you can use the up arrow key to retrieve the last two commands above and edit them, replacing 0 with 1, as in:

$net.pool(3).proj(2).weight(10, 13) = 1;$

Set the external input of Jets to 1.00. Run the network for 100 cycles and observe what happens. Reset the network and set the external input of Jets back to 0.00. Now, set the input to *in20s* and *JH* to 1.00. Run the network again for 100 cycles; you can ask it to generalize about the people in their 20s with a junior high education by providing external input to the *in20s* and *JH* units.

Q.2.1.6.

> Consider the activations of units in the network after settling for 100 cycles with *Jets* activated and after settling for 100 cycles with *in20s* and *JH* activated. How do the resulting activations compare with the characteristics of individuals who share the specified properties? You will need to consult the data in Figure 2.1 to answer this question.

Now that you have completed all of the exercises discussed above, write a short essay of about 250 words in response to the following question.

Q.2.1.7.

Describe the strengths and weaknesses of the IAC model as a model of retrieval and generalization. How does it compare with other models you are familiar with? What properties do you like, and what properties do you dislike? Are there any general principles you can state about what the model is doing that are useful in gaining an understanding of its behavior?

Ex2.2. Effects of Changes in Parameter Values

In this exercise, we will examine the effects of variations of the parameters *estr*, *alpha*, *gamma*, and *decay* on the behavior of the **iac** program.

Increasing and decreasing the values of the strength parameters. Explore the effects of adjusting all of these parameters proportionally, using the partial description of Ken as probe (that is, providing external input to *Shark* and *in20s*). Click the reset button in the network window. Make sure all units have input of 0. To increase or decrease the network parameters, click on the options button in the network window. This will open a panel with fields for all parameters and their current values. Enter the new value(s) and click 'ok'. To see the effect of changing the parameters, set the external input of in20s and Sharks to 1.00. For each test, run the network til it seems to asymtote, usually around 300 cycles. You can use the graphs to judge this.

Q.2.2.1.

What effects do you observe from decreasing the values of *estr*, *alpha*, *gamma*, and *decay* by a factor of 2? What happens if you set them to twice their original values? See if you can explain what is happening here. For this exercise, you should consider both the asymptotic activations of units, and the time course of activation. What do you expect for these based on the discussion in the "Background" section? What happens to the time course of the activation? Wny?

Relative strength of excitation and inhibition. Return all the parameters to their original values, then explore the effects of varying the value of *gamma* above and below 0.1, again providing external input to the *Sharks* and *in20s* units. Also examine the effects on the completion of Lance's properties from external input to his name, with and without the connections between the instance unit for Lance and the property unit for Burglar.

Q.2.2.2.

Describe the effects of these manipulations and try to characterize their influence on the model's adequacy as a retrieval mechanism.

Ex2.3. Grossberg Variations

Explore the effects of using Grossberg's update rule rather than the default rule used in the IAC model. Click the 'reset' button in the network window. Make sure all units have input of 0. Return all parameters to their original values. If you don't remember them, you can always exit and reload the network from the main pdp window. Click on the options button in the network window and change $actfunction$ from st (Standard) to gr (Grossbergs rule). Click 'ok'. Now redo one or two of the simulations from Ex. 2.1.

Q.2.3.1.

What happens when you repeat some of the simulations suggested in Ex. 2.1 with *gb* mode on? Can these effects be compensated for by adjusting the strengths of any of the parameters? If so, explain why. Do any subtle differences remain, even after compensatory adjustments? If so, describe them.

Hint.

In considering the issue of compensation, you should consider the difference in the way the two update rules handle inhibition and the differential role played by the minimum activation in each update rule.

Ex2.4. Construct Your Own IAC Network

Construct a task that you would find interesting to explore in an IAC network, along with a knowledge base, and explore how well the network does in performing your task. To set up your network, you will need to construct a .net and a .tem file, and you must set the values of the connection weights between the units. *The PDPTool User Guide* provides information on how to do this.

Q.2.4.1.

Describe your task, why it is interesting, your knowledge base, and the experiments you run on it. Discuss the adequacy of the IAC model to do the task you have set it.

Hint.

You might bear in mind if you undertake this exercise that you can specify virtually *any* architecture you want in an IAC network, including architectures involving several layers of units. You might also want to consider the fact that such networks can be used in low-level perceptual tasks, in perceptual mechanisms that involve an interaction of stored knowledge with bottom-up information, as

in the interactive activation model of word perception, in memory tasks, and in many other kinds of tasks. Use your imagination, and you may discover an interesting new application of IAC networks.